Interaction of Hardware Transactional Memory and Microprocessor Microarchitecture

Dissertation
to obtain the academic degree
Doktor rerum naturalium (Dr. rer. nat.)

presented to
Technische Universität Dresden
Faculty of Computer Science

by

Stephan Diestelhorst
`stephan.diestelhorst@gmail.com`

born 12th September 1981 in Halle(Saale), Germany

Supervisor: Prof.Christof Fetzer

Defense: 28th March 2019

Cambridge, 9th June 2019

[1] https://www.flickr.com/photos/130561288@N04/47049179901

Contents

List of Figures

List of Tables

Acknowledgements

They say that “No man is an island”. Similarly, no doctoral thesis surfaces in the sea of knowledge due to the efforts of just one author; in particular not this one. Instead, I have had lots of help, and I am very grateful to those who helped in the various stages of this work.

My thesis advisers and reviewers–Christof Fetzer, Paolo Romano, Pascal Felber–for guidance, experience, and expert knowledge of the field. The examination committee–Christel Baier, Sebastian Rudolph, Hermann Haertig–for making time, reviewing my work, and keeping me on my toes with excellent questions.

My colleagues, collaborators, and co-authors at Arm, AMD, TU Dresden, University of Neuchâtel, Barcelona Supercomputing Centre (BSC), and the VELOX project.

Especially (at Arm): the Memory & Systems group–Andreas Hansson, Andreas Sandberg, Ciro Santilli, Giacomo Travaglini, Ilias Vougioukas, Jonathan Beard, Juha Jaykka, Nikos Nikoleris, Prakash Ramrakhyani, Radhika Jagtap, Wendy Elsasser, William Wang–what a pleasure to work with all of you; Nigel Paver for allowing me to take time off work to finish this; Matt Horsnell for many excellent HTM follow-on discussions; James Myers for endless amounts of constructive positivity; and Lucy Gonzalez for keeping me sane (and caffeinated).

Especially (at AMD): Michael Hochmuth, Martin Pohlack, David Christie–for setting me up for success, endless deep dives on HTM, CPU architecture, and tons of experience; Sherry Hurwitz–for being the best manager one can hope to have.

Especially (at TU Dresden): Martin Nowack–for an endless supply of goofiness, enthusiasm, and compiler insights; Torvald Riegel–for great technical discussions.

Most importantly, however, my support group: Antje, mum, dad.

You all rock, many many thanks!

Chapter 1

Introduction

1.1 Context

Microprocessor developments In their 47 years of existence, microprocessors have transformed the way we deal with information. Thanks to exponential growth in the number of transistors that can be integrated per square area, caused by shrinking feature sizes of the respective transistors (observed and postulated by Gordon Moore from Intel and dubbed “Moore’s law”), and related switching speed increases, the resulting performance (executed instructions per unit time) also increased exponentially. As a result, complex, large applications run on many layers of abstraction to provide computation power for business and recreational use; while at the same time, simpler micro-controllers power the logic in embedded devices, such as white goods. The exponential performance growth basically causes a price reduction per unit of embedded “intelligence”.

Initially, the shrinking feature sizes also allowed faster switching of the transistor elements and reduced transmission times (thanks to reduced RC effects) on wires between them. Eventually, *wire scaling* stopped, and microprocessor designs turned to shallower logic trees through the use of *pipelining* to increase overall clock speed and instruction throughput.

Clock speed was the main driver for performance, especially in the PC market, with a fierce marketing battle for reaching specific frequency targets–causing the design of longer pipelines with fewer work being done per pipeline stage. Eventually in the early 2000s, however, *voltage scaling* stopped and with it the property that power per chip area remained constant (by reducing power per transistor, known as Dennard scaling). While transistors were still getting more plentiful, and switched faster, their overall power consumption did not reduce exponentially anymore. Microprocessors hit the *TDP wall*, because their thermal design power reached the levels that could be dissipated economically.

Instead of relying solely on increasing throughput by increasing clock frequency and pipeline depth, new designs would execute more instructions in parallel; using parallelism in the instruction stream (ILP) and the memory system (MLP) through sophisticated use of transistors for speculation logic (branch prediction), multiple execution units (super-scalar execution, vector execution), and instruction scheduling (out-of-order execution)–relying on parallelism in a single stream of instructions to provide higher performance with more transistors.

Eventually, however, these techniques became more expensive for the performance they were unlocking (due to parallelism becoming harder and harder to extract); instead, commercial microprocessors switched to *multi-core solutions*, where increasing transistor count is predominantly used for replicating the core so that application, thread, and task level parallelism are used to increase overall performance per microprocessor chip.

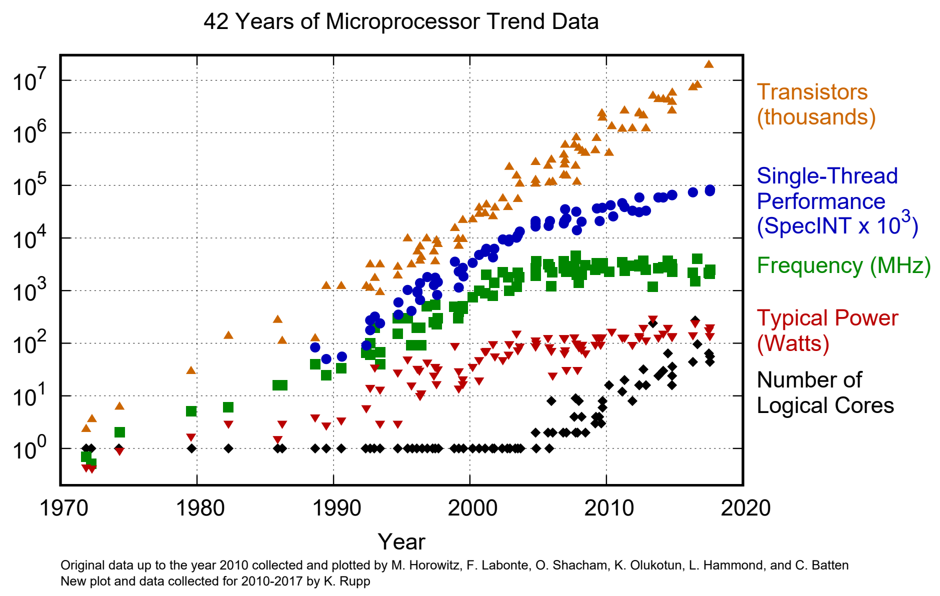


Figure 1.1: Microprocessor trends since 1971 showing transistor growth, levelling single-thread performance, frequency, and TDP; and recent increases in the number of logical cores per chip. From [381].

Figure 1.1 shows the overall trends for performance, frequency, core count, and TDP for microprocessors of the last decades; while Figure 1.2 provides a more detailed look into the (normalised) SPECint and SPECfp performance of all systems listed on the SPEC web page since 1995.

Application impact As a result of these changes, applications need to adapt to new constructs exposing parallelism: wider vector extensions for data-level parallelism in numerical computations, and splitting of applications into multiple tasks / threads / processes to make use of modern multi-core CPUs.

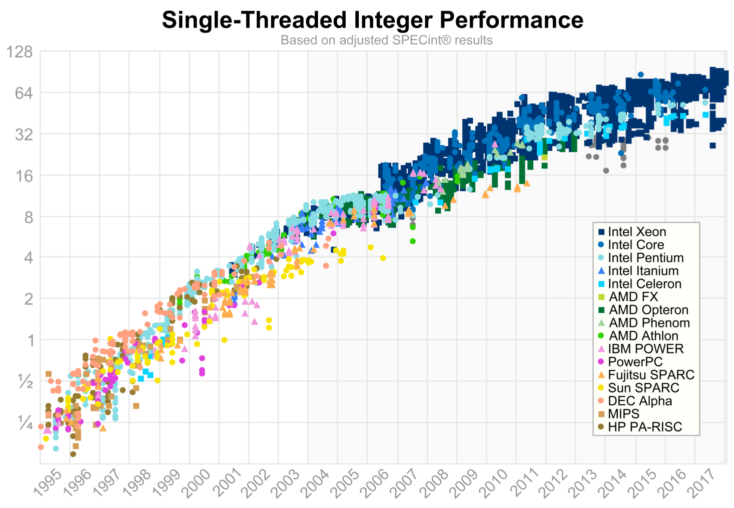
Parallelising applications is often challenging; with a key part of that being coordinating the access to shared state, in particular memory that is being shared between tasks or threads. While it is possible to completely partition the application logic and state in some cases, many other cases benefit from shared data between concurrently executing instruction streams. Sharing reduces data duplication and transfer; if data is shared read-only, no coordination is necessary. In many cases, however, data is only *mostly* read-shared and sometimes manipulated. In those cases, coordination among the concurrent threads is important.

1.2 Transactional Memory

Transactional memory (TM) is a mechanism that controls access to shared memory by concurrently executing threads, and provides other desirable properties, such as failure atomicity. This dissertation will, however, focus on the usage of TM as a method for concurrency control.

There are effectively two ways of controlling access to shared resources: *pessimistic* concurrency control ensures that concurrent threads that could potentially conflict will coordinate among one another. For that, locks are frequently used and they often (safely) approximate the area they protect and the operations performed; causing more complicated use and higher overhead when more fine-grained tracking is required. As a result, locks often overly limit the amount of parallelism in an application; while that may not be a problem for small core / thread counts, this will limit the amount of speed-up enjoyed on future systems with more available cores (Amdahl's law).

Optimistic concurrency control, on the other hand, structures applications so that they assume free-



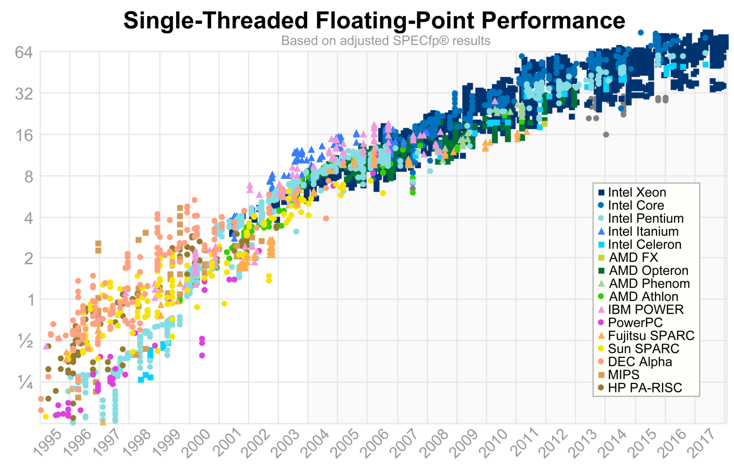


Figure 1.2: Historical adjusted SPECint (top) and SPECfp (bottom) trends for various microprocessors. Updated from [277].

dom of interference and have a backup mechanism in case that assumption did not hold. Because of the intricate reasoning and the low-level of abstraction provided by ISAs in general (single location load / store instructions, single location atomic read-modify-write), mechanisms such as lock-free programming are, however, extremely complicated to devise (top-tier conference paper for lock-free algorithm for a single data structure) and thus not scalable for wide deployment by programmers.

Transactional memory raises the level of programmer abstraction and simplifies optimistic control. It allows multiple instructions (especially memory accesses) to execute as one isolated unit; similar to database transactions. Devised in the mid-1990s, transactional memory was only looked at more closely in the early 2000s when it became clear that multi-core systems would become mainstream and thus concurrency control would, too. By reasoning about the atomicity and isolation on a block granularity, transactional memory allows programmers to write complex concurrent algorithms with optimistic concurrency control easily. Subsequently, from 2004 to 2010, academic and industry interest in TM rose significantly (more details in Chapter 2).

Using TM for concurrency control exhibits higher performance than pessimistic locking if actual data conflicts are rare; furthermore, programming with TM gives several benefits of lock-free data structures (for HTMs and lock-free STMs); and provides failure atomicity by allowing roll-back to a known good state. This was the outset for my Master's thesis [159] at AMD in 2006–2007, where I evaluated a very simplified version of a hardware TM proposal.

While the ASF 1 proposal that I investigated was rather simplistic, its evaluation in a full-system, detailed CPU simulator, and level of detail in the ISA description were not. Many pieces of academic work studied feature-rich extensions to the original HTM proposals, with deep changes to core and memory system architecture; yet evaluated those in simplistic (simple in-order core model, fixed one instruction per cycle, lack of OS interactions) simulation environments. Furthermore, due to having to change applications for HTM, many available applications were only small micro-benchmarks.

In that context, several partners with diverse backgrounds and AMD started the EU-funded FP7 VELOX project to study the full stack from application to core micro-architecture for transactional memory. Within that project, AMD's work was to provide a realistic ISA extension for HTM, and a performance model to evaluate it. Together with my colleague Michael Hohmuth and Dave Christie, we translated our experience from the ASF 1 work and built a new ASF 2 HTM ISA spec. I extended the PTLsim (and later Marss86) simulator to model ASF 2's performance realistically. Later, with colleagues Martin Pohlack, Luke Yen, and Jae Woong Chung, we looked at further microarchitecture implementations and extensions, and extensions to the ISA of HTM. Together with colleagues from TU Dresden Torvald Riegel, Martin Nowack, Diogo Becker, Jons Wamhoff, and Christof Fetzer, we refined the low-level application / compiler and ISA interface. Finally, Anurag Negi from Chalmers worked with us on simulator integration and advanced cache design and microarchitecture.

1.3 Thesis

The central thesis of my work is that detailed analysis and ISA modelling of HTM is necessary to understand actual implementation and usage challenges, and get more realistic results. Instead of overly complicating the design of HTM with features that would be extremely hard to implement right in a more detailed microarchitecture and ISA proposal, I suggest that getting a base-line HTM specification and micro-architecture right is a challenge in itself. Yet, despite the complexity, there are interesting implementation options and extensions that can provide benefits to applications using HTM–but they are not on the trajectory taken by most papers published between 2004 and 2010.

1.4 Contributions

When I started my work on transactional memory (starting with my initial investigations during my Master's thesis [159]) in 2006, interest in transactional memory was rising; see Figure 2.2. Reviewing the timeline of my contributions, most of them were made in 2009 / 2010, coinciding with the peak of TM publications. This was largely driven by the time needed to bring up the necessary infrastructure, both in AMD, but also in the VELOX project. In the project, the compiler and TM library work needed time to ripen; whereas in AMD, we carefully designed the ISA structure, and I spent a significant amount of time on a very detailed simulator implementation including a lot of time spent on repairing and extending baseline simulator functionality.

In a second wave, in around 2012, we investigated several higher-level concerns, such as observing passage of time from inside transactions, and communicating from inside transactions with parallel nesting.

Finally, the AMD Research office closed by the end of 2012, and I ported our transactional memory framework to a more advanced simulator at TU Dresden, which ultimately culminated in our publication on resurrecting aborted transactions in 2013 (brief announcement) and 2015 (full paper).

In comparison to the field (more detail in Chapter 2), I believe that my work was starting a bit later, and I missed some of the early wild invention phase, especially on the hardware side. In hindsight, however, this allowed us to put together a much more realistic baseline TM architecture, both for hardware and compiler quality; a less feature-loaded TM specification, which we subsequently extended to support more advanced use cases; and observe other challenges when using TM.

My contributions to the state-of-the-art summarised in this thesis report are the following:

Simulation and micro-architecture To get accurate results for HTM evaluation, a good baseline simulation platform is important. I have improved the stability and increased the functionality of the PTLsim and Marss86 simulators: adding coherence, repairing core timing models, bank conflicts, general crashes, deadlocks in the coherence protocol [135, 253, 262, 304]. I have served as the maintainer for PTLsim, after the original maintainer stepped down.

HTM baseline ISA The ASF ISA is an HTM proposal that is different from other HTM proposals: it offers non-transactional memory accesses, does not snapshot registers at transaction entry, exposes exceptions from inside the transactions, and provides a minimal capacity / obstruction-free progress guarantee [186] (attached in Appendix A). Because of its different features, we had to define new interactions (ordering of transactional and non-transactional memory accesses, exception models, overlaps between transactional and non-transactional accesses); and because of the industrial background, the specification has to deal with all use cases (usage inside the OS) and real-world deployment constraints (no additional register footprint for compatibility).

Microarchitecture for HTM In conjunction with defining the ISA, I implemented the ASF specification in a detailed, full-system, out-of-order CPU core model and detailed cache model. From that implementation experience, I was able to not just evaluate performance characteristics, but also understand feature complexities and derive understanding for handling undefined ISA corner cases such that the hardware would not get too complex. In comparison, most related works use simplified in-order, one instruction-per-cycle models.

Application work While most application, compiler, and library development work in the VELOX project was performed outside of AMD (mainly at TU Dresden and University of Neuchatel), I helped debug low-level correctness issues, and helped translate software requirements back into ISA and microarchitecture changes (overlapping non-transactional / transactional accesses, ordering of aborts and non-transactional atomic read-modify-write instructions). Furthermore, I added support for ASF to the Oracle Hotspot Java just-in-time compiler / virtual machine to execute `synchronized` blocks with transactions [290]. Hotspot is a production-level commercial JIT / JVM and integrating ASF required extensive modifications to highly optimised, multi-path locking code.

Extensions and challenges In addition to the definition and evaluation of an industry-grade HTM, we investigated extensions to the ISA and microarchitecture of HTM, and identified challenges when using HTM for transparently eliding locks. Annotating memory accesses to hint cyclical value behaviour and lazy versioning, and transactional "resurrection" are an example of the former, while dealing with timestamp anomalies when eliding is an example of the latter [221, 263, 289, 337].

Thanks to the great collaboration inside of AMD, inside the VELOX project (especially with TU Dresden and University of Neuchatel), and other researchers (Michael Spear, Lehigh University), a significant amount of my work has been reflected in publications. Those publications are:

- Hardware Acceleration for Lock-Free Data Structures and Software-Transactional Memory (EPHAM 2008 [158], Appendix B.1)
- ASF: AMD64 Extension for Lock-free Data Structures and Transactional Memory (MICRO 2010 [214])
- Evaluation of AMD's Advanced Synchronization Facility Within a Complete Transactional Memory Stack (EuroSys 2010 [213])
- The Velox Transactional Memory Stack (IEEE Micro Journal [210])
- Implementing AMD's Advanced Synchronization Facility in an Out-of-Order x86 Core (TRANSACT 2010 [220], Appendix B.2)
- Compilation of Thoughts about AMD Advanced Synchronization Facility and First-Generation Hardware Transactional Memory Support (TRANSACT 2010 [215], Appendix B.3)
- Sane Semantics of Best-effort Hardware Transactional Memory (WTTM 2010 [221], Appendix B.5)
- From Lightweight Hardware Transactional Memory to Lightweight Lock Elision (TRANSACT 2011 [254], Appendix B.4)
- Delegation and Nesting in Best Effort Hardware Transactional Memory (SPAA 2012 [274])
- Safely Accessing Time Stamps in Transactions (WTTM 2012 [263], Appendix B.6)
- Between All and Nothing–Versatile Aborts in Hardware Transactional Memory (SPAA 2013 and TRANSACT 2015 [289, 337], Appendix B.7)

Although I am not the first author of some of these publications, I have been significantly contributing to the microarchitectural implementation ideas, ISA design decisions and evaluations of AMD's Advanced Synchronization Facility feature. Beside the attached publications, my contributions are available in source code form in the two (public) prototype implementations: PTLsim/ASF and Marss86/ASF, and the extension to the Oracle Hotspot JVM [262, 290, 304]. I have undertaken additional implementation and analysis work, but was unable to publish that before AMD decided to close the Dresden office and let go all employees, including myself, by the end of 2012.

The vast amount of granted patents and patent applications (19 granted patents and 27 applications to date) relating to ASF and extensions to it under my name gives an insight into additional topics analysed in a proprietary setting.

1.5 Terminology

During the course of the creation, the terminology in the research area and in my various technical groups at both universities and microprocessor vendors has changed. Some terms are thus used differently both spatially and temporally, a quick map highlighting synonyms is as follows:

“Snoop”, “probe” . . . messages in a coherent system to notify participants of reads, writes and other operations;

“Speculative region”, “transaction” . . . unit of work that needs to be executed atomically wrt. code running on other CPU cores / thread. “Speculative region” is the more AMD ASF-specific term and “transaction” the standard academic term; “speculative” and “transactional” memory access are derived where the context of speculative is usually clear.

“Barrier”, “(memory) fence” . . . instruction or compiler directive to ensure ordering between memory accesses; not to be confused with thread barriers (waiting for all threads to arrive at a specific point in the code flow before continuing), or read / write barriers in Software Transactional Memory and Garbage Collection, where single read / write operations are augmented with additional logic for locking / marking.

“ASF”, “BeHTM” . . . AMD’s Advanced Synchronization Facility (ASF) is an industry instruction set extension proposal for an extended transactional memory primitive. ASF offers a limited set of guarantees regarding the size and progress of transactions, and also provides additional non-transactional operations (see Chapter 3). Best-effort Hardware Transactional Memory (BeHTM) describes transactional memory implementations with few or no assertions regarding supported transaction sizes and types; thus lending to simpler hardware implementations. As such, ASF is in instance of a BeHTM with additional guarantees.

“Protected”, “transactional” . . . Memory accesses that are treated specially in the transaction / speculative region, i.e., providing both conflict detection and data versioning. In simple BeHTMs, all accesses are of this kind; ASF allows marking of transactional / non-transactional accesses.

“in-tx”, “non-tx”, “out-tx” memory accesses . . . Memory accesses can bei either inside a transaction (in-tx), or outside of a transaction (out-tx). For those that are inside a transaction, in most BeHTMs accesses are transactional (tx); they participate in conflict detection and versioning. In some BeHTMs (such as ASF), one can disable the transactional properties of in-tx accesses and make them non-transactional (non-tx).

1.6 Outline

The rest of the thesis is organised as follows: Chapter 2 will present an overview of related work in the areas of transactional memory and simulation, and provide a background for microprocessor design. Chapter 3 will show architectural features of hardware transactional memory and present our proposal ASF; and Chapter 4 will show various implementation options and trade-offs between them. Chapter 5 focusses on the application and evaluation of ASF; with performance results, software usage options, and a detailed walk-through through the simulator implementations of ASF. Chapter 6 will extend use cases of ASF by implementing communication channels between transactions and a mechanism to resurrect aborted transactions; while Chapter 7 will provide further challenges, and implementation sketches for

HTM, together with some HTM product experiences. Finally, Chapter 8 will summarise my work, my experiences, and conclude the thesis. Appendix A contains a copy of the ASF specification, and Appendix B papers that are not in formal proceedings.

The goal is to provide a single, self-contained collection of the work that is encompassed by this dissertation. Several of these documents have moved since their creation, and websites have stopped being available. I think it is therefore valid and useful to provide these copies verbatim. Of course, copyrights apply: where applicable, these are author's copies of the papers and should be treated as such.

The remainder of this thesis will, however, focus on providing a structured presentation of the published work in conferences, journals and high-impact, refereed workshops. I will provide additions and clarifications, and also provide background and identify common themes to allow proper placement of the respective publications.

Chapter 2

Overview and Related-Work

2.1 Introduction

Many facets of transactional memory have been researched and published about. In this chapter, I will present an overview of relevant transactional memory research work, and also provide background material that underpins the hardware aspects (modern microprocessor basics, simulation methodology) of my thesis.

Of course, transactional memory does not exist in a vacuum; instead, it is a step in a series of synchronisation techniques which interestingly dates back many decades; and its key components (as *speculative synchronisation*) have been proposed about thirty years ago.

The development of TM did, however, get a significant boost with changes in microprocessor performance scaling in the early 2000s.

The remainder of this section will provide more background for each of these aspects; before diving into the review of the many related pieces of research in following sections. While structuring that review, I have found that putting all these papers into a single ontology is not easy; many papers combine aspects from across the stack: microarchitecture, ISA, application behaviour and support, and conceptual points. In some cases, I separate aspects of a single paper under multiple headlines, yet I mostly aim to group papers around their key contributions.

2.1.1 Basics and Beginnings

Atomic operations for synchronisation date back to the early 1960s (with the Burroughs B5000 `RDLK` instruction [9]) and as early as 1978 in the x86 ISA (with the Intel `LOCK` prefix being introduced with the 8086 [6]). These come in different varieties and strengths, and it can be shown that the strongest ones (such as compare-and-swap) are generic in that every concurrent algorithm can be mapped to them [19]. Typical examples are single instructions, such as *compare-and-swap*, *load-op-store*, and multi-instruction sequences modelled after *load-linked / store-conditional* [18]. All these instructions operate on a single word in memory, but as shown above, this is not limiting in the algorithmic sense. The key insight for all of them is that one can perform a load-op-store sequence without any intervening stores from other cores. Practically, however, multiple ISAs [257, 258] have provided additional primitives, for example double-wide compare-and-swap which enables operation on pointer with added timestamps, or full double CAS operating on two distnct addresses (DCAS) [24].

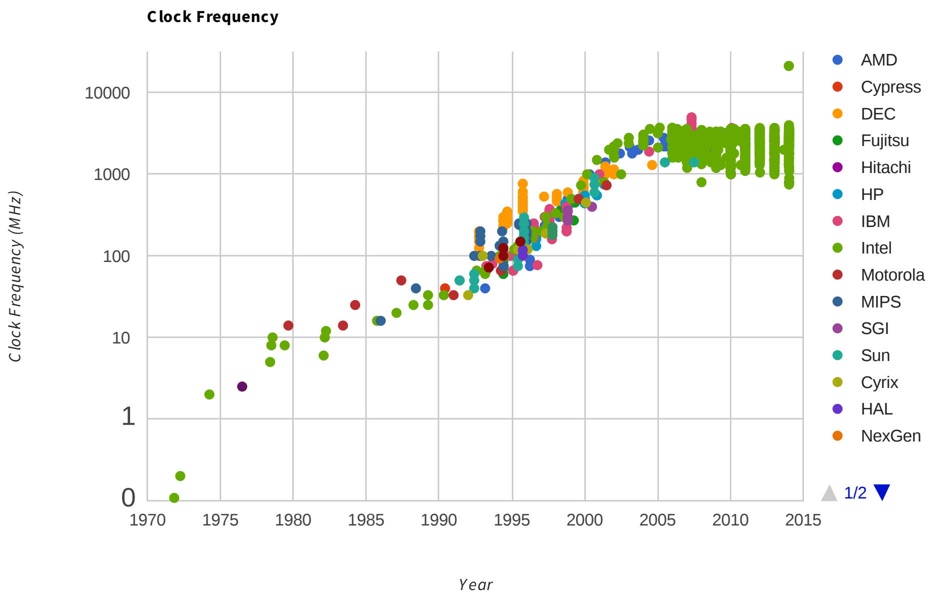


Figure 2.1: Scaling of CPU clock frequency showing a clear stall from around 2002. Data from CPU DB. [283]

2.1.2 Speculative Synchronisation

Knight proposed in 1986 a programming model and hardware architecture [16] that breaks code into transactional blocks which are comprised of a side-effect-free prefix (loads, computation), and end with a single store to memory. In the proposed hardware, the CPU executes these *concurrently*, tracks dependencies between the prefixes and stores, detects conflicts, and re-executes those prefixes that have executed but were invalidated due to stores overwriting the data read. A block counter is used to schedule the execution and completion of the blocks, and a special cache is used to buffer side-effecting stores speculatively so that they can be executed before the block is scheduled – a first example for speculative synchronisation, conflict detection, and atomic blocks.

In 1993, two groups experimented with extending the number of memory locations that can be operated on in with atomic instructions: the Oklahoma Update protocol [35] from IBM, and Transactional Memory (TM) [34] proposed by DEC / University of Massachusetts provide multi-word atomic primitives. Both add a special buffer (reservation buffer / transactional cache) that is used for both conflict detection and to buffer / make visible the multiple stores. Furthermore, both proposals require special instructions to mark the special accesses, and by default convey failure to hold on to the reservation at the end of the sequence when changes are to be committed. The Oklahoma Update paper adds on top a per-location *immediate abort* specifier which lets the CPU jump to a specific handler if that memory location was conflicted upon. The proposal also guarantees progress (and deadlock-freedom) by sorting written-to locations and locking them. Transactional Memory, on the other hand contains ingenious re-use of the transactional cache after the instruction sequence is completed; which allows very fast aborts and commits without having to write data back to the memory hierarchy.

A few years later, the first proof-of-concept showed a software solution – Software Transactional Memory (STM) – that provided a similar abstraction on top of normal single-address atomic instructions [39].

It took, however, until the obvious “death” of CPU frequency scaling in 2002 (quite dramatically visible in Figure 2.1) and the associated stagnation of single threaded performance, to spark renewed interest in techniques to accelerate and simplify concurrent programming. Transactional memory was seen as a possible, mechanical way to convert sequential data structures and algorithms into parallel versions that could side-step stalling single thread performance and instead make use of the rising number of cores in

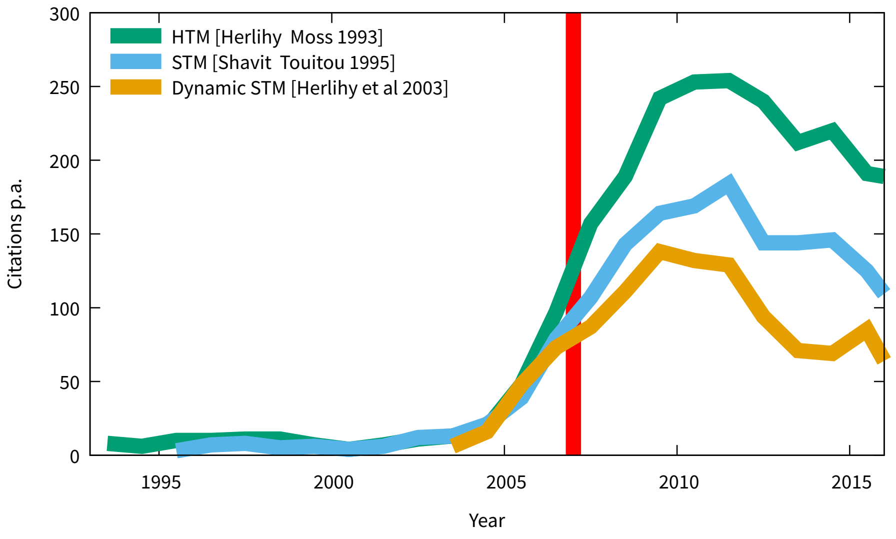


Figure 2.2: Citation count of seminal transactional memory papers [34, 39, 68]. Marked start of my work on the subject. Data from Google Scholar.

commodity SMP (Symmetric Multi-Processor) systems.

2.1.3 Creative Explosion

Academic interest in transactional memory (and other technologies that exploit the growing number of CPU cores) rose almost proportionally to the growing disparity between previous frequency scaling and the significant stall of actual CPU frequencies. Figure 2.2 shows the academic interest by virtue of citation count of three early and seminal publications of transactional memory [34, 39, 68].

Despite all early papers suggesting hardware changes to enable transactional memory, Shavit, Touitou, and Moir showed early on (1995, 1997) that software solutions were possible, as well [39, 44]. Both solutions are similar in that they provide / utilise an n-CAS operation and provide non-blocking progress guarantees. While an improvement over earlier universal constructions [28], the former solution only permits concurrency when accesses do not overlap, requires a static working set, and requires unrealistic low-level instruction primitives (nested LL/SC); while the latter employs indirections to support dynamic discovery (with bounded transaction sizes) and concurrent read-sharing.

In the early 2000s, both hardware and software solutions for transactional memory (HTM / STM), received renewed interest due to the rise of the multi-core CPUs (in turn fuelled by the lack of single-core frequency scaling). In the following sections, I will present in broad strokes the key developments; focussing in more detail on hardware. A much more complete summary is available elsewhere [224].

2.1.4 Organisation

The remainder of this chapter is organised as follows: Section 2.2 will highlight research work for Software, Hardware, and Hybrid Transactional Memory proposals respectively which were part of the "creative explosion". Section 2.3 will contrast this with actual industry specifications and showcase available chips that implement TM. Section 2.4 will show use cases for TM and how the existing software stack is modified to accommodate them. Section 2.5 will will explain the background and key developments in CPU microarchitecture in Subsection 2.5.1, and Subsection 2.5.2 will highlight tools that are used to simulate computer systems.

2.2 Transactional Memory Research

2.2.1 Software Transactional Memory

Weakened guarantees and dynamic data layouts The “creative explosion” of software transactional memory papers starts in 2003 with Herlihy, Luchangco, Moir, and Scherer’s STM for Dynamically-Sized Data Structures [68]. The authors weaken the progress guarantees from wait-free / lock-free to *obstruction-free*, eliminating helping altogether and in turn allowing fully dynamic discovery of the working sets of the transaction. They also add *contention managers* which decide which transaction to abort and when to retry them. The usage of contention managers provides practical progress, as obstruction-freedom in itself is prone to livelock. Another novelty of DSTM is the usage of object-granularity at tracking conflicts; objects can be opened for reading or writing, and also upgraded later on. Furthermore, transactions may remove entries from their read sets. Finally, DSTM uses *validation* to make sure that all transactions observe a consistent view. Using a double indirection in transactional objects, transactions can create a new tentative object and switch versions instantly with a single atomic change of the writing transactions status register. The results, however, of this paper seem mostly negative: none of the proposed algorithms scale, and they have a significant overhead (about a factor of 10x) compared to simple locking for a single thread. On closer inspection, however, this is partially caused by performing 100% updates on a small data structure.

Harris and Fraser are the first to implement a dynamic STM on memory word granularity, rather than objects [66]. They introduce the concept of *ownership records* (orecs) that allows tracking status of the words on the heap. In addition, both reads and writes are added to the local transaction descriptor (a read and write log). Writes will acquire ownership at the time of commit, at which conflicts between transactions will be detected. They provide non-blocking progress by implementing helping in the commit phase. One interesting addition is the notion of a *wait* primitive which aborts the transaction, but will only allow restart once entries of the working set are changed.

In 2005, Marathe, et al, categorise, analyse, and formalise the timing of conflict detection, versioning, progress guarantees (obstruction / lock freedom), direct / indirect access and per-object / per-transaction meta-data placement [89]. Versioning defines where transactional stores live while the transaction is still ongoing. *Eager versioning* stores the speculative copy in the location of the actual data as soon as the write happens; *lazy* versioning keeps the store in a separate location until the transaction commits. Similarly, *eager conflict detection* checks for conflicts as soon as transactional loads and stores happen; *lazy* conflict detection defers such checks until the transaction is about to commit.

Weaker progress and removed overheads Weakening the progress guarantees from wait-free, to lock-free, and obstruction-free allows for reduced levels of indirection, greater algorithmic flexibility, and ultimately higher performance. Consequently, in 2006, in a controversial publication [117], Robert Ennals proposed to drop the non-blocking progress guarantees altogether, and proposes a fast, object-based STM that removes indirections and subsequently achieves higher performance.

Saha, et al, also publish about their non-blocking STM, and show the cost of different STM design decisions in McRT-STM [114]. They argue for careful transaction scheduling rather than non-blocking design, show the cost of visible reads to be high (10x), and also show that undo logging outperforms write buffering significantly.

Indeed, the introduction of time-based STMs with invisible readers, such as TL2 [116], LSA-TM [103], and TinySTM [160], showed significant performance improvements due to reduced numbers of *validation* operations and reduced work for transactional reads. These time-based STMs use a logical (commit count) clock which can be a point of significant contention in itself. Ruan, et al, show benefits when

```
1  void privatise (..) {
2    node *my_node = NULL;
3    // Dequeue from shared list
4    tx {
5      my_node = root;
6      root = my_node->next;
7    }
8    // Work outside of TX
9    my_node->data += 42;
10 }
11
12 void publish (..) {
13   // Create a new node
14   node *my_node = malloc(..);
15   my_node->data = 0xCAFFE;
16   // Publish node
17   tx {
18     my_node->next = root;
19     root = my_node;
20   }
21 }
```

```
25 void inplace_upd(..) {
26   tx {
27     // Process list in TX
28     node *c = root;
29     while (c != NULL) {
30       c->data += 17;
31       c = c->next;
32     }
33   }
34 }
```

Figure 2.3: Examples of privatisation and publication of memory showing both accesses from inside and outside of a transaction to the same data. In this example, the non-transactional access in line 9 could observe clean-up activity of an aborting, concurrent tread executing the transaction in lines 2 – 9 (privatisation safety failure). The failure mode of publication safety is more subtle.

using the hardware time-stamp counter (TSC) instead and show good benefits especially for small transactions [294].

Finally, a significant shift in STM occurred with STMs that abolish the orecs, and instead use *value-based validation* and serialise the commit phase of writing transactions. Notable examples of such designs are the aptly named NOrec [217], and transactional mutex locks [218].

Coordinating transactions and non-transactional accesses One complexity of STMs is the interaction with code that accesses the same data both inside and outside of transactions. The STM relies on instrumenting all reads / writes inside a transaction with code that performs the checking, versioning, etc. (also, somewhat confusingly, called *read / write barriers*); sometimes accesses inside transactions are not instrumented (especially when performing manual instrumentation) if they are to thread-local data, thus reducing the significant overhead of STMs. Clearly, *concurrently* accessing objects both from inside and outside of a transaction is a sign of an improperly synchronised application. However, careful hand-off can allow such accesses between non-transactional code and a running, but doomed transaction or post-abort clean-up code: privatisation and publication transactions control the global visibility, Figure 2.3 shows some example code. In privatisation, a transaction is used to remove an object from a global data structure and thus allow the removing thread to operate on the object without additional synchronisation. Publication reverses the process: a private object is added to a globally visible data structure through a transaction, for example, enqueuing a locally created node to a linked list.

Despite these idioms looking well formed, they are not properly supported in some STMs. The problem is caused by the delays between logical commits / aborts and the data being updated in the actual location referenced directly by the non-transactional access. Spear, et al, provide a comprehensive treatment of the subject and analyse the performance impacts of the different techniques [147], further refined by Marathe, et al [164]. Generally, STMs that provide *privatisation safety* need to make sure that all transactions that could still access a privatised object have completed (committed or aborted), before the privatised object can be accessed without transactional mechanisms. A similar problem exists in the read-copy-update (RCU) mechanism in the Linux kernel [47].

Underlying the privatisation problem is the incomplete conflict detection and versioning between STM accesses and accesses outside of transactions. This characteristic is referenced as *weak isolation*, and systems that provide proper isolation between transactions and non-transactional accesses are *strongly isolating* [82]. With strong isolation, privatisation is not a problem as the non-transactional accesses to the privatised object will properly interact with any transactions that might still have a reference to that object.

In a similar vein, there has been discussion about how transactions should actually synchronise the application around them. Among transactions, *linearisability* or the equivalent *strict serialisability* remain the obvious correctness criteria. With respect to other code, however, various semantics exist. Arguably the most intuitive model, *single lock atomicity* (SLA), is not easy to provide: (1) it enforces publication and privatisation safety, and (2) enforces certain order with empty / non-overlapping transactions. Both Spear, et al, and Menon, et al, discuss possible semantics and their trade-offs [165, 182].

Tuning and managing contention The different described STM implementation ideas each work well for specific transaction sizes, scalability levels, and expected contention. Depending on the actual characteristics of the application, selecting the right algorithm can have significant performance impact. Even with algorithms from a single family, the placement and number of orecs for the used transactional data can have a huge impact on performance. Felber, et al, present an analysis for run-time tuning of the used hash functions and aggregation of orecs for less memory traffic [160]. Marathe, et al propose partially invisibility tuning to adjust the cost / benefits of visible and invisible reads according to the transactional workloads [164]. Wamhoff, et all, generate several possible code paths to dynamically deal with varying contention levels: for low thread counts they use a very lightly instrumented master thread that is assisted by asymmetrically instrumented helpers [300]. Further tuning options are explored in [170, 232].

When two transactions conflict, only one of them can continue to make progress. The other one either has to wait, or abort and retry. The *contention manager* and *contention policy* are responsible for deciding which of the two conflicting transactions perform which operation in case of conflict and when and how often to retry after an abort. Contention management policies can be simple, for example *requester-wins* where the transaction accessing the conflicting entry wins, or complex, for example *Eruption* which transitively tracks progress of transactions and gives higher priority to those transactions that are stalling a larger number of dependent transactions [87].

Further tuning happened with the release of HTM implemented in silicon; more detail for those can be found in Section 2.4.5.

2.2.2 Lock Elision and Thread-Level Speculation

Following the proposal from Herlihy and Moss [34] in 1993, academic computer architecture research focussed on enabling *thread-level speculation*, a technique that extracts parallelism from sequential programs, especially through parallelising the execution of multiple loop iterations in parallel [48, 49, 51, 52, 54, 55].

Eventually, instead of extracting parallelism from sequential applications, multiple researchers concurrently investigated the usage of speculation hardware for executing lock-based critical sections concurrently, and converting them into (non-blocking) transactions. Rajwar and Goodman propose to detect lock acquisition and release operations in the hardware, without changing the ISA of the underlying CPU [58]. They use prediction logic to learn if a specific *atomic* operation (they use load-linked / store-conditional) is in fact a lock acquire operation, and if so, whether the lock acquisition operation can be *elided*–in turn coining the term *lock elision*. Instead of acquiring the lock, the CPU switches into transactional execution and only adds the lock variable to its read set; that way, multiple critical sections can execute concurrently if there is no data conflict. The transaction ends if the CPU detects a *silent* store that will return the lock into the free state. The invisible approach of Rajwar and Goodman has complications: in my experience, several types of locks will be rather elusive to such an approach. Ticket spin-locks, for example do not return to the original state, but instead both acquire and release operations involve an increment operation. Another issue is self-inspecting code where a critical section inspects (for example in an assertion) whether the lock has been properly acquired. Finally, more elaborate locking algorithms have symmetric acquire / release operations, making it hard to distinguish between "inside" and "outside", i.e., may be prone to elision of the wrong part of the application. On a microarchitectural level, Rajwar and Goodman's proposal extends that of Herlihy and Moss and makes it more practical: transactional conflicts are detected by adding mark bits to the L1 data cache, whereas data versioning is performed in the store queue that sits between the core and the L1 data cache. They however oversimplify the commit logic, by allowing the cache to point to said buffer to make all content of that buffer visible in one atomic operation.

Martinez and Torrellas propose a different, ISA-visible approach with significant benefits and additional use cases [64]. They approach from a TLS perspective and declare one thread as being *safe*– not prone to aborts or stalls. In the example, this thread is the one that would follow the correctly synchronised execution, for example the thread that actually acquires the lock. Because of the safe thread, their scheme guarantees progress, but also needs cooperation from the application: adding the lock variable to the read set of every transaction would not suffice as threads will acquire the lock and thus cause conflicts with other speculative transactions. Instead, Martinez and Torrellas propose a special component, the speculative synchronisation unit, which understands the logic behind the lock implementation and will perform operations such as wait for remote lock release, and delayed (non-spinning) lock acquisition. In their proposal, the speculative threads will execute the critical section without acquiring the lock, but can only commit once they themselves either managed to acquire the lock, or if the critical section completes, if they see the lock becoming free. During the waiting, the authors propose several novel ideas, such as continuing the execution past the actual critical section instead of aborting / stalling to wait for the lock to become free; also, they propose elision of thread synchronisation barriers, and producer / consumer constructions.

Rundberg and Stenstrom evaluate a similar approach, and recognise the issue of conflicts between transactions and how they can cause cascaded restarts and aborts [65]. They propose a mechanism for lazily ordering (and lazily versioning) transactions only once they have completed execution. By carefully tracking the overlaps and data (anti-)dependencies between transactions, the authors show that they can abort and re-execute a minimal set of transactions and order the remaining transactions logically without conflicts.

In a follow-on publication, Rajwar and Goodman also improve the progress of their lock elision solution [62]. They propose a time-stamp based priority mechanism (and perform eager conflict detection and resolution) that allows the oldest transactions to win conflicts and thus eventually succeed execution. They use the locally committed transaction count as logical clock and add a Lamport-like scheme for keeping these counts loosely synchronised. The net result is a system with strong progress guarantees; stronger than the original lock-based code.

2.2.3 Hardware Transactional Memory

The initial transactional memory proposals were hardware based; an unsurprising choice–the cache coherency mechanisms employed in all main multi-core / SMP systems provide most of the functionality to track conflicting concurrent memory accesses. Similar to STMs, the development of Hardware Transactional Memory (HTM) accelerated only during the 2000s, coinciding (again) with the limited progress of single-core performance. Herlihy and Moss, and Stone, et al, laid out the basics [34, 35].

Basic Differences Broadly speaking, HTM proposals differ on two levels: the *instruction-set architecture* (ISA) defines the set of application / programmer visible behaviours and mechanisms to perform transactional operations such as instructions, visibility of aborts, progress guarantees, etc. The *micro-architecture* (uarch) of the core / CPU / SoC / system describes how pipelines, caches, interconnects etc. interact to provide the ISA-level features. In many HTM publications, there is not a clean distinction between the two; instead, they will often be presented hand-in-hand. HTM proposals rarely exist in a vacuum (particularly true for commercial proposals / implementations), but rather will work with the existing substrate of both ISA and uarch.

2.2.4 Transactions as Standard Synchronisation Primitive

Instead of transactionally executing serial / parallel lock-based programs transactionally, Hammond, et al, propose Transactional Coherence and Consistency (TCC) – transactions as the main synchronisation and execution primitive, and as a replacement for complex coherency and memory consistency models in CMPs [74]. They collect the transactions' write sets and broadcast those at the end of the transactions. Reads use the caches for tracking the correctness of the read sets. Programmers start out with large transactions and subdivide them into smaller blocks, in order to reduce the cost of aborts and reexecution; there is no separate mechanism for mutual exclusion, instead, not breaking up a critical section into multiple transactions serves that purpose. In addition, transactions can be ordered to support TLS. In case of overflow of the transactional resources, the authors suggest an overflow buffer, and eventually advocate halting execution of the CPU and asking for global commit permissions (effectively serialising the entire system). One interesting extension that Hammond et al propose is the hardware automatically breaking up / merging transactions, while not splitting specially marked critical sections. In a second publication, the authors show the performance impact of different designs for TCC [84]. They find that double buffering only provides small benefits, mainly because the workloads are not limited in interconnect bandwidth. They also show that a small (8 - 32 entries) victim cache is effective in preventing cache overflows, and that the difference between invalidation and update-based designs is small across all workloads.

Njorge, et al synthesise a TCC-enabled system (called ATLAS) by adding new caches and interconnects in FPGAs to PowerPC hard-macro cores running at 100 Mhz [139], depicted in Figure 2.4. They show results for various L1 data cache sizes and observe that a few workloads scale, but their implementation does not scale well for a simple hash table micro-benchmark, due to lack of memory bandwidth and frequent misses. Assuming that TCC will only increase the required snoop bandwidth, this may be an indication of challenges to TCC's bulk coherency approach. The authors find that the transactional tracking complicates the cache design and adds 14% area overhead to the control logic and a 29% increase in memory structures needed on the FPGA (the design keeps one additional bit per 32 byte cache line). Unfortunately, the authors do not present any of the corner cases, state diagrams, etc., that they will undoubtedly have faced when implementing TCC in a real system. Finally, the system is not compatible with existing application code and does not support operating system level transactions.

2.2.5 Virtualising Transactional Memory

The argument for making transactional memory working despite capacity limitations stems from the observation that one cannot rely on the programmer to be able to cut down all transactions. Instead, the

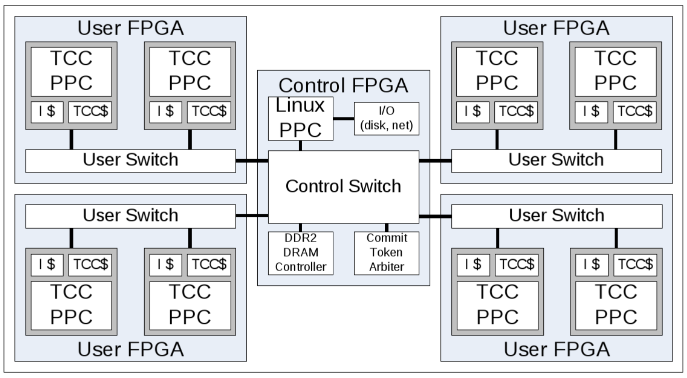


Figure 2.4: Block diagram of the ATLAS system that implements TCC. From [139].

system should support in some form transactions that exceed the limitations of simple HTM proposals, such as length (getting hit by scheduling quantum) and space (overflowing versioning and conflict detection structures). Similar to exceeding cache capacity, or size of memory, a fall-back mechanism is in place so that the application programmer does not have to work against hard limits, or needs to provide a fallback path; but enjoys good performance in the small common case. A significant body of academic work on HTM deals with the limited capacity of structures for tracking conflicts and buffering writes, and how transactions can survive events such as page faults.

Ananian, et al, propose unbounded TM (UTM) and a lightweight version (LTM) thereof [80]. Their UTM proposal is effectively a first-generation STM cast into silicon: visible readers, transaction state variables, and undo logging require complex changes to the entire CPU memory system; none of which are detailed in the paper. Together with physical addresses that remain stable across paging, the authors show a system with arbitrary capacity that can survive context switches, interrupts, etc. The more lightweight proposal (LTM) drops the requirement to survive interrupts and spills cache content into a hash table in memory that is consulted if a cache set indicates that it spilled. While closer to reality, the authors fail to explain the crucial aspect of this design, namely how the commit of a transaction will get the content of the (redo) hash table written into the proper memory locations without significant changes to the interconnect. Further, with only light evaluation (application traces and statistical properties), this approach remains a academic curiosity.

Similarly, Rajwar, et al, propose Virtualized TM (VTM) [86] and use the analogy and mechanism of virtual memory to present their design. To virtualise in case of context switches, they propose to dump all transactional content of the cache into virtual memory backed structures, and also use a single shared overflow table in virtual memory. They use a transaction status word that indicates the logical state of each transaction, and will have to walk the overflow structures in cases when transactions have committed. They suggest value-based validation for overflowed read-set entries, and perform filtering of accesses to the overflow table through bloom filters and remembering overflows per set. Opposed to UTM, VTM has more detail on how to commit entries from the overflow table into the real memory location: after having switched the logical state of the transaction, remote accesses to the hash table will stall the requester and allow write-back of all entries, after which the overflown entries can be reused and the transactions can be recycled. Unfortunately, the authors only describe a high-level system, without actually showing any interesting design detail, or present any evaluation.

Chuang, et al, take the concept of virtual memory even further and actually use the virtual memory mapping mechanism to provide both speculative and non-speculative versions of the data [109]. For that, they add another remapping structure that provides a physical address with a possible separate location for the old version of speculatively modified data (undo logging / eager versioning). Due to the page-level granularity, the simple mechanism has to copy over a full 4kB page from the old to the new physical location, which can cause slowdowns; in another option, the authors propose using a finer-grained remapping sub-table that can distinguish where new and old versions are stored. Overflowed conflict detection happens through per-cacheline TXR / TXW bits in the new remapping table, and a linked list structure of transactions accessing a specific page. The authors propose two caches that cache the required meta-data tables (one for the mapping between non-speculative and speculative data, and the other holding the transaction access logs). The mechanism is interesting as it splits the meta-data at page-granularity. However, the changes to the memory hierarchy and especially the timing of all required memory operations require complex interactions especially at commit and abort. The authors implement their scheme in Simics [61], and add large, fully associative caches to support it.

2.2.6 Improving Commit Path Performance - Eager Logging Unbounded TM

For safety reasons, many overflowing designs opt for a lazy versioning approach: speculatively written data will be hidden from other accesses, and only be made visible at commit. Depending on the visibility mechanism used, this may cause significant slow-down for the common case: a committing transaction. Moore, et al, propose the opposite in their LogTM proposal [102]: they propose a hardware-filled undo log in virtual memory, that is used to restore the pre-transaction copy *by software* in case of transaction abort. Adding entries to the log is easy: a simple pointer bump, and storing of address and data keeps a lot of locality and can often be hidden in the existing ILP of the CPU which does not saturate the write-bandwidth of the caches. Conflict detection is handled through per-cacheline TXR bits and their intersection with incoming snoop requests. In case of overflow, the authors suggest to either mark the entire set, or entire cache as TXR–thereby losing conflict detection fidelity. They do, however, assume presence of a directory that can potentially track more entries than the cache, and will filter unnecessary snoops that could cause false transaction aborts. If a conflict is detected, the authors propose a NACK message that will allow conflict resolution at the requester, which will usually wait and then retry. Unfortunately, the authors *fail to address important complexities* of their protocol: while they claim the design supports strong isolation, there is no detail given for the abort procedure that explains how the software handler can reinstate the pre-transaction value, if a transaction has to abort. Furthermore, the design does not investigate the important boundary condition of capacity limitations at the directory. In further publications, the authors extended LogTM to support closed / open nesting [118], and added signatures [134].

Signatures provide a compact, constant space, but lossy representation for sets of addresses. They use bloom-filters [3] (or similar structures) to hash addresses into patterns of set bits that are `OR`ed into the signature. Lookups check whether all bits of the queried address are set in the signature. In the same year as LogTM-SE, SigTM [126] advocated signatures; I will discuss that proposal further down.

Proper (as opposed to flat) nesting requires separate tracking of the working set of the nested child transactions, to allow independent abort and restart. Signatures allow a compact (but lossy) representation of the working set of transactions, and together with the undo log in virtual memory used by LogTM, enable the design to not require any modifications to the caches themselves. Due to their compact representation of the working set and support for set operations, signatures can easily be used for tracking the working sets of nested transactions.

2.2.7 Micro-Architecture of Transactional Memory Implementations

Memory System Details One of the obvious bottlenecks of TCC is the single commit lock which forces serialisation of commits. Chafi, et al, extend the single commit point by allowing commits to happen in parallel if they are to different addresses (clearly) and to different directories [138]. Overall, the protocol stays pretty true to TCC, by executing transactions to completion and collecting read and write sets, and sending them to the responsible directories for commit. These in turn use a two phase validation, then write-back approach. The authors ensure progress with a ticket-lock-like approach where waiters queue at the end of a logical queue [25]. The proposal is not fully parallel and decentralised, as parallelism is very much limited by the number of directories and the address distribution across them. The birthday “paradox” will quite likely restrict available parallelism. Finally, the authors have an unrealistic assumption that the directories can track the entirety of physical memory with tx-read and tx-write bits. The paper is notable for the level of protocol interleaving that the authors analysed; thanks to a fairly realistic implementation of the interconnect mesh (yet quite simplistic single-issue cores). Furthermore, the authors extract useful information about transaction sizes (instructions, number of loads / stores, etc. for various workloads).

Similar attention to detail is present in Tomic, et al, EazyHTM: they propose eager conflict detection through coherence messages, yet resolve conflicts only at the end of execution [196]. They also stress that serialised commit in hardware transactions is a big scalability issue and propose fully parallel commit by detecting (the absence of) conflicts in a distributed fashion eagerly. They employ two way tracking, i.e., incoming and outgoing conflicts for additional stability of the algorithm, and add typical tx-write bits per cache lines for reduced traffic. While commit is parallel between cores, each writing transaction needs to serially write-back its write set, which reduces local MLP and commit performance. On the core / ISA side, this proposal is quite straightforward with simple cores, full register snapshot; the attention of the proposal is in the memory system interactions.

Another detailed look at TM implementation aspects is Sanchez’, et al, detailed look at signature implementation options [143]. They show that instead of using a single k-valued Bloom filter, it almost as good to use k single-valued functions in a split filter; and much easier and smaller to implement. They also find that degenerate signatures can be especially limiting for larger systems, and note that other snoop filtering techniques (such as inclusive higher-level caches, snoop filters, or directories) can restore precision and thus reduce the impact of the Bloom filter.

Yen, et al, also investigate optimisations to Bloom filters for Transactional Memory [178]. They show that selecting the right subset of bits (lower level) for hashing can improve the Bloom filter degradation, and filtering thread-local accesses from the TM (and the filter) further leaves valuable room for those accesses that require synchronisation.

Bobba, et al, lift the concept of *token cache coherence* [71] to transactions and support arbitrary sized transactions by tracking tokens for data items in the entire memory hierarchy (using ECC for main memory) [157]. Readers need to acquire a single token, while writers need to acquire all tokens for a specific data item. On top of the full credit / token mechanism, the authors implement faster simplifications for small transactions that fit into the L1. Writes are buffered (again!) in a software-visible log. In their evaluation, the authors remove the logging problem by actually performing logging in software (with very little overhead claimed).

Blundell, et al, propose an interesting way to precisely capture a larger working set in their OneTM extension to LogTM [121]. On overflow, they convert a part of the L2 cache to cache only permissions, instead of also data for a specific address range. That effectively increases the coverage of a cache-line to $N * 8/2$ cache lines (with N being the cache line size in bytes). Similar techniques are being used in sub-sectored caches [36], and for AMD’s probe filter design that uses part of the LLC capacity to track only the coherency state of more lines [216]. For full capacity virtualisation, the authors propose a somewhat heavy scheme in the CPU that supports transactional meta-data bits in memory. In comparison to the heavy proposals in VTM [86], OneTM has a lighter approach, as only a single overflowed transaction can exists (concurrently with non-overflowed transactions) and so only a single copy of dedicated transactional meta-data bits is ever required. OneTM allocates these bits in physical memory and thus reduces the available memory space.

Blurring the Eager / Lazy Distinction Lupon, et al, revisit the logging TM systems and propose an eager / lazy hybrid, the fast-path of which turns out to look very similar to what I explore in this work (and most industry proposals) [205]. Their main contribution is improving the abort speed for logging TMs and showing that this indeed has a significant impact on overall application throughput. While they are trying to hang on to the eager / lazy terminology, it becomes apparent that in hardware TM implementations, these distinctions are not always useful / obvious. They do, however, identify the technique to eagerly acquire write permissions for transactionally written lines, and lazily hiding their values in the L1 data cache (plus also putting them in a log for virtualisation). In the fast path (no overflowing lines), their aborts can execute in hardware, rather than using a software abort handler to consult the undo log. They also explicitly mention that overflowed lines need to NACK a requesting sender and require SW involvement before the requester can actually read the value – a property which is often unacceptable in modern, complex, deadlock-free interconnects and coherence protocols. One interesting observation, however, of the authors is that transaction write sets very often fit into the L1 cache, and using signatures for tracing the read sets and keeping transactionally written data in the cache as long as possible often avoids the need for a logging slow path.

In a follow-on publication, Lupon, et al, further dissect the coexistence of eager / lazy transactions and switching between them with a predictor; they also seem to be the first to realise that with the right amount of eager / lazy mix, distributed commit seems possible [228]. They show how “eager” (really fully logging, decoupled from caches) transactions and “lazy” (using the cache hierarchy and coherence mechanisms) can coexist. Both modes are similar when the working set (the write set) fits into the L1 cache: conflicts are detected eagerly; in the “eager” mode, they abort transactions immediately, in the “lazy” mode, handling the conflict is postponed until transaction commit time. The authors also propose several extensions for the coherence protocol, enabling coexistence of eager / lazy transactions, but also significantly making the design more complex.

Improving Progress One important aspect of flexible eager / lazy switching are the different progress characteristics. Similar to work in software TM, improving progress of hardware TM implementations has been looked at in the literature. The challenge is to balance the complexity (resources, changes to coherence protocol and structures) of the scheme employed and the provided benefits.

Bobba, et al, investigate performance pathologies in HTM, and they find that under low contention levels, most systems perform similarly [142]. Under high contention, however, they identify several patterns depending on the different conflict detection / conflict resolution / versioning design points. In particular, they find that no single outperforms the others for all workload cases. Simple additional policies, such as exponential back-off, selective early write acquisition, and adding timestamps to memory requests to abort younger transactions provide significant performance improvement in the pathological cases.

Ramadan, et al, go further than Bobba, et al: they maximise transactional throughput under contention by *tracking dependencies* between communicating in-flight transactions, rather than aborting conservatively on any communication [167]. For that, they order commits (and subsequent writes) of communicating transactions, and only abort in cyclical dependency cases. The authors use a distributed ordering vector that is read and updated frequently, and extend the cache coherence protocol to eleven stable states to track all occurring forwarding conditions (typical protocols such as MOESI have five stable states!), they also show that their mechanism requires sub-cacheline coherence tracking to unlock the full performance potential. Extending a MetaTM implementation, they obtain 30% speedups in STAMP.

Similarly, Titos, et al, also track dependencies and use a hybrid eager / lazy mechanism to get transactions committed in conflict cases [203]. Instead of aborting writers in RAW conflict cases, the readers read the pre-transactional value from where it is stored. In turn, the reader then has to commit before the writer. Similarly, in WAR cases, aborts are not necessary, if the writer commits after the readers. Experiments on STAMP in a tiled SMP, extending a LogTM-SE protocol in GEMS, the authors show that their system adapts to the better eager / lazy policy and thus reduces memory traffic and improves throughput.

Typically, it is either caches storing transactional data, or the data is appended to a write log in most TMs. Yan, et al, identified the required data movement in case of aborts / commits as a major source of bottleneck [302]. The authors instead propose to store both transactionally produced and pre-transactional versions of data in the same *pseudo-associative* cache [32], and flip the N-th bit of the address to denote the N-th version of the data. Together with tracking the partial order of concurrent transactions, they can enforce that there are no cycles, by having transactions read the right version instead of creating a dependency in the wrong direction. In addition to a changed coherence protocol, the authors use *counting* bloom filters [45] to track overflowing locations.

Negi, et al, propose a similar design, but keep all the required additional information in a separate cache array [293]. Their separate cache tracks the transactional dirty copy, the according pre-transactional clean copy of the data in the same cache, and also supports clearing of the transactional dirty bits per way. When the main cache observes multiple tag hits, the separate cache distinguishes which version of the data to access. Furthermore, the by-way management can allow simple partitioning of the transactional resources for and simplify conflict detection between multiple threads on a single core. Similar to Yan, et al, the authors here also find that storing both copies in the same cache can have a significant benefit for high-contention scenarios, and observe speedups of 30% in STAMP intruder.

2.2.8 Instruction-Set Design for Hardware Transactional Memory

Most previous papers focussed on the microarchitectural design of HTM. Several papers put an emphasis on the instruction set architecture (ISA) for HTM. McDonald, et al, propose an extensive set of instructions in [113]. They expose features such as "two phase commit"[1], handlers for all transactional state changes, and support for open and closed nesting. In addition, they argue for allowing in-flight transactions to synchronise through memory without causing aborts, and also propose an escaping (suspend / resume) mechanism that allows pausing of transactions. In later chapters of this thesis, I will show similar extensions, but implemented much more elegantly, without the need for a huge number of new instructions, memory locations, and other state registers.

With MetaTM, Hofman, et al, introduce various interesting extensions to the vanilla transaction begin / end instructions [124]. They argue for simple, yet interesting semantics, and remove complex concepts such as open nesting in favour for simple practical solutions, such as flat nesting. In addition, they add support for *transactional suspension* through `xpush / xpop` instructions allowing a transaction to perform non-transactional execution for a while. This can be used to suspend transactions during system calls, handling page faults, and for communication. The IBM Power HTM ISA extension features a very similar primitive [353]. Furthermore, MetaTM provides primitives that pause the transaction until a specific value is seen in a memory location, and optionally can then also CAS a new value into them. That way, MetaTM can wait (instead of aborting, spinning, and then restarting) for a spin-lock to become free. The paper shows a lot of detail for ISA design and the authors show how HTM can be extended to make it useful in a realistic software context without overly complicating the hardware. The ISA extensions in this thesis follow a very similar trend.

[1]Their 2PC provides merely validation, without guaranteeing that a transaction cannot be aborted after passing the first phase.

Zilles, et al, work in the opposite direction: they extend a VTM and synthesise more high-level primitives on the already strong HTM implementation [120]. Similarly to MetaTM, the authors propose pausing / unpausing a transaction, and rely on the virtualisation mechanisms of the underlying strong HTM. They, too, face challenges with clear semantics around overlapping transactional and non-transactional accesses and data flow between the transaction and the code running while the transaction is paused. In addition, Zilles, et al, propose blocking the victim of a transactional conflict until the winner has completed (similar to enqueueing on a taken lock, rather than spinning on it [191]). Blocking happens through an exception being raised at the victim which marks the thread as blocked. Wakeup is more complex and requires an elaborate list of waiters in the winners transactional data structures. That mechanism is also used to synthesise a retry primitive for HTM.

Swift, et al, show that “unbounded” HTM systems using logs can be problematic in virtualised use cases, as operating systems do not necessarily know the right physical addresses of objects [173]. They extend LogTM-SE’s log with tracking of the working set on *virtual addresses* and therefore can elegantly allow page remapping, and virtualisation. Context switching the signatures (for conflict detection), they also provide suspend / resume similar to MetaTM, but instead of only allowing brief interruptions, transactions can survive long context switches.

FlexTM exposes the separate concerns of HTM as separate instructions to an ISA: signatures for tracking conflicts, a notification mechanism (alert-on-update) to push / get notifications when the transaction status changes, memory mapped data structures detailing conflicts (between any two cores), and software contention management [171]. On top of that, Shriraman, et al, synthesise TM in both eager and lazy versions. Eager transactions immediately call the conflict handler on the requester when hitting transactional remote data. The handler can then either abort the local transaction, or abort the remote transaction through updating the remote TX status flag and the alert-on-update mechanism. Lazy conflict detection uses the conflict summary tables to mark conflicts as they happen and are evaluated at the end of the transaction. Furthermore, the authors describe complex interactions such as paging, page move, and overflow. Their evaluation finds that exposing the mechanisms independently does not cause significant performance overheads – the only significant overheads come from checkpointing the register file manually. While exposing the sub-mechanisms independently clearly seems feasible, the FlexTM proposal contains a few critical interactions: several memory-mapped data structures need to be updated at the same time under contention; the commit procedure is effectively a small monolithic mini-transaction and coordination between committing transactions needs to be carefully coordinated. The overflow mechanisms suffer from the typical challenges such as careful memory access coordination, and potential protocol deadlocks. Overall, however, the proposal appears complex but well thought through.

2.2.9 Other Concerns: Energy and GPUs

Transactional memory can affect the power consumption and energy efficiency of a system in both positive and negative ways: if conflicts are rare, parallel transactions reduce the application runtime, and thus energy consumed due to leakage. If the application would have spun on a lock otherwise, transactions will also reduce dynamic energy consumption. If, however, transactions abort frequently, and have to roll back, they execute more instructions, cause more memory traffic etc. than if execution serialised – overall causing an increase in dynamic energy consumption. Furthermore, in such cases, transactions can actually increase the application runtime and thus increase static energy consumption. The transactional mechanisms (conflict detection, register checkpointing, keeping a copy of the data for versioning) make CPU design more complex and require additional logic consuming both dynamic and static energy.

Ferri, et al, investigate the effects of TM on energy and performance for an embedded Arm SoC. They design the HTM with energy efficiency in mind and show that they can *reduce* EDP by a factor of four [223]. Their design has several unique points, such as a small victim buffer for transactionally written data that helps avoid the pathological cases of transactions exhausting the L1 cache buffering capacity.

In a follow-on paper, Ferri, et al, further extend their implementation and push it entirely into the cache (similar to IBM's BlueGene/Q design [250]) and extend it with mechanisms to avoid performance (and energy) pathologies of TM: they add ordered commits (guaranteeing progress for at least one thread), and share expensive logic (bloom filters for conflict detection) between multiple cores [247]. Together with per-core scratchpads and a PGAS-style (for example [133]) programming model, they show that TM can be very beneficial for application performance and energy-efficiency.

Transactional Memory for GPUs Energy efficiency is also a main concern of another, highly threaded domain: graphics processing units (GPUs). The original problem of graphics exhibits a lot of data-level parallelism and therefore GPUs are keeping thousands of threads in flight and often offer vector units for computations. Recently, GPUs have also been exploited for general purpose computations (GPGPUs); there and in several graphics algorithms, synchronisation has become an issue. Fung, et al, propose HTM for GPUs – in a very different base line to what can be found in SMPs / CMPs: 1000s of threads, no or rudimentary cache coherence, tight execution coupling between threads of a single warp. The resulting design is thus significantly different from the ones explored for CPUs: their initial design performs value-based validation at the word level at commit time and performs logging. They perform parallel commit, and detect hazards with currently committing transactions [248, 265]. At first sight, value-based validation is prone to the ABA issue [41], but Fung et al show that this is not an issue in HTM [266]: the key insight is that the ABA problem occurs because only one memory location is used for coordination; missing an update on that location masks a missed update in another memory location. In VBV, however, *all* memory locations are validated; there cannot be a silent change of a memory location that the transaction may have relied upon.

Fung, et al, also show the benefits of transactional memory on *programmer productivity*: in Figure 2.5 they show that both single-threaded and transactional code reach correctness / feature completeness quickly, and both require tuning for highest performance [265]. The TM version of the code, however, provides for a significant higher performance once fully optimised. The locking version of the code will eventually be faster, but significant effort is spent by the programmer to ensure correctness and functional completeness.

Finally, Fung, et al, adapt well-known techniques from other TM implementations in their GPU HTM [291]. They enable fast read-only transactions through time-based reasoning, hierarchical conflict detection that detects conflicts inside a warp first, and prioritises lower lane numbers for conflicts to ensure progress.

Villegas, et al, in contrast focus on implementing transactional memory in the *local* memory of GPUs. That memory is used for synchronisation between threads (work items) of a single work group and does provide stronger coherence benefits. Consequently, they can perform local, eager conflict detection (per memory bank with bloom filters) rather than commit-time centralised conflict detection [377].

2.2.10 Enhancing Transactions

Extended Transaction Models and Concepts Conceptually, transactional memory provides a simple programming model which can improve application performance and simplify programmer reasoning. It does, however, suffer from several performance pathologies and implementation complexities. Therefore, several proposals tackle the structure of transactions directly to improve throughput or simplify implementations.

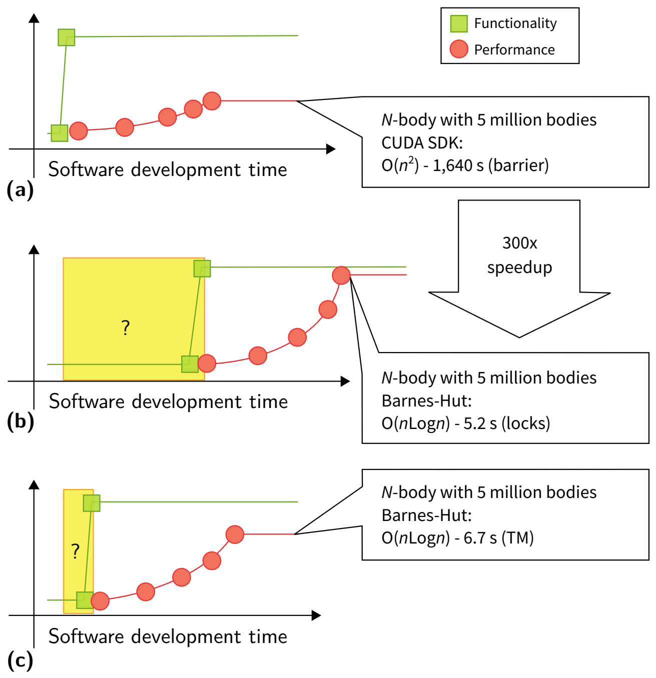

Functionality
Performance
Software development time
(a)
N-body with 5 million bodies
CUDA SDK:
O(n2) - 1,640 s (barrier)
300x speedup
?
Software development time
(b)
N-body with 5 million bodies
Barnes-Hut:
O(nLogn) - 5.2 s (locks)
?
Software development time
(c)
N-body with 5 million bodies
Barnes-Hut:
O(nLogn) - 6.7 s (TM)

Figure 2.5: Comparing development time for functionality and performance for HTM (on CUDA with KiloTM on a GPU), (a) single-threaded, (b) high performance with fine-grained locks, and (c) transactional memory. From [265].

Felber, et al, propose *elastic* transactions; a weaker form of transactions that captures the intuition that in many linked data structures, the consistency of the prefix of a traversal can be modified as long as an environment of a specific size around the current read location / modification stays intact [207, 380]. For that, Felber, et al, define the concept of a sized cut which corresponds to this notion of "transactional bubble" around the current read / write location. Allowing elements to effectively drop from the working set permits higher concurrency, yet is a cleaner model than direct *early release* of memory locations [68, 106]. Still, defining the right size of the cut depends on the data structure implementation, and correctness and composition become non-trivial to prove. With an STM implementation, the authors get speedups sometimes similar to the fine-grained locking versions of the data structures.

Other authors explore the idea of splitting transactions for improved performance. Such concepts are: *consistency-oblivious programming* [240, 307, 308], *partitioned transactions* [346], and transactions with isolation and cooperation [152]. These works make the programmer manually define the read-only (or read-mostly) prefix of a data structure operation, and then perform an update in a second part. Often, the prefix is further split into a non-transactional *traversal* phase with subsequent *validation* to ensure that the found location is still correct. For correctness, the validation is executed in the same transaction as the actual data structure modification.

Code modification and correctness verification for these techniques remains a challenge, as is composition. Conceptually, it should be possible to execute the multiple prefixes of the sub-operations, first, and then have a single transaction that contains the different validation and modification phases. However, the required code motion is not trivial and compiler support does not exist. In other cases, the traversal phases are not independent so they cannot be "pulled out" from the composed transaction.

In the cases where composition is not required, splitting transactions into phases can unlock significant performance gains, authors report gains up to 60% in STAMP applications (and significantly higher for micro-benchmarks) due to the reduced number of conflicts [346]; often restoring scalability to workloads levelling off.

Smaragdakis, et al, propose a way to split the transaction[2], while maintaining semantic variants across the split [152]. Through the type system, they enforce that every caller of a transaction provides a set of semantic invariants that get checked when the split transaction continues. Composition and nesting are supported through that, because these invariants are collected from the outermost caller and they are all run on continuation. Effectively, the authors propose a suspend / resume mechanism with semantic (rather than by value) revalidation. During the break, the transactions can perform system calls, I/O, and can also communicate or synchronise with other concurrent threads and transactions.

Lev and Maessen show how to use a typical best-effort HTM to implement any of the described concepts (and others, such as true closed nesting) [163]. They effectively use the hardware transactions to ensure the validity of the read set for every sub-transaction (while writes are buffered in a software log) and as the commit stage of the final transactional part [163]. Similarly to Smaragdakis, et al, these split hardware transactions need to revalidate the invariants at the beginning of a split. Instead of semantic handlers, Lev and Maessen use direct validation of the values in the read set which they keep in a separate software log. Suspend / resume effectively work by committing the current hardware transaction split (with the only modifications being the changes to the local SW read / write set structures), and resuming starts a new hardware transaction, and reads again the read set and ensures no values have changed.

If the code that runs during the break checks the SW write set, communication between the phase and the suspend code is possible. While possible through compiler instrumentation, this may require extra work for calling unmodified code, such as with libraries or system calls. An instrumented shim layer could be used that pulls the right values out of the write set, and puts them in a temporary object for consumption by the binary code.

Finally, Saad, et al, observe that many conflicts at the data value level do not change application semantics; they propose a higher-abstraction interface to TM that encodes application knowledge and can reduce aborts [362]. Two key classes of operations are easily offloaded to the TM system: predicates / comparisons and associative and commutative operations. The former can stay true / false, despite the underlying values changing, and the latter can be efficiently grouped and performed concurrently.

Through an extended API, applications can tell the TM system about these operations, and it will use the higher-level predicates for re-validation (instead of comparing value equality) and keep shadow copies and reduce the commutative operations in case of overlap. Of course, when an application actually performs plain reads, full value equality has to be ensured. The authors report significant (4x) speed-ups, largely due to significantly reduced conflict / abort rates while still providing a composable and abstract programming interface.

Transactional Memory and Non-volatile Memory Transactional Memory derives its concepts largely from database transactions and adapts these from a block-based storage medium to byte-accessible main memory (implemented through caches and DRAM). Recently, non-volatile memory has appeared as a feasible technology that combines DRAM-like access granularities and performance characteristics, and furthermore retains state under power loss [169, 192, 195, 198].

Due to the performance and access characteristics, it makes sense to address that memory using loads and stores. There is, however, one important complication: the CPU registers and caches will lose their state when power is lost, and usually only the NVM will retain its state. Writes to that memory may be arbitrarily ordered by the memory system (as they should be cached and only written to the memory on cache eviction), and power may be lost at any point in time. That latter property can be modelled as having a concurrent observer (namely the recovery code that inspects the memory state after a power loss event) that can concurrently observe memory state – turning even single-threaded applications into concurrent ones. A further complication is that the coordination with that concurrent observer can only

[2]and eat it, too!

be performed through the NVM. There is no additional side-channel through the coherence protocol, or grabbing a lock to protect against power loss.

Transactions are a useful construct to structure applications so that power loss only rolls state back to a previous completed transaction state, effectively providing atomicity and durability. On top, isolation properties need to be incorporated for when actual multi-threaded applications need to synchronise. Several papers propose implementations for transactions (both HTM, and hybrid transactional memory) in NVM, or use critical sections as a mechanism to provide failure atomicity [309, 334, 354, 372].

Failure atomicity for persistent memories is achieved through logging values to the NVM either through a redo-log in NVM, and not modifying the values in NVM before the transaction commit, or though undo-logging, where each in-place modification can be performed eagerly, but the corresponding undo-log entry needs to be written before. Atomicity is then achieved by either switching the redo-log to "live", or "disarming" the undo-log; both can be achieved through a single write which will be the persistent linearisation point.

2.2.11 Hybrid Transactional Memory

A fairly obvious idea is to combine Hardware and Software Transactional Memory in order to get the benefits of both: HTM provides a fast common case, while STM handles the complicated cases that for example overflow the HTM's limited capacity or run for very long. While straight-forward, the challenges are in the details: making sure that the HTM and the STM can coordinate among one another without adding too much meta-data burden to the HTM, e.g., per-access instrumentation also in the HTM path. Compared to unbounded HTM, HyTMs are simpler in their hardware requirements, because they offload the hard cases to software with a performance penalty, yet still offering composability and scalability as opposed to falling back to a single global lock.

Beginnings Moir laid out the general idea in his 2005 technical report [94], and suggests a simplistic STM design using orecs, word-granularity, and blocking write-backs; similar to the combination of the proposals of Harris & Fraser, and Ennals [66, 117]. He argues that such a simple STM could easily interface with an HTM by ensuring that for every access the HTM checks and adds the relevant orec entries to the HTM read set. One caveat is that reads must be tracked semi-transparently: a reader count is kept per orec; with invisible readers, the HTM might commit writes without the STM noticing. Moir also notes that with a simple STM transaction counter, the HTM can also execute without any overhead. Damron, et al, extend the initial idea in their 2006 paper and present a full HyTM stack, with compelling initial results for also just the STM part [111].

In the same year, Kumar, et al, approach the problem from an entirely different angle [98]. They extend the lock-free STM from Herlihy, et al [68] with an HTM. In doing so, they leave in place the indirections and the object-level granularity of the DSTM design; forcing HTM transactions to walk the indirections. They provide an interesting ISA extension that allows non-transactional accesses in hardware transactions, both by default and with special markup. Furthermore, they rely on the HTM as a single location notification mechanism for *all* software transactions, which is problematic: timer interrupts and cache displacements due to non-transactional accesses will be likely causes of aborts for those hardware transactions. Together with the excessive usage of indirections, the authors fail to show compelling performance data.

An interesting solution that side-steps the issue of coordinating between STM and HTM is Phased TM [151]. There, only one *scenario* of transaction can be executing at a time, for example, only (full) hardware transactions in one phase (`HARDWARE` mode), only software transactions in the next (`SOFTWARE` mode), a mix of both in a third (`HYBRID` mode). In addition, two modes operate in a single-threaded way by using a global lock, and, if possible, also do not perform logging. Careful switching of the modes is necessary; hardware transactions can immediately be aborted, while software transactions are being waited for to complete / abort (quiescence). The paper does not give definite answers for when to switch phases. Initial results show, however, significant gains from having an HTM-only phase with reduced overheads in the hardware transactions. They also show the overheads of the semi-visible tracking of readers in the software part of the HyTM to be significant in *STM-only* cases.

Coordination lower bounds Generally, in HyTMs the interaction between fast hardware and slow software path depends also heavily on the type of STM that is being used. In the simplest case, the fall-back path simply grabs a global lock and executes the transaction (the hardware path directly in most cases) without instrumentation–analogously to (reverse[3]) lock elision. In these cases, however, the transaction cannot use aborts, for example for failure atomicity, or thread-level speculation; furthermore, the progress guarantees of the HyTM will be that of the global lock. Somewhat intuitively, there is a trade off between the intricacy of the instrumentation (and the resulting loss in performance) and the progress guarantees that can be made (similar to plain STM design trade-offs); HyTM could, in theory help, by running transactions in a lower progress class with higher performance, and only when progress stalls, switch to a more complex regime. Interestingly, there is proof by Alistarh, et al, that there is a limit to this: every strictly serialisable HyTM needs instrumented accesses in the fast path even for weak progress guarantees. Further, the instrumentation cost will be linear in the working set size [333]. This suggests that a minimal phased approach might work best: full HTM speed, with maybe limited STM support, phasing to instrumented HTM with better-progress STM, if necessary.

Privatisation safety In 2008, Vallejo, et al, continue accelerating indirection-based, object granularity TM [185] using a queued reader-writer-lock mechanism, similar to the semi-visible readers outlined earlier. They note, however, that the visibility of the readers allows their design to be privatisation safe: they can delay a committing writer until all readers have completed. The authors also suggest optimisations for the HTM path, in line with earlier work cited here, and claim that the HTM path can offset the higher cost induced by the heavier locking in the STM. Somewhat surprisingly, the authors perform fine-grained checks per read / write barrier whether they are in a hardware or software transaction, rather than using two code paths that perform this check once at transaction start and subsequently diverge. Consequently, their HyTM has considerable overheads, still, and they do not show a performance comparison to a plain HTM, sequential version, or a single global lock.

In the same year, Baugh, et al, also present a privatisation-safe (even strongly isolating) hybrid TM system without overheads for the HTM [183]. For that, they use fine-grained memory protection hardware that protects the STM's working set from concurrent read / write accesses caused by either HTM transactions or non-transactional accesses. While the authors argue strongly that hardware should stay as simple as possible, no such fine-grained memory protection exists up until today. Their proposal shares some similarities with the Alert-on-Update proposal from [153] and SigTM [126], which I will discuss further down.

Using non-transactional accesses in hardware transactions The NOrec algorithm introduced by Dalessandro, et al, in 2010 [217] contains a sketch for an efficient HyTM. As there are no orecs in the algorithm, the HTM path does not have to perform per-location checks, but only needs to (1) serialise with the HTM commit phase, and (2) announce its updates to the software transactions and force revalidations. Both Riegel, et al, and Dalessandro, et al, extend this basic idea [244, 255] in 2011. Earlier,

[3]In lock elision, an existing lock in the application is turned into a transaction. This mechanism turns a programmer transaction into a critical section protected with a lock.

Felber, et al, showed new techniques for performing the required coordination between the STM and HTM code path with novel non-transactional idioms [238], both for NOrec and orec-based HyTMs.

In summary, the following techniques are proposed: (1) monitor only meta-data (orecs) and use non-speculative accesses for the actual data (reduce HTM capacity requirements), (2) use non-transactional reads of HTM to let an ongoing STM transaction commit, while still receiving all actual data conflicts, (3) use non-transactional, atomic read-modify-write operations inside the transaction to signal re-validation to the STM without causing aborts between multiple HTM transactions, (4) using multiple (up to per-core) HTM-to-STM signalling paths to reduce cache misses and HTM-HTM interference, (5) lazy subscription to STM commit messages, with non-transactional per-access checks in the HTM.

Riegel, et al, also extend their time-based STM to work with HTM, using their proposed optimisation of monitoring only the orec meta-data, and updating the global time-stamp with an atomic RMW instruction [255]. Both proposals use ASF (the HTM introduced in this dissertation) and make clever use of the allowed non-transactional accesses inside transactions. Dalessandro, et al, in addition evaluate their HyTM also on Rock, but due to the limited hardware implementation (aborts at function calls, aborts at branch misspeculation), they cannot show compelling results [244].

Matveev and Shavit propose a different type of HyTM and also a way to emulate transactional performance on real silicon without simulation [305]. Similar to previous authors, they identify that the loading of STM meta-data in the HTM part introduces additional memory traffic and conditional branches per access. They produce *reduced hardware transactions* that sit between full STM and HTM transactions. The first level of transaction are full hardware transactions, with additional bumping of the Orec of writes, and subscription to the global version clock. In the second level, transactions start in software and perform lazy writes in software, but the commit path uses a single hardware transaction to (1) validate the read set and (2) perform the write-back operation. By that, the slow path will still have capacity limitations (in particular since it also needs to read the read / write log in the hardware transaction), but mechanisms can allow long-running transactions, and system calls inside transactions. Finally a full STM will then handle those transactions that cannot be handled in the first fallback path. In another option, the first slow path does not perform the read set validation in the hardware commit transaction, but then needs to put additional work to the fast path. Non-transactional accesses would simplify a few aspects here by allowing to reduce the HTM capacity requirements in the commit HTM path. Finally, the authors evaluate their system without HTM silicon or a simulator, but instead use *dummy accesses* and abort probabilities for timing analysis on existing CPUs.

Using commercial HTMs With the broad availability of actual HTM implementations starting in 2013 (desktop) / 2014 (server), several hybrid TM implementations use the available HTM mechanisms. Similar to Matveev and Shavit, Calciu, et al, also propose multiple levels of transactions in their Invyswell proposal [332]. Their HTM is the actual Intel Haswell RTM [303, 367], and the authors lament the lack of escape actions / non-transactional accesses. The proposed TM system is designed around visible readers through per-transaction software signatures. That way, transactions do not need to self-validate. Instead, all types of transactions can actually perform the write-back in the commit phase and *afterwards* check for conflicts. In their hierarchy, the fastest HTM path uses solely uninstrumented hardware transactional accesses, which can only commit if there is no concurrent software transaction. The signature design allows for a slower hybrid mode that runs transactions in hardware, but also records signatures in software; allowing full concurrency with software transactions. At commit time, the HTM makes all these modifications visible (both data and signatures) and then intersects the working set with other software transactions (the hardware transactions will abort through conflicts on the actual data).

Calciu, et al, also investigate lazy subscription for the global lock in the simplistic single-global-lock fallback path [331], and show small performance gains in STAMP.

More recently, Matveev and Shavit revisited their reduced hardware transaction approach and apply it to the NOrec algorithm [305]. In their design, transactions will perform their read prefix part in hardware, then read the global clock and commit this read-only hardware transaction whenever they encounter a write. Writes are then buffered in software, and committed through a write-mostly hardware transaction at the end of the logical transaction. With this construct, full fast-path hardware transactions can safely perform lazy subscriptions, without any lack of opacity or privatisation safety. Results show that these designs can be faster on the limited Intel RTM (no non-transactional accesses) than previous hybrid NOrec algorithms, especially when hardware capacity is constrained by hardware multi-threading.

Optimising performance and removing lazy subscription As outlined earlier, all the hybrid TMs shown so far must trade off the instrumentation cost in the fast path, the available concurrency between fast and slow path, and provide that with a specific progress guarantee. In [333], Alistarh, et al, show that one cannot build a hybrid TM that is strictly serialisable without any read / write instrumentation in the fast path; even for very weak progress requirements. Furthermore, the authors show that linear-sized meta data is required in a HyTM, if progressiveness is required. In summary, that means that one cannot get concurrent hardware / software transactions with constant (not per-location / no orecs) meta data. The authors prove their theorem, and also show several new algorithms that are feature-pareto-optimal.

Afek, et al, propose a quite different HyTM design for performing lock elision [349]; in their approach, the non-concurrent software fallback path uses visible reads, which the hardware fast path will check; and lazy versioning for writes. These writes will be written back with a hardware transaction at the end of the slow path ensuring that no fast path transaction can see a partially committed state. As a net effect, the system permits free overlap of fast / slow transactions, while limiting the number of concurrent slow path transactions to one. One interesting additional usage of HTM in the slow path is the merging of multiple metadata updates for the reads. The authors note that the memory fence required between the lock acquisition and the subsequent data read (even on TSO / x86 memory models) can significantly reduce performance. Merging multiple instrumented reads allows to remove the intermediate memory fences. To that effect, the authors propose an adaptive mechanism to set the number of merged read allocations. The resulting HyTM mechanism is opaque, but performance is close to lazy subscription, which is not safe.

Similarly, Ruan and Spear show that a HyTM can have performance similar to lazy subscription, but without the associated safety issues [350]. Their design also uses hardware transactions to perform parts of the software work. The key insight of this design is the concept of *cohorting*; a micro-phased approach that synchronises transactions so that all running slow-path transactions wait to reach the commit phase at the same time (and no new transactions may start once they do). They then use a hardware transaction to perform validation and write-back; if that fails, they will try again with a sequential commit. Hardware transactions can execute during most of the lifetime of the software transactions, but they cannot commit while software transactions are live; effectively allowing hardware transactions to live longer than software transactions and then commit. Ruan and Spear show how to decompose the state (phase selection) variable such that different transactions do not experience false conflicts when updating / checking the state. They too argue that with non-transactional operations, the state management could be simpler and in addition they could implement waiting in the hardware transactions instead of aborting when they try to commit when there are still software transactions in-flight before their commit phase. Similar to Afek, et al, the results obtained are close to the lazy subscription performance without the associated safety issues.

2.2.12 Composite Transactional Memory

Another form of transactional memory that is hybrid in nature is that of composite / symbiotic transactional memory. Instead of merging a fully functional (extended) HTM, and a fully functional (adapted) STM, the composite / symbiotic TM systems *rely* on both software and hardware mechanisms. As reads usually outnumber writes, the functionality is usually split such that software will perform data versioning for transactional stores, and the hardware provides conflict detection. In some cases, these mechanisms are established as “STM accelerators”, in particular, if the HW mechanisms indeed are optional.

SigTM [126] employs bloom-filter [3] based signatures for both read and write sets, while the alert-on-update (AOU) family of proposals uses the caches similar to HTM [105, 153, 154]. Hardware-assisted STM (HASTM) [119] provides a similar type of marking memory locations; their proposal marks on sub-cache-line granularity.

The implementations differ mainly along three dimensions: (1) immediacy of aborts, (2) capacity overflow behaviour, (3) tracking precision. In total, these three determine the amount of work the STM still has to perform. The proposals are all similar in that they focus on reducing overheads for the read path and rely on software to handle transactional stores. With read-heavy transactions (more reads per transaction than writes) and workloads (more reading transactions than writing ones), results are usually not much worse than HTM; with additional benefits of simpler hardware and (in the case of lossy / compressing tracking) larger capacity.

HASTM has asynchronous aborts (applications need to query if there was a conflict), increments a counter on overflow, and tracks on the granularity of 16 byte. The authors use the mechanism strictly as an accelerator; the STM still handles orecs, read logs, and performs versioning. The acceleration results from the fact that the hardware will detect conflicts cheaper avoiding costly validations and checks for it, and also avoids redundant checking of the same orec. The benefit of having the full STM is that the marking mechanism is strictly a performance optimisation. On context switches, the marks may be cleaned, and the transactions continue fully in software. In a more aggressive mode, the STM can stop read logging, at the cost of having to abort at such events.

AOU had been concurrently proposed (originally as part of a much more complex HTM system [105]), and similarly allows marking of cache lines. In contrast to HASTM, however, it is used in a synchronous fashion as alerts will be forwarded to the application immediately. In [153, 154], the authors use the mechanism to accelerate a *non-blocking* STM by removing indirections. They use a *single* entry AOU mechanism to implement a revocable lock that makes the write-back code path non-blocking. They also propose a cache-based multi-location AOU system and similarly use it to remove the validation overhead. Their basic implementation is very similar to the aggressive HASTM mode; performing only checks and marking for reads, and relying on locking orecs for writes. They suggest to anticipate the HW capacity limitations and track the remaining locations through SW validation.

Finally, SigTM [126] occupies another point in the design space: signatures do not have capacity overflow conditions, but instead become more and more imprecise in tracking. In this publication, the authors assume immediate / synchronous notifications if specific classes of remote snoops intersect with the local read / write set signatures. The TM does not require any form of orec, but instead only has a local write set with redo log in software. Similar to HASTM, read barriers intersect with the write signatures to reduce lookups into the local write log. Commits are heavily supported by hardware by acquiring exclusive permissions for all entries of the write set (lazy conflict detection), and then switching the signatures into a “NACKing” mode which prevents progress of any other access to locations in the signature from another core.

Casper, et all, prototype a full Transactional Memory accelerator in an FPGA without any changes to the cores, caches, or coherence protocol [243]. Due to their FPGA implementation, they solve many corner cases that were not mentioned or handled by the previous simulator proposals. The most important observation is that a TM accelerator will have higher latency if it is not tightly integrated into the system. The authors solve that by an *epoch* mechanism that allows them to remove round-trips and synchronous communication with the accelerator. A similar design can be found in IBM's BlueGene/Q design ([250], and Section 2.3.3), and I will discuss challenges with such designs more in Chapter 4. The implementation is a hardware Bloom filter for conflict detection; versioning, register checkpointing, and abort handling (though polling a mailbox) happen in software. Communication with the accelerator happens through uncached writes. The resulting mechanism is (again, similar to BG/Q) more useful for larger transaction where the communication latencies per transaction start / commit matter less.

2.2.13 Concepts and Theory of TM

Various authors have investigated the conceptual interface and semantics of TM; in addition, many others have worked on proofs and abstract conceptual notions for progress and transaction safety. This thesis focusses on the former for brevity.

Harris introduced several concepts that go beyond the simple transactional properties of atomicity and isolation. He explored the interplay of (language) exceptions and side-effects with transactions, and proposes a type solution to the question of whether exceptions should abort, or commit a transaction when they leave its scope [91]. He also proposes handlers that get called for transaction abort and commit to deal with I/O in the transaction: output is buffered until commit, while consumed input in the case of abort is reconstituted.

Subsequently, Harris implements transactions in Haskell, and extends them with `retry` and `orElse` primitives [90]. These two primitives with transactional memory create a solution that is composable for both isolation (atomicity) and *blocking*. The `retry` is similar to a condition variable, in that it releases the synchronisation mechanism (abort transaction vs free lock) and then waits until a condition changes (monitor members of the read set changing vs declared variable changes its value and a signal is received).

Blundell, et al, introduce the concepts of strong and weak isolation (then dubbed “atomicity”), and show that transactional lock elision and strong isolation can cause deadlocks in specific corner cases of properly synchronised applications [82]. The cases they propose rely on either code under different locks executing concurrently and not causing conflicts, for example by adding a synchronisation barrier between two critical sections protected with different locks; or on a similar concept of a synchronisation barrier between unprotected code and code inside a critical section that is executed as a strongly isolating transaction.

Grossman, et al, investigate the interaction of memory consistency models and transactional isolation semantics [108]. They formalise the notions of isolation and ordering, and find that strong isolation allows local reasoning per code block, whereas weak isolation can have many surprising interactions. They also investigate ordering of transactions and look at typical properties such as transitivity, and *coherence* ordering of write-conflicting transactions. For these examples, the authors provide litmus tests that show specific desirable / undesirable interactions.

Contrary to that work, Menon, et al, attempt to formalise weak isolation models, as they can be costly to implement in STMs [165]. They propose several weaker forms than Single Global Lock Atomicity (SLA): disjoint lock atomicity, asymmetric lock atomicity, and encounter-time lock atomicity. They further identify use cases where weak isolation breaks and how to distinguish between the models. In their evaluation, they compare STM implementations for the different models, and find that strong isolation can actually scale well, but has significant overheads (their benchmarks had very little non-transactional code, though), and that the weaker semantics are usually faster than SGLA, but not always; in some cases the additional tracking required exceeds the additional performance from less restricted interleaving.

Spear, et al, perform similar work to Menon, et al, but deliberately avoid using semantics based locks [182]. Instead, they propose ordering-based semantics that relate transaction order, program order, and how non-transactional accesses order with respect to *local* transactions. Marking only specific transactions as privatising / publishing (in analogy named acquiring / releasing) reduces the overhead required in the STM, while still providing privatisation- and publication-safety when needed. In their results, Spear, et al, show that their *Selective Strict Serialisability* indeed performs significantly better than SLA. Similar to Menon, et al, however, they find that additional relaxations (for transactions that are ordered by anti-dependence) cost most to track precisely and do not offer any additional performance advantages.

In their analysis, Spear, et al, deliberately restrict the semantics to committing transactions. They assume that *doomed* transactions and *delayed clean-up* effects are not visible to applications, and show and compare various implementation options. Complimentary, Guerraoui and Kapalka focus on the behaviour of transactions while they are actually executing, i.e., are still prone to aborting, and define what behaviour such transactions should expect [179]. They propose *opacity* as a correctness criterion that guarantees for all running transactions that they must observe consistent state during their execution, even through they are ultimately going to abort.

2.3 Industry Adoption

2.3.1 Early Industry Approaches

Arguably some of the first work on transactional memory started in industry. Herlihy and Moss' widely recognised 1993 paper shows Herlihy at DEC [34]; and in parallel, Stone, et al, from IBM worked on the "Oklahome Update", and published their work in the same year [35].

Similar to Herlihy and Moss, the Oklahoma Update was an attempt at turning an N-element CAS operation into a more RISC-like structure. Stone, et al, extend the known LL/SC primitives, by allowing multiple locations to be loaded (and then operated on), and then conditionally stored to. One key requirement for N-CAS is, however, that either all stores successfully become the next value in the coherence order, or none. Therefore, having multiple independent store-conditional operations would not be correct. Instead, the authors propose essentially a two-phase commit protocol that performs (1) load-register, (2) store-contingent, and eventually (3) write-if-reserved. The third operation gets all lines in the exclusive cache state and performs an uninterruptible update. Similar to much later work also from IBM, Stone, et al, propose an automatic retry and exponential back-off in hardware. One further key observation of the Oklahoma Update is that when addresses are acquired in sorted order, the algorithm is deadlock-free, even though other write requests are held off.

2.3.2 First Industrial Silicon: Sun Rock and Azul

With the big wave of transactional memory research starting 2004, two companies published about silicon with transactional memory: Sun with their SPARC Rock CPU [188, 201], and Azul with their proprietary CPU used for running Java [199].

Sun Rock Sun's Rock was a radically redesigned CPU, based on the concepts of *scout threading*, and *deferred execution* of stalled instructions. The later (also dubbed execute-ahead) keeps most of the core in-order, and on a long latency operation (such as a cache miss), puts the corresponding instruction *and all its dependents* into a deferred queue, while continuing to execute (and retire) independent instructions.

Stores will buffer their data until the long-latency operation has been resolved. Once that is done, the deferred instructions are executed until they have "caught" up with the independent instructions. In case of resource depletion of the deferred operation structures, the execute-ahead mechanism becomes non-architectural and warms caches and branch predictors. When the long-latency operation completes, architectural execution commences from a checkpoint created then.

Scout threading (or simultaneous speculative threading) uses a separate hardware thread to execute instructions simultaneously when the long latency operation resolves: one thread executes the independent front, while the other executes the instructions from the independent instruction stream.

Rock provides many mechanisms needed for HTM: register checkpoints exist so that execution can turn non-architectural and be rolled back; stores need buffering until the deferred queue has caught up because of the strong memory model of SPARC (TSO [40]). Similarly, loads need to mark their cachelines as speculative so the core gets notified if a concurrent store changes the loaded value. In summary, adding HTM support to Rock mostly exposes the underlying hardware features to applications directly.

Therein lies, however, the biggest weakness of the Rock TM implementation: while the micro-architectural mechanisms need to be fast and correct, they are free to fail and abort the speculation in many cases. Due to the close coupling between the microarchitectural speculation mechanism and transactional speculation, however, these microarchitectural events cause a high number of transaction failures. For example, branch mispredictions, TLB misses, register window overflows (for function calls), etc. can cause transaction aborts. In summary, the programmers have to perform many tweaks to their code to get reasonable TM performance [201].

Overall, Rock is an impressive design; unfortunately, it was never released commercially; according to the publications (the technical report has many more details) the biggest challenges were in validating the design (both the base line and the HTM) due to the many logically parallel instructions executing from very separate parts in the application instruction stream. For the HTM, the biggest challenge was actually to define the logical *commit point* [188].

Looking back at Rock, it becomes clear that (1) decoupling the microarchitectural ILP speculation method and the transactional speculation is advisable for usability and verification reasons, (2) usability of transactions is crucial (including meaningful abort codes), and (3) even support for small transactions (32 stores) can be very useful.

Azul Vega Azul's Vega system is a specialised system designed to be massively parallel, with custom-made multi-core CPUs and specialised to run large-scale Java workloads [234]. For scalability, Vega systems support HTM for Java's `synchronized` methods, effectively performing lock (or rather "monitor") elision; wanting to scale applications that were written for small core counts to their massively parallel systems. In 2009, Click gave a presentation with Azul's design of and experience with their HTM solution [199]. Their cores are simple in-order cores, but up to 54 of them are on a single die. The HTM is implemented through tx-read, tx-written bits in the L1 data cache, and the memory system including the L2 and further is unchanged. The ISA offers the usual instructions, all loads / stores inside the transactions are also transactional. One interesting point is that the register checkpoint of the beginning of the transaction is kept by software, rather than hardware. Click specifically mentions that their deisn is unaffected by TLB misses or branch mispredictions – a clear side-remark at the Sun Rock design. Several applications lend themselves well to HTM usage, yet, according to Azul, the heuristics for when to acquire the lock and when to try the transaction are hard; they resort to profiling at runtime and switching the mechanism.

An additional complication is typical "anti-patterns" that artificially limit transactional throughput, especially with binary code that would otherwise lend itself well to transactional execution: single "number of elements" counters, and centralised performance counters. Often, when the code is rewritten, adding fine-grained locking has better performance characteristics and less erratic performance than HTM, according to Click's experience.

2.3.3 IBM's HTM: A Bouquet of Architectures and Microarchitectures

HTM in the L2 Cache: IBM BlueGene/Q An even more parallel and specialised market is served by IBM's BlueGene/Q system and CPU (BGQ): high-performance computing [267]. The main cores are relatively simple in-order cores, that achieve throughput through being 4-way threaded. The interesting aspect about BGQ is not only that it is the first commercially available CPU with HTM support, but more importantly, that transactions are implemented entirely *outside of the core*: a multi-version L2 cache keeps track of the read / write sets and speculative writes of all the connected cores. Transactions are started and committed through a special interface that is hidden behind a system call API and implemented through memory-mapped I/O to the L2 controller [281].

The implementation of TM in the L2 only simplifies core design, but has several significant consequences: transaction start and end are costly operations, hiding transactional stores from other threads on the core requires either flushing (and subsequent bypassing) of the L2, or remapping of locations to different physical addresses per hardware thread. A significant software layer deals with detecting and handling aborts, register checkpointing with liveness analysis, and other bookkeeping. In a follow-on publication, Wang, et al, dissect TM performance even further and find that the high overheads (118 clock cycles) of just entering and exiting transactions together with the large L2 capacity (20MB for transactional data) and its versioning abilities make BGQ useful mainly for *larger* transactions, and they advocate the use of STMs for small transactions on this system [345]. Another interesting aspect is that small transactions cause the L2 to run out of version numbers for newly written data faster than it can recycle older, aborted / committed version numbers.

Suspend / Resume and Small Transactions: IBM Power 8 In their server-line Power series, IBM released support for HTM with Power 8 [287, 340, 353]. The ISA extensions are noteworthy for two aspects; first, IBM chose to include support for suspending / resuming transactions, for example to support short switches to the OS kernel during a transaction; and second, for integrating TM as a strong synchronisation primitive in a weak baseline memory model. Because Power is not multi-copy atomic, and also permits a lot of reordering of memory accesses locally, Cain, et al, needed to specify many behaviours such as fencing behaviour and transitivity preservation explicitly. Furthermore, Power 8 also specifies *rollback-only* transactions which will not check for conflicts, but allow local rollback of all modifications inside the transaction; this can be useful for trace compilation and optimisation techniques.

In the microarchitecture, Power 8 uses a write-through L1 data cache, with the point-of-coherence being in the L2 (similar to BGQ). Similarly, most of the tracking and conflict detection happens in the L2; the L1 caches therefore have to forward transactional read hits, and the LSU is used for tracking conflicts in in the transit time. For simplicity, Power 8 adds an additional CAM (content addressable memory) next to the L2 to track the transactional properties of the accessed cache lines, rather than extending the entire L2 with transactional memory tracking hardware. The size of that CAM limits the overall transaction footprint to 64 cache lines – which can become a limitation for larger transaction. In this work, I've experimented with similar additional buffers and found that 256 entries can be too small, particularly for the read set of transactions.

While useful, the suspend / resume feature introduces a significant amount of complexity in the microarchitecture and nuances in the architecture, as well. One challenge is how to deal with transactions that abort while they are suspended and which values can be seen by code running while the transaction is suspended. Similarly, communicating values back into the resuming transaction can be tricky. For ASF, similar challenges arise due to the mixing of transactional and non-transactional access; these can be particularly unexpected when they are to two different words of the same cacheline – a case of transactional / non-transactional false sharing.

Configurable and Guaranteed Progress: IBM z-Series Finally, the z-Series marks the third HTM design and implementation in IBM's processor families. According to IBM, the zEC12 is the first commercially available CPU with HTM [270, 276].

Again, the z-Series HTM architecture differs from those in BGQ and Power 8: they add constrained transactions with *guaranteed progress*, have a configurable policy for register checkpointing (programmer can decide to not checkpoint), and they offer an instruction that assists with waiting after an abort has happened using hardware knowledge about the size of the system. Additionally, applications can decide whether to forward in-transaction exceptions to the operating system, or not.

The constrained transactions provide a limited progress guarantee even under contention, as long as the transactions observe stringent limitations for size (instructions and memory operations), and structure (no backward branches). The hardware will try these transactions until they succeed, and can use heavy hammer mechanisms such as full *bus locks* which effectively stall all other cores in the system if all other methods are unsuccessful.

On the microachitectural level, zEC12 also features a write-through L1 and L2 data cache that does not store dirty lines; instead, there is a store-gathering cache that can sink overlapping stores and feeds data into both L2 and the L3. Transactions use this buffer as the versioning widget; therefore, transactions are limited to 64 128 byte cache lines of written data. The L1 also temporarily stores transactionally written data so that the transaction can observe its own writes from there. That cache is cleared on transaction aborts; and values have to be fetched from the L2.

Conflicts are avoided with sending a limited number of NACKs per transaction so that the currently holding transaction has a higher chance to complete. Another interesting detail that Jacobi, et al, mention is that they mark read set entries speculatively (based on branch prediction), because they do not want to add a second access to the L1 data cache when the load becomes non-speculative. I will show more detail for this *transactional overmarking* later in the thesis. Finally, zEC12 uses a clever trick (first found in VTM [86]) to extend the reach of the read set tracking in the L1: they mark an entire set as transactional when a transactional read set entry is evicted and abort the transaction if *any* remote store hits in this set. Due to the inclusive L2 cache, however, this only increases transactions to the size of the L2, as entries evicted from the L2 will *back invalidate* from the L1 and thus hit the set-matching overflow mechanism.

2.3.4 Intel TSX: (Semi-)Transparent Hardware Lock Elision

Intel also released both ISA extensions and CPUs with support for HTM with their fourth generation "Haswell" CPU design [303, 367]. Architecturally, Intel implements a typical best-effort transactional memory system with register checkpoints, transaction start / end primitives, user visible abort, and no option to poke through the transactions or any for of guarantees in their RTM design. An interesting addition to that is the hardware lock-elision (HLE) extensions which is undoubtedly informed by Rajwar's earlier work on lock elision [58, 62]. The idea there is that the application consists of normal lock acquire / release operations and the hardware converts them transparently into transactions (with some additional logic to control the lock variable itself and retry). That way, the same application binary can run on newer systems using TM speculation, and will fallback to standard locks on older systems without any additional code paths.

Unfortunately, there is not a single standard locking instruction or code sequence, and it is furthermore hard to differentiate locks from other uses of similar instructions (such as atomic increment of a ticket lock vs a stats value), and in some cases, determining the polarity of the lock is not straightforward (as the unlock path performs also complex operations). During my work at AMD on ASF, I have found these issues when attempting fully transparent lock elision on arbitrary binaries. Intel apparently faced similar issues, and decided to require (backwards compatible) programmer annotation: `XACQUIRE` and `XRELEASE` prefixes in front of the acquiring / releasing instructions are ignored by legacy CPUs and will cause the switch to transactional execution on newer CPUs.

Intel has not released much detail about their microarchitecture; they do employ the L1 data cache for transactional read and write set tracking, and also employ a secondary, fuzzy structure to track read set elements that evicted from the L1 cache. There is also an additional buffer that holds the lock value during HLE without making it globally visible so that multiple concurrent HLE critical sections on the same lock can execute, yet, each local thread sees the lock as taken.

Despite the simple implementation and the L1 data cache performing most of the work, transaction entry / exit latency is higher than that of normal lock acquisitions and single instruction atomics; therefore, Intel suggests to batch multiple updates together (*lock coarsening*), and removing the acquisition of multiple locks and replacing them with a single transaction (*lockset elision*). Further performance improvements can be achieved by implementing simpler, faster algorithms, or algorithms which are more clever but did not have a fine-grain locking implementation before.

Unfortunately, Intel had issues in the first *three* generations of their HTM implementation [365, 366], and had to switch the feature off. Unfortunately, little is known about the actual detail, but presumably under specific corner cases, the transactional isolation properties do not hold.

In summary, Intel provides supports for transactional memory in mainstream x86 architectures; and support for it is being integrated into standard locking libraries such as glibc [273, 317, 318], and runtime environments, such as Java [315, 324, 330]. Interestingly, all public software changes prefer the explicit transactional elision mode with RTM, rather than the HLE mode. Reasons cited are better visibility of aborts, and more careful control over the number of restarts, and the need to patch the code in any case. Furthermore, it seems that even a straightforward TM system and a commercially very successful CPU manufacturer struggle to provide a fully correct HTM implementation – this contrasts those academic publications which add significant complexity in both ISA and microarchitecture very keenly.

2.3.5 Comparison of Commercial HTMs

Given the size of the design space covered by the different HTM implementations both in terms of ISA, but also microarchitecture, an obvious question is: who has got it right? Qualitatively, the transaction core functions very similarly between all the designs; however, each implementation adds its own architectural feature – there, if a feature actually simplifies software development, it is useful. For now, there is no clear verdict on that, however, first indications suggest that Intel HLE extensions are not that useful in practice, as their added value is small and they hide useful transactional characteristics [317].

Due to the similar core functionality, yet different implementation choices, an interesting comparison is also the quantitative one: how do the different design decisions affect which transactions benefit, how do they limit the overall parallelism that can be extracted in different workloads? Nakaike, et al, compare all four major (BGQ, zEC12, RTM, POWER8) HTM implementations [343], and also look at some of the qualitatively different features. In short, they find that there is no clear scalability winner, but HTM as a feature is overall useful. A summary of the different designs analysed can be found in Figure 2.1.

Comparing the details of the different implementations shows that in particular BGQ has significant single-threaded overheads, but thanks to its large transactional working sets can support applications with larger footprint. Furthermore, zEC12 delivers the largest speadups, while the smaller capacity of POWER8 can sometimes be a limiting factor.

Processor type	Blue Gene/Q	zEC12	Intel Core i7-4770	POWER8
Granularity	8 - 128 bytes	256 bytes	64 bytes	128 bytes
TX load capacity	20 MB (1.25 MB per core)	1 MB	4 MB	8 KB
TX store capacity	20 MB (1.25 MB per core)	8 KB	22 KB	8 KB
L1 data cache	16 KB, 8-way	96 KB, 6-way	32 KB, 8-way	64 KB
L2 data cache	32 MB, 16-way, (shared by 16 cores)	1 MB, 8-way	256 KB	512 KB, 8-way
SMT level	4	None	2	8
#abort reasons	-	14	6	11

Table 2.1: Comparison of the characteristics of commercially available HTMs. From [343].

Interesting conclusions are (1) tuning the retry policy is not trivial, and sometimes retrying even though hardware suggests otherwise is beneficial, (2) microarchitectural interactions can limit performance, such as with Intel RTM causing aborts due to the prefetcher pushing out transactional data or causing additional aborts, (3) high abort rates (80% - 95%) can still produce speedups.

Finally, the additional features can be useful: zEC12 constrained transactions provide similar throughput without requiring careful tuning of retry policies for small data structures; Intel HLE suffers frm the non-tunable retry / abort polcies; suspend / resume in POWER8 can give small performance advantages by allowing transactions to spin on taken locks.

Nakaike, et al, give suggestions for future HTM systems: conflicts should be detected as precisely as possible, especially the interaction with prefetchers can be crucial; SMT significantly reduces the HTM resources, and thus needs to be carefully controlled; non-transactional loads / stores can be useful for debugging, and thread-level speculation; most transactions are small: 10kB fits most transactions with a few outliers using up to 32kB.

2.4 Transactional Memory Use-Cases

Outside of Lock Elision, transactional memory requires changes to applications: locks and critical sections need to be converted to use transactions, lock-free algorithms can be simplified also using transactions. Because of that, the evaluation of transactional memory has a classic *chicken-and-egg* dilemma: applications do not exist, because production support for STM and HTM is not available; that support, however, does not get created, as there is no code-base that could be accelerated.

In the course of the last decade, there was a concerted effort to produce new HW and SW prototype platforms, and create workloads that showcase the potential of Transactional Memory.

2.4.1 Algorithms and Micro-Benchmarks

A typical starting point for performance analysis are micro-benchmarks implementing a simple data structure, such as an integer set. Usually, implementations are based on hash tables, linked lists, and RB-trees and are present in most publications evaluating TM. They are available as part of TinySTM[4] [160]. One key thing to note is that the performance of TM strongly depends on the sizes of the data structures, and the update rate. While larger structures can permit more parallel updates, they also can cause more cache misses and thus exceed TM capacities and make transactions longer. With hardware with support

[4]`https://github.com/patrickmarlier/tinystm`

for HTM becoming widely available, these data structures have been looked at again since; showing interesting performance differences and tuning requirements for different hardware implementations. The main challenges are: transactional tracking capacity, especially when these resources have to be shared by multiple concurrent threads, and tuning retry policies [359, 363].

Several algorithms were shown to perform well with TM, well before hardware support was widely available. Ansari, et al, convert Lee's circuit routing algorithm [4] into a coarse- and medium-grained locking version, and add transactional and optimised transactional versions of the algorithm [177]. Conflicts in Lee-TM are caused mostly on a 2D spatial grid. Kang and Baader show a minimum spanning forest algorithm [204] with TM, in which conflicts occur when multiple threads clash on the same graph nodes. Scott, et al, present a in implementation of Delauney triangulation and show the usability of their revised library TM interface [141, 383]. They conclude that library TM interfaces are still too clumsy, despite their improvement; and they argue for non-indirecting STMs to improve performance.

Algorithms on commercial HTMs Karnagel, et al, show how HTM support can significantly simplify performance of a B+ tree [316]. The authors show that performance with simple elision is good, but can subsequently be improved by making small adjustments to the critical sections to make them transaction friendly. Finally, Li, et al, show how transactional memory can accelerate Cuckoo hashtables and can both be simpler and faster than existing hash data structures [320]. Similar to Karnagel, et al, the authors find several principles that need to be applied to the algorithm to extract most benefit out of the HTM: remove unnecessary shared data (statistics, debug); reduce size and length of transactions; and proper tuning. They also show that custom-made fine-grained locking versions remain viable for maximum performance, but require significant algorithmic redesign.

Makreshanski, et al, use a combination of HTM, locks, and lock-free techniques on tree data structures (B trees, and Bw trees) at Microsoft [341]. They find that HTM generally simplifies data structure design and has good performance, but they find that several performance enhancing techniques when locking are not possible with HTM and collect a wish list for future hardware designs. They argue for adding only selective entries and later removing entries from the working set of a HTX, and capacity and liveness guarantees.

2.4.2 Other Use Cases

Several challenging algorithmic "infrastructure" tasks can be simplified and improved with HTM: Gupta, et al, propose data race detection with HTM [200], Liu, et al, use HTM's strong isolation properties to create consistent snapshots of concurrently running virtual machines [321], and several authors use TM for memory allocation and garbage collection [180, 245, 269, 306, 325, 339].

Another area with interesting overlap with TM is that of security. Guan, et al, use an HTM to protect a private key while performing signing and decryption [338]. They use the HTM to ensure that the decrypted private key does not leave the caches and cannot be read by concurrent attackers or devices. At rest, they protect the key by encrypting it with another master key that is only accessible to the OS kernel.

Muench, et al, use hardware transactions around indirect control flow constructs (such as returning from a function); and use specifics of Intel's HLE implementation to implement labelling [360]. In their basic version, transactions protect the return path to the caller, and limit the amount of damage an attacker can do with techniques such as ROP by disallowing any I/O, limiting the length of the intervention, and rolling back all malicious updates on abort. Furthermore, the authors use the lock value matching (transition $free \rightarrow taken \rightarrow free$) in HLE to implement labelling so that caller and callee need to agree on an ID for the call and return points.

On the side of the attackers, HTM can be used as a fast way to perform side-channel attacks on address-space-layout randomisation. Jang, et al, present such an attack using the timing of transaction aborts as a way to distinguish between kernel addresses that are mapped (but illegal to access from user space), and those that are unmapped [357]. Using HTM, their attack can be executed entirely in user-space, and is much faster and less noisy than comparable techniques actually causing exceptions in the OS.

2.4.3 Language Support

Adding transactional memory support to programming languages is a significant research question. The problem and solution space spans several dimensions:

Programming Language manged languages that are either interpreted or JIT-compiled generally permit more opportunity for unsound / incomplete optimisations ensuring correctness at runtime; their de-emphasis or outright lack of pointers makes it possible to perform more operations "under the hood"; unmanaged languages generally cannot rely on type-safe memory, runtime instrumentation support, or garbage collection

Language and Compiler Support extending languages and compilers to understand transactions generally unlocks cleaner semantics and easier code, and can also unlock optimisation options in the compiler that take into account the specifics of transactions; *library-based* designs, however, allow much faster prototyping and do not require detailed knowledge of and modifications to language semantics and compiler internals

Visility of TM in many cases, especially in interpreted high-level languages, TM can be used in the actual language implementation itself, without being visible to the programmer

Java With its detailed memory model, built-in supports for concurrency, and a familiar, imperative syntax, Java has been subject to significant work with respect to transactional memory. Some of the first software TMs were build in / for Java. Harris and Fraser propose a low-level load / store TM interface directly to the programmer; while Herlihy, et al, use a `TMObject` indirection wrapper class that requires explicit opening of objects for read / write [66, 68]. Later, Herlihy, et al, refine their model to automatically wrap existing classes / objects into their transactional counterparts through a *transactional factory*; similarly, they implement transactions as continuations fed into a *transactional execution core* [112] – extending significantly what can be achieved without changes to the compiler / runtime.

Adl-Tabatabai, et al, however, show that *with* language and compiler support, the programming model becomes very easy to use and the awareness of the compiler and JIT can significantly reduce the overheads associated with STM [95]. They extend the Java language with a `atomic { ...}` keyword, add markers for the transaction boundaries, instrument loads / stores, and perform aggressive optimisations: hoisting open operations out of loops, subsume read under write opens, and inline the fast-path of the read / write TM barriers. Furthermore, drop instrumentation for immutable and transaction-local variables. As a result, they show overheads only in the 20% range even for memory intensive data structure benchmarks. They perform these operations in their production-level Intel Java environment and the resulting tools are not available.

With further optimisations, Shpeisman, et al, later even manage to add strong isolation to that system: they add barriers also for non-transactional accesses and perform extensive hybrid analysis on which objects remain thread private, and which are never accessed from both transactions and non-transactional code [146]. For the much improved stronger behaviour of the resulting system, they manage to keep overheads below 40%.

Finally, Korland,et al, propose a hybrid that does not require modifications to the compiler or JIT, does not require programmer annotation of every object / memory access, and allows simple replacement of the underlying TM implementation [226]. Their tool, called Deuce dynamically instruments Java byte code at class load time and adds callbacks for transaction start / commit, and for every load / store. Their flexible approach does, however, cause significantly higher overheads than those reported by Adl-Tabatabai, Shpeisman, et all. They make their tool available as part of the VELOX TM Stack[5].

More recently, Zhang, et al, revisit STMs for Java and propose a strongly isolating with visible readers, undo logging, and low overheads [347]. With careful tuning of the used locks (using biased locks), the authors achieve low overheads (30% - 70%), despite also instrumenting non-transactional code; which is significantly lower than that of Deuce, and comparable to the Intel Java STM. Despite supporting visible readers (and better progress guarantees), transactions are not opaque because of the lazy read set validation.

Haskell Given its strong functional, side-effect free properties, Haskell requires annotations for shared memory accesses already. Harris adds support for STMs to the language and proposes several higher-level TM language features, such as `orElse` and `retry`. Subsequent work by Perfumo, et al, created Haskell TM benchmarks and characterised their working set sizes and other behaviour [145, 176]; they find that the serialised commit phases is one of the contributing factors for limited scalability. The authors further propose an early release primitive for higher performance [144].

C / C++ Dalessandro, et al, use C++'s advanced type and meta-programming mechanisms to provide a cleaner interface to their library STM [149]. Using a smart pointer pattern, they simplify the interface significantly, reduce clutter, and make the interface much safer to use. However, after porting a larger application (Delauny triangulation mentioned earlier), the need for accessors, and lack of optimisation, they are advocating for for full language and compiler support.

Similarly to their significant efforts in Java, Intel's Wang, et al, integrate TM into their production-level icc C / C++ compiler [132]. They observe that adding such support to unmanaged languages is much harder than their companion effort for Java; mainly because of lack of safety, run-time inspection, and garbage collection. Their optimisations largely resemble those of Java: redundant barrier elimination, fast path inlining, and register snapshotting optimisations. For simple data structure tests (the worst case), they achieve overheads of about 60%, and reduce those in workloads that perform computation, for example to 6.4% in SPLASH2. Wang, et al, use pragmas for code blocks and functions to mark them as transactional and subsequently generate two versions that can be called inside and outside of transactions respectively.

Mirroring the situation in Java, Felber, et al, add a "transactifying" pass to the modular LLVM framework and use that to add transactional memory to C and C++ [150]. They add calls to an STM library for loads / stores inside transactions as a separate pass after optimisations have already removed redundant accesses; and can also run optimisations again to inline the fast paths of those instrumentations by performing whole program optimisation across the application and the TM library. Their annotation tools Tanger and Tarifa are available as open-source as part of the DTMC suite[6].

Instead of annotating entire functions, Crowl, et al, argue for a simpler programmer interface: they suggest simply prefixing a (compound) statement with `transaction` should turn the entire code transactional, rather than relying on the programmer to provide function annotations which they argue is

[5]`https://github.com/DeuceSTM/DeuceSTM`
[6]`https://github.com/basicthinker/PTMC`

too tedious. They also explore the tricky behaviour of exceptions that occur inside of transactions, and suggest privatisation-safe weak isolation, rather than strong isolation.

Yoo, et al, investigate larger applications on top of STMs and find that especially with STMs, the overheads of *automatic* instrumentation (as opposed to manual instrumentation) can be as high as 10x for some workloads (genome in STAMP) [174]. With careful hash table design (false conflicts are a significant problem); filtering of thread private data, and local variables; annotating non-privatising transactions; and even replacing short transactions with a global lock can improve performance of STMs significantly. They authors also argue for compiler support for these techniques and for mechanically creating transactional copies of code.

Invisible Usage of TM Instead of explicit language extensions, or a library interface, TM can be used in the infrastructure of a programming language system. Several publications elide the *global interpreter lock* (GIL) in popular interpreted programming languages such as Python and Ruby.

Riley and Zilles modify the PyPy Python interpreter and execute it on a behavioural VTM full-system model to elide the GIL that synchronises access to the internal structures when multiple threads exist in the application [110]. They also add higher-level primitives such as pause, compensation callback hooks, retry, and alert-on-update to the HTM, and use these for example to also elide locks held by the application. In order to get scalability, the authors need to undo several optimisations that are helpful when executing with mutual exclusion, but needlessly induce conflicts when used in transactions. Several fields that are logically per-thread, are held in single global variables and updated on “thread switch”. Finally, the authors propose an interesting way of dealing with system calls inside transactions: they abort the transaction, but on the retry, push the system call to the commit hook. If the result of the system call is needed earlier, the transaction will abort. Due to the behavioural simulator, Riley and Zilles do not present performance data, but show transaction characteristics for several workloads.

The reference implementation for Python is not PyPy, but instead CPython. Several authors investigate GIL elision for that interpreter, as well: Blundell, et al, use it as the “poster child” workload for their Retcon approach that allows transactions to defer arithmetic operations in HW [236]. Instead of causing WaW conflicts, the authors buffer and subsequently aggregate operations on detected memory locations, and proxy the information locally. Conditional branches that depend on such proxy data are predicted and the mathematical relation is added to a log that is checked at transaction commit. In the analysis, CPython is the workload that benefits the most from this rather invasive technique, because it heavily relies on reference counting that increments and decrements the reference count of objects also for readers thereby destroying all parallelism for readers of said object. The authors evaluate their idea in simulation and after similar code restructuring to Riley and Zilles, and with their technique to permit concurrent reference counting improve the workload from no scaling to almost linear scaling (25x on 32 cores).

Tabba uses Rock prototype hardware and also attempts to elide the GIL in CPython [233]. In addition to the code restructurings earlier, he tweaks the lock (elision) granularity: typically, the GIL is held for multiple consecutive Python instructions (100 by default), but that can cause transactional overflow, so Tabba conservatively only runs a single instruction per transaction. After essentially switching off reference counting, he achieves good scalability for a shared-nothing application with multiple threads.

Finally, Odeira, et al, build upon this body of work, but can rely on commercial HTM support (they evaluate Intel’s Haswell Xeon with TSX, and IBM zEC12 with HTM support). Instead of Python with reference counting, they elide the GIL in Ruby which has a mark-sweep garbage collector that permits concurrent readers [322]. Thanks to hardware support, they are also able to run larger workloads in the improved Ruby interpreter. In addition to the now “typical” code modifications, and similar to Tabba’s earlier work, the authors perform *adaptive* transaction size control: when the abort rate of a transaction exceeds a specific threshold, it will execute fewer Ruby instructions before committing. Other surprising sources for aborts are calls into the garbage collector that of course cannot scan large regions of memory while running inside a transaction, and shared caches that are used for accelerating name lookups, and then get updated on a miss causing conflicts.

Simple lock elision of application locks is, of course, the other big “invisible” use case of TM. Azul originally built their own HTM to elide Java’s monitors and `synchronized` blocks [199]; in the meantime, Hotspot, Oracle’s default JVM, has acquired support for using Intel TSX to the same effect [315, 324]. In the C and C++ world, the Pthread Mutex in the standard C library is the synchronisation “staple”; and also got support for lock elision with Intel TSX [272, 318].

In the course of my thesis, I have added support for ASF to the same code base [7], and my colleague Martin Pohlack experimented with Python (unpublished). We have also published on *semi-transparent* lock elision that replaces the lock implementation at load time (through library preloading), and performs lock elision with ASF, and in cases of failure, reverts to calling the original locking functions [254]. Finally, in the VELOX project, we have evaluated the DTMC compiler framework with TinySTM and ASF for HTM support [210], and added support for transactional memory to GCC.

2.4.4 Benchmark Suites

Several benchmark suites exist that wrap various algorithms and smaller benchmarks into a larger easy to use set.

Guerraoui, et al, were the first to release STMBench7, a TM benchmark that is derived from a database benchmark [140].

Arguably the most popular TM benchmark suite is STAMP, released by Minh, et al, in 2008 [166]. STAMP is is used in many transactional memory publications; and contains a variety of applications. These applications exist both in a hand-instrumented STM version that marks only the critical memory accesses in each transaction in order to reduce the impact of the overheads of STMs; and a HTM version that simply demarcates the transactions and has all accesses be hardware transactional.

Ruan, et al, review STAMP and the criticism it has received, and repair and optimise several several aspects of it [326].

Eigenbench, by Hong, et al, abstracts away the actual algorithmic content of transactions, and instead measures several orthogonal characteristics of them [225]. These characteristics include transaction length, frequency, sizes, and concurrency; and are then used to synthesise transactions that are then replayed into a transactional memory implementation under test. The authors show that they can measure STAMP transactions and achieve similar resulting performance characteristics as running the full application.

Finally, RMS-TM is a benchmark suite by Kestor, et al, and focusses on different application classes, namely recognition, mining, and synthesis [251].

2.4.5 Application Studies

Investigating larger applications is important, because it lifts transactional memory research from the Petri dish into the realm of real code; with unclean patterns and optimisations, interactions that challenge TM semantics, and performance characteristics that allow meaningful decision making for the design of HTM and STM solutions.

Initially, only STM support was available on native machines, HTM was only available in time-consuming and impractical simulation, and compiler support was lacking; with the recent (since 2014)

[7]`https://bitbucket.org/stephand/hotspot-asfsle`

availability of HTM hardware, compiler toolchains, this area has recently picked up a lot of activity again.

Dice, et al, experiment with an HTM simulator and investigate RB-trees, n-ary CAS, and lock elision in C++ STL and libc [184]. They find that debugging (for correctness and performance) with HTM is hard, and several interesting interactions and observations are wiped away in the rollback at transaction abort. One interesting realisation is that JITting can have interesting interactions: if the code is not JITted, the interpreter is likely to cause a transaction abort, undoing the statistics counter that tracks how often a sequence of code is executed.

Click presents on Azul's experience with Java workloads and finds that especially financial modelling applications scale well as they are mostly data parallel, while web-tier application servers often require more tuning to scale to around 50 cores [199]. Click observes that the heuristics for eliding the right locks are hard, as uncontended locks can have lower overheads than uncontended transactions. He points to typical TM unfriendly idioms, such as element counters for data structures, and performance counters. These can be rewritten easily, but have different semantics as a result.

Quake The game Quake has been looked at in various publications: in a first approach, Zyukyarov, et al, convert a Quake server with fine-grained locks to using transactions [206]. They observe that especially the hierarchical spatial locking in the BSP tree that contains the level and the entities becomes much simpler. They also find, however, cases of non-block structured lock usage that require complex code motion to be transformed into transactions. Finally, they find that several functions need annotations for being excluded from the TM mechanism, manually delay I/O (often debug print out), and find issues with the Intel STM C++prototype compiler. In their evaluation, they show that transactions do scale, but observe a 4x - 5x overhead over locks.

The second paper by the same authors, Gajinov, et al, starts with a sequential Quake game server and then uses transactions to parallelise it; again using the Intel C++ STM compiler [202]. As a result, the authors create only eight unique transactions, while the fine-grained version had 58. With ten man-months of work, they still suffer from overheads of 3x - 6x compared to locks, depending on the player count; and find that game physics only accounts for a small fraction of the run time and the application spends about 85% of time inside transactions. This a strong contrast to the overheads observed in [132], but can be explained by the fact that SPLASH-2 (the application suite used for benchmarking there) workloads spend significant less time in critical sections (max. radiosity with 22%, geomean 2.6%). Figure 2.6(top) shows the performance of the different Quake solutions. The authors invent a progress meter for aborted transactions, called *reach points* which effectively are breadcrumbs implemented by non-transactionally incrementing a set of counters corresponding to lines passed in a transaction.

Finally, Lupei, et al, only extract the spatial game tree from Quake and use it to run synthetic game scenarios with a high number of agents (600 - 2k), and manually instrument the data structure with their own libTM STM library [227]. Interestingly, in their use case, game physics plays an important part in the result: with game physics enabled, the STM overheads are amortised by the better scalability compared to a fine-grained hierarchical approach at two threads; without physics they too incur a roughly 4x overhead over the fine-grained locking scheme; see Figure 2.6(middle, bottom).

Testing GCC TM Support Skyrme and Rodriguez port the Lua interpreter with the luaproc package from PThread primitives to using GCC's TM support (with an STM backend)[298]. Luaproc is a different case to the programming languages discussed in more detail, below, as it does not share memory between Lua co-routines, and also does not employ a single interpreter lock. Instead, programmers have to explicitly send messages between concurrent processes. The authors find that half of the locks convert easily to transactions along the obvious transformation `lock(L); <stmt>; unlock(L);` $\rightarrow$ `tm_atomic { <stmt>; }`. There are, however, significant challenges when converting condition variables, especially

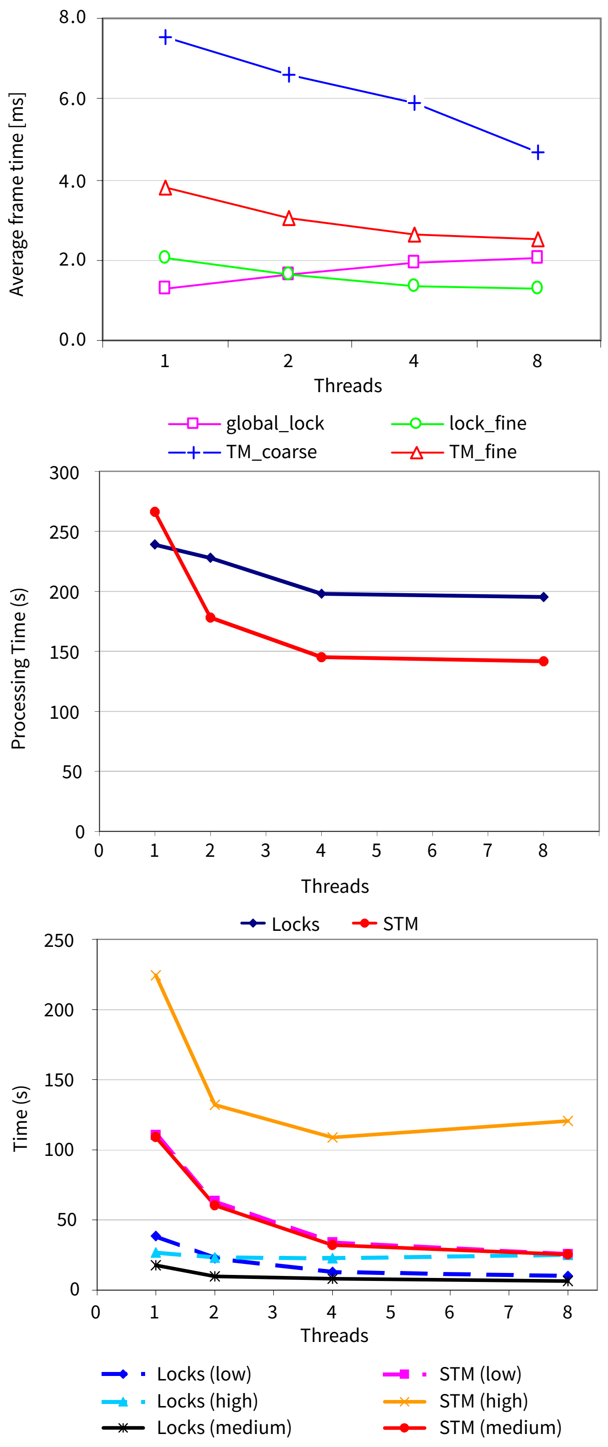


Figure 2.6: Performance of various implementations of Quake game logic.
Top: coarse and fine-grained locking in AtomicQuake and QuakeTM. From [202]. Middle: STM and lock performance of SynQuake with physics. Bottom: removing physics in SynQuake with various contention levels. From [227].

around the synchronous channels used for communication in the application. Skyrme and Rodriguez use *relaxed* transactions that can resort to acquirnig a global lock in case of unsafe transaction content. Their resulting prototype with STM is about 2x slower than the locking version.

In contrast to that, Vyas, Ruan, et al, experiment with memcached and report their findings of replacing PThread-based synchronisation with GCC's TM support [327]. Similar to Skyrme and Rodriguez, they find that some locks convert easily, while several patterns require code transformations: conditional synchronisation, and multiple and non-local unlock operations. In contrast to the Lua work, the authors here use the stricter *atomic* mode for their transactions, because they illustrate that the relaxed mode can easily stumble over an unsafe construct and then acquire the global lock. Very unfortunate examples of this are for example nested PThread lock acquisitions (even uncontended) that are marked *unsafe* and force serialisation of all transactions. With the stricter `tm_atomic` primitives, the compiler at least highlights the problem and subsequently forces transitive transactification for those locks as well, even though they would have not contended. Furthermore, Vyas, Ruan, et al, deal with other unsafe operations through reimplementation of some functions, marshalling of data to thread private locations and then marking the original functions as `tm_pure`, and postponing I/O to `onCommit` handlers. They finally remove the GCC-internal reader / writer lock for atomic blocks that is there to guard them against relaxed blocks going non-speculative, and show performance very close to that of the original fine-grained locking memcached.

Early Hardware Applications Dice, et al, use a Sun Rock prototype to explore data structures (double-ended queues, work-stealing queues, scalable-non-zero indicators) and larger algorithms (memory allocation, simulated annealing) with HTM support [219]. They find that their HTM makes algorithm design very easy, and generally provides good performance; while STMs perform poorly (8x slow down). They also find that hybrid approaches can have cascading performance pathologies, either when deciding to switch entirely back from STM to HTM, or when requiring instrumentation on the HTM fast-path. Dice, et al, find that transactions are generally short, and they argue against a lock fallback path, because that would thwart composability – which can clearly be worked around by only acquiring a single global lock for the *outermost* transaction. Instead, they encourage hardware providers to an HTM that should have small transactions commit eventually, yet, admit that such specification and implementation of these guarantees would be hard – they also ignore that composition may very well be problematic in this case, too, as that would increase the size of transactions.

Schindewolf, et al, investigate HTM support of IBM BlueGene/Q and find that none of the existing TM benchmarks are structured similar to other HPC applications [280]. They create a new benchmark, Clomp-TM, write a new Monte-Carlo simulation application, and convert Parsec's fluidanimate to run on BG/Q HTM. Overall, they find that transaction overheads are high; needing at least 10 - 20 memory accesses per transaction to ammortise. For the applications, they observe that the simple TM scales better than a coarse lock, while fine-grained locking still remains slightly faster than even a fully tuned HTM system.

Tuning Diegues and Romano further investigate into the tuning of HTM, and develop an adaptive learning mechanism that tunes the retry policy for Intel's RTM [312, 336]. They gain about 60% performance over the best static policy, especially in cases where the latter fails to observe changes of topology (going from one thread per core to multi-threading), or workload (different transaction types in the same workload). They also observe that more complex fallback backs (SGL vs NORec) often perform worse, but do offer additional performance when transactions consistently are too large for the HTM.

Dice, et al, observe similar results, and additionally offer a software optimistic (seqlock-based) code path together with a tuning mechanism [310]. They show that for two workloads (hashmap, and an in-memory database), the adaptive policy can extract more performance than any of the static choices. An interesting case is when for the same data structure the usage changes so that transactions are too large to fit into the HTM resources. Their approach quickly reacts and does not perform wasteful transactional attempts.

Usui, et al, also did similar work but *before* HTM support was available to the public; instead, they perform similar statistics collection for locks and switch between acquiring the lock and running the critical section as a software transaction [235]. They show that their system correctly switches to the STM when the high overheads are amortised by offering more scalability than than the single coarse lock.

Didona, et al, merge multiple STM, HyTM, and HTM back-ends behind the same transactional front-end (for GCC), and implement a recommendation system that learns and predicts the best algorithm to use [355]. They show that their system learns quickly and with low overhead (3%), and tracks the best performing solution per workload precisely.

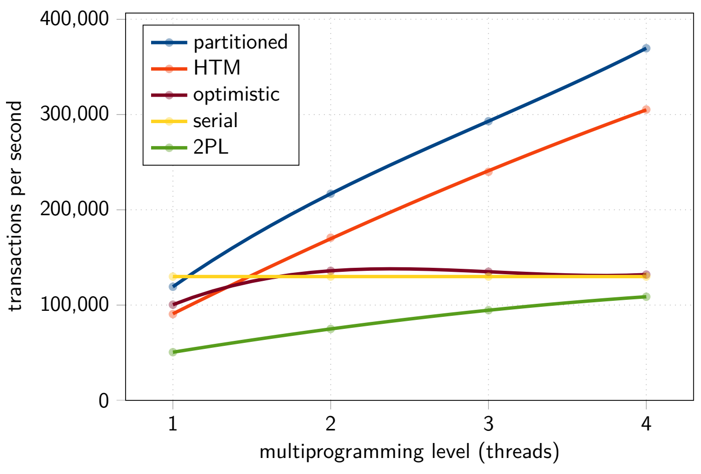
Instead of switching the TM / concurrency control mechanism, another aspect is concurrency control; in many cases, increasing the number of concurrent threads attempting a transaction can lower the overall throughput by causing additional contention. While several precise techniques for scheduling of hardware transactions have been proposed [167, 187], current generation best-effort HTMs do not employ these techniques and generally give very little information about conflict reasons. Diegues, et al, show that with a probabilistic approach of announcing transactions before they are started and recording matrices of concurrent transactions when aborting / committing, they can build information about which transactions to avoid running together [312]. Using a pre-transaction fine-grained lock for these transactions, they show up to 60% improvement of throughput, especially in high contention scenarios under SMT.

Brown, et al, use a similar technique of controlling concurrency for systems with multiple sockets; they find that spreading execution to the other socket can often have a significant performance impact [354]. Instead of controlling transaction pairs, they monitor execution and if necessary, switch through sockets in a round-robin fashion, and let only the designated socket execute transactions (similar to cohort locks [261]).

Fresh Applications One new class of applications are *in-memory* databases. These store their data sets not on disks (either spinning HDDs, or SSDs), but instead in main memory. Interestingly, this means that techniques for concurrency control such as two phase locking (2PC) do not work as well as before, because their costs are not hidden behind the media access costs anymore. Therefore, multiple groups are experimenting with HTM as a concurrency control mechanism for their in-memory DB systems.

Wang, et al, use Intel TSX twice in their system: once in the underlying memory store accelerating a B+ tree and hash table, and then also as the validation and commit engine for their higher-level transaction layer [329]. They use a technique similar to boosting [181] where the respective data structures are thread safe, and the higher-level consistency is ensured via proxies. Their high-level transaction system logs reads (results of data structure query operations) and buffers writes in software, and then uses HTM to validate the results and perform the write-back of the higher-level transaction. A similar method was used for Hybrid TMs by Matveev and Shavit in [305]. The authors use higher level sequence numbers attached to the data store as the proxies for conflict detection. The resulting in-memory data base performs twice as fast as state of the art fine-grained locking versions.

In parallel with Wang, et al, Leis, et al, also rework an existing in-memory database to use transactions [319]. Their approach is very clearly described and very similar to a visible reader version of Riegel, et al, LSA / TinySTM [103, 160]. Again, they do lift these operations, however, from normal memory to be performed in the actual memory store. They use HTM to perform the tracking / update of the read



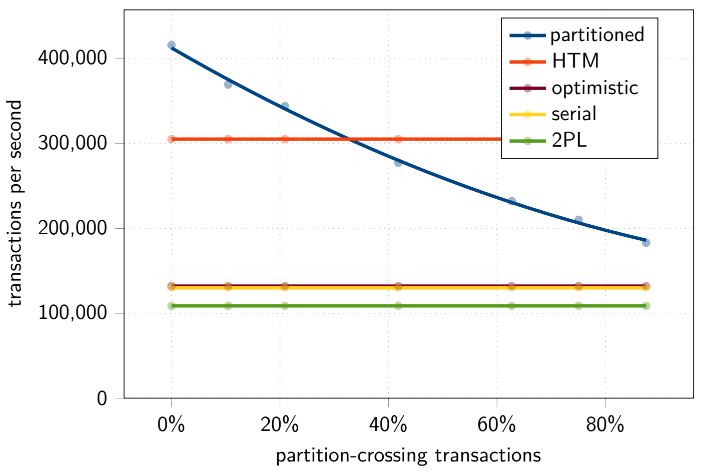


Figure 2.7: In-memory database performance with scalability of different synchronisation mechanisms (left) and varying number of partition-crossing transactions (right). From [319].

/ write timestamps per element, and also for concurrency control in the backend of the memory store. Interestingly, Leis, et al, use the HLE versions of Intel's TSX, rather than manually controlling retry of the smaller transactions. They show that their approach scales as well as a static optimal partitioning approach (and much better than 2PC and a single lock approach), while maintaining performance when the partitioning scheme does not align well with the access pattern; see Figure 2.7.

Finally, Odaira and Nakaike revisit the earlier research area of thread-level speculation (TLS), and try to see if current best-effort HTM solutions without dedicated TLS support can be useful [323]. They manually instrument promising workloads from the SPECCPU 2006 suite, and get a best-case speedup of 11%. In most cases, however, they find that simple best-effort HTM is not suitable for performing TLS. Their main source of slow-downs is not the lack of ordered transaction commits (they emulate that with a counter), but instead loop-carried dependencies that can be forwarded in TLS, but cause aborts in the simple best-effort HTM case. On top of that, they also find that conflict detection granularity on c ache lines needs careful splitting of the loop iterations in order to not cause false conflicts between parallel executions of a loop body.

2.4.6 Transactions and Operating Systems

Transactions Inside Operating Systems In two contrasting publications, Rossbach, Hofman, et al, use the MetaTM hardware transactional memory proposal [124] inside the Linux Kernel: TxLinux is an adaptation of Linux 2.6.16 which already employs fine-grained locking as a baseline [168]. The authors do not just blindly add transactions on some places, but observe and tackle typical *deployment challenges* of HTM in complex software code bases. One example is the interaction with spinlocks: they don't just need to be adapted for elision, but they also need to be *nesting aware*: a spinlock inside a transaction should not actually acquire the lock variable, even if the decision would have been for that lock to not elide. Otherwise, there will be unnecessary WAW conflicts with other transactions on the writes to the nested lock location. The authors thus extend *all* spinlocks to be aware of whether they are being executed inside a transactional context and if so, they do not perform the lock acquisition.

Rossbach, et al, face several idioms that are very challenging to support on any best-effort HTM implementation: locks that are acquired on one CPU and then released on another, and flat nesting which can turn short nested critical sections into much longer ones, as the working set of the nested transaction is checked for conflicts of the duration of the outer, longer transaction.

Furthermore, the authors use MetaTM's suspend / resume mechanisms so that applications can perform short system calls or exceptions. Also, MetaTM itself is interesting, as it restarts transactions with a "restart" code and lets applications explicitly abort, rather than making the abort and restart explicit.

The results for the TxLinux work are mixed: spending 5 man-years, the authors converted about 30% of the locks manually, added 5.5k and modified 2k LoC; and then switched to a mechanised routine. With various benchmarks, especially stressing the file system, the authors achieve speedups of about 5% on 16 cores, while losing 1% of performance on 32 cores. Reduced lock contention provides the speedup in the former, while the aborts cause a net slow-down in the latter case.

Overall, due to the fine-grained locking base line, the kernel spends very little time actually inside critical sections / transactions. The overall speedup is therefore limited. In their own words:

> Transactional memory is supposed to make the programmer's life easier, but by allowing transactions to cooperate with locks, it appears to be making the programmer's life more difficult. However, spinlocks can be converted to `cx_optimistic` with little effort. The resultant code should be easier to maintain because a cxspinlock can be held for longer code regions than a spinlock without compromising performance. Cxspinlocks that are rarely held exclusive can be merged to use smaller numbers of lock variables, further simplifying maintenance. Our experience with TxLinux has convinced us that some data structures can be greatly simplified with transactional memory. However, no synchronization primitive is so powerful that it makes high-performance parallel programming easy.

Contrasting this experience is the retrospective work by Hofmann, et al, where the authors take the Linux 2.4 kernel which still largely relies on the *Big Kernel Lock* and use MetaTM to accelerate that much simpler code base [190]. The authors show that even without expensive virtualized TM support, an appropriately designed best-effort HTM can accelerate a significant fraction (they report > 99%) of the critical sections in the Linux 2.4 kernel. Using the lock as a fallback path, and committing a transaction (or transactional prefix) when they encounter a nested lock that is known to be hard / impossible to elide, they show speedups of up to 40% for filesystem / compile workloads, bringing the much simpler Linux 2.4 big kernel lock to similar performance levels as the much more complex[8] fine-grained locking Linux 2.6 (and beyond) designs.

Invoking the OS from application-level transactions In addition to direct usage of TM inside the OS, another source of TM and OS interactions is applications invoking OS system calls inside of a user-level transaction. In both STMs, and HTMs, these invocations are problematic. With STM, the instrumentation with read / write barriers stops at the kernel level, and the kernel might not read the latest values, or

[8]That can introduce subtle bugs such as `https://dirtycow.ninja/`.

cause silent conflicts. In HTMs, supporting system calls either needs support for cross-protection-level aborts, or a facility for suspend / resume. Even on a conceptual level, the behaviour of system calls can be hard to roll back, as they can involve I/O or other non-retractable operations. Baugh and Zilles investigate system calls in critical sections of Firefox and MySQL and classify their behaviour as proxies for transactions [137]. They make three important discoveries: (1) many system calls can be easily compensated for on abort, (2) syscalls often happen in long transactions, (3) in a significant minority of cases, system calls happen *early* in transactions. In summary, this suggests that supporting system calls inside transactions should not need parallelism restraining fallback mechanisms, if possible. However, they also find that system calls are rare ($<$ 1% of transactions) – unsurprisingly, all but one commercial HTM proposals today do not support system calls inside transactions.

Continuing from Baugh and Zilles' experience, Porter, et al, describe how to make the OS system call interface transactional [194]. Instead of relying on the programmer or library to classify system calls and provide compensation actions, Porter, et al, create the concept of an OS API transaction: adding `sys_txbegin`, `sys_txcommit`, and `sys_txabort` system calls, applications can treat a sequence of syscalls as an atomic unit. Instead of abstractly performing locking, or plain compensation actions, the authors actually instrument the write accesses in the kernel and perform lazy versioning. This comes at a significant cost (the authors report a 4x slowdown for the system calls inside transactions) because it is implemented through a per-object copy-on-write mechanism. It does, however, simplify application development and also fundamentally solves cross-system call atomicity issues, especially around file systems.

2.5 Background

In Section 2.5.1, I will describe the related work of modern CPU architecture, and in Section 2.5.2 show existing work on how current and future CPU designs can be evaluated through *simulation*.

2.5.1 CPU Architecture and Micro-Architecture

Microprocessors came to light commercially with the Intel 4004 in 1971, with 2300 transistors, manufactured in a 10 μm process, running at 740kHz. Today (2017), a typical example CPU such as the AMD Ryzen 1800X is comprised of 4.8 billion transistors, manufactured in a 14 nm process, and can run at up to 4.0 GHz, and also can execute 16 threads in parallel.

Generic CPU design Obviously, in 46 years, many advances in CPU design, and manufacturing were made – many more than there is space for in this section. Standard text books present the key development steps and are continuously updated [123]. Key techniques of CPU designs that are optimised for performance are:

Pipelining splits up the actions of instructions into multiple stages so that each stage becomes shorter, allowing higher clock speed, and multiple instructions can be executed at different stages [14, 20, 60]

Super-scalar execution can execute more than one instruction at the same time, making use of instruction-level parallelism (ILP) and is often combined with pipelining [23]

Out-of-order execution furthermore allows instructions to be executed independently of program order, extracting more ILP and memory-level parallelism (MLP) [1, 10, 38]

Caches store frequently used data and instructions closer to the execution units and reduce access latency and increase bandwidth [12]

Vector architectures exploit data-level parallelism and perform the same (arithmetic) operations on all elements of a vector [2, 13, 43, 161, 376]

Multi-core CPUs extract higher-level (task / process) parallelism and allow multiple application processes or threads to execute at the same time on their own cores, but inside the same system (disk, memory, I/O) [42, 75]

Speculation predicts application and CPU behaviour and allows to shorten the critical path and unlocking more parallelism [37, 50, 77]

Branch prediction is a special type of speculation that predicts whether conditional branches are taken or not, and where indirect branches will point to. That reduces fetch / decode latency, and unlocks a deeper instruction window for out-of-order execution [26, 56, 63]

In addition to extending the structure of CPUs, the instruction set ISA of CPUs is constantly evolving to give applications access to performance enhancing features, and also to add other features, such as security. Currently, two ISAs are dominating the market, Intel and AMD's x86 ISA and its extensions [257, 367], and the ARM ISA [351].

Synchronisation The most relevant aspect of multi-core CPUs and ISAs is that of *synchronisation*. Parallelism and the ability to find and exploit it are great as they make things go fast. In some cases, however, we do want to very carefully control the parallel execution, particularly, when coordinating access to a single shared resource, such as the screen, disk, or when waiting for several parallel computations to end so that their results can be combined.

All modern ISAs provide at least one primitive for synchronisation with infinite consensus number [28]. On x86, compare-and-swap is offered (`cmpxchg`), and IBM Power and Arm provide equally strong load-linked / store-conditional instruction pairs `ldarx / stdcx` and `ldrex / strex`, accordingly [18, 257, 351, 353, 367]. Further extensions are double-wide CAS (for timestamps and pointers to avoid the ABA problem [41]) as `cmpxchg8b, cmpxchg16b`, and many other atomic load-op-store instructions on x86. The Arm ISA recently also got support for atomic load-op-stores in the Armv8.1-A version of the ISA [352].

Even through CAS and LL/SC have infinite consensus number and can emulate all other synchronisation primitives, several extensions to it have been proposed: double compare-and-swap instructions (DCAS) operate on two independent data items. The Motorola 68k processor provided such an instruction and it has been used in OS kernels [30]. Knight proposes a TM-lite proposal with a load / compute prefix and a single store to shared memory [16].

A good recent summary article was written by David, et al, evaluating different synchronisation primitives and how they perform [288]. They compare a variety of synchronisation primitives on a variety of different cache coherent systems. Generally, synchronisation performance in focussed testing depends very much on system topology decisions and choices made in the cache hierarchy. For example, communication within the same socket is usually faster, in some systems, however, *home nodes* are used to provide a central ordering point. These points can cause unnecessary round-trips for messages despite the data living on the same socket. Furthermore, simple locks seem to work well in single socket systems, while more complex locks (hierarchical, MCS, ticket) perform better in systems with more complex topologies.

2.5.2 Simulation

Manufacturing CPUs is expensive, therefore, new features such as Transactional Memory are first implemented in simulators that model key aspects of future CPUs and systems so that performance and operational studies can be performed ahead of manufacturing time.

Similar to the complex CPU microarchitectures that are modelled, a wide variety of simulators has been proposed in academia and commercial offerings. Furthermore, many CPU manufacturers use their own in-house simulators that are sometimes fully home-grown, and sometimes extend other public simulators.

Broadly speaking, there are four types of abstractions for simulators exist:

Behavioural or ISA-level simulators execute instructions and ensure that all instructions behave as if they were executed on a real CPU. They are useful for testing new ISA extensions and usually are used interactively. They only provide a crude approximation of temporal (IPC) and micro-architectural behaviour (such as branch predictors, caches, pipelines). They main goal is typically to provide comprehensive instruction emulation, and high speed. Well-known examples are Simics [61], and QEMU [81]. Slowdowns typically are on the order of $10x$ compared to native execution.

Cycle-level simulators allow performance analysis for new CPU features. As such, they model key micro-architecture features, such as pipeline structures, branch predictors, caches and memory. Sometimes, these simulators also provide a behavioural model in *execution driven* simulation, and in other cases, they are *trace-driven* or merged directly into the execution binary for example through *binary translation* [162, 295]. Due to the added modelling complexity, cycle-level simulators are significantly slower than ISA-level simulators, and typical slowdowns are $1000x - 100000x$. Simulation times can be reduced by limiting focus to the most interesting time periods [69, 361], and the relevant subsystems [93].

Abstract or analytical simulators do not execute instruction by instruction, but instead abstract away low-level details. Ranging from models that inspect the instruction dependency graph to estimate impacts of memory stalls [356], simple stall models [242], parallel phase synchronisation models [344, 382], all the way to fully analytical regression models [57, 76, 78, 83, 96, 97, 99, 100, 264], all of these can significantly reduce simulation time and allow faster *design-space exploration* – sometimes even with mathematical help such as differentiable abstract models and gradient descent.

For my work, behavioural and cycle-level simulators are the most appropriate as we want to study both the ISA-level behaviour, micro-architectural behaviour, and performance in more detail.

Behavioural Simulators Simics [61] is a flexible, extensible behavioural simulator that simulates multiple ISAs, including x86-64 and ARM, and executes full system stacks including the kernels of Linux and Windows. In addition to running applications and operating systems on an ISA level, for example for software porting, Simics has a big API that can hook timing simulation modules for the CPU and devices to the behavioural engine. One interesting aspect is that Simics supports out-of-order execution to drive such timing simulators and can execute instructions with infinite reordering windows and can undo speculation in case of exceptions. Simics is closed-source software, under a proprietary license. Originally (with Virtutech), academic free licenses were available, but these seem to not exist anymore. Simics is often used to drive the behaviour and system IP of academic simulators that focus on the memory subsystem or add a more detailed timing core model. Simics achieves about 5 - 10 MIPS on a 933 MHz Pentium-III host; roughly a 100x slow-down, without additional analysis plugins.

QEMU [81] is an open-source simulator that uses binary translation to provide execution of both full-system and application-only execution of different ISAs on a variety of host systems. In addition to instruction emulation, QEMU also provides emulation of devices. In that capacity, QEMU is frequently used for virtualisation with KVM [125]. Because of the focus on speed, QEMU does not provide explicit hooks for additional timing simulation simulators.

PIN [92] is a proprietary dynamic-binary translation tool that executes applications and allows "mixing" in instrumentation. Instead of emulating instructions, PIN rewrites the existing instruction binary stream, while adding code that provides additional (analysis) functionality, such as calls to cache models, or data flow tracking. PIN executes x86 on x86 only, and depending on the amount of additional instrumentation, the overheads can be very small. Other DBT tools are: DynamoRIO [59], Valgrind [128], fastBT [230], and HDTrans [131].

Several other hardware vendors have provided simulators / emulators similar to QEMU and PIN with focus of faithfully modelling the ISA and system models. AMD's SimNow [260] is used for bringing up new operating system kernels and applications, for example it has been used to port Linux to x86-64 before hardware was available [79].

ARM provides FastModels [369] which similarly is used for platform, OS, and application bring-up.

Cycle-level Simulators Several families of cycle-level simulators exist: gem5 [241] is a combination of the memory subsystem simulator of the GEMS [93] simulator (Ruby) that provides a detailed, configurable memory hierarchy with a lot of detail for evaluation cache coherence protocols, and the M5 simulator [115] which provides the CPU models, easy composability of models, and event-based timing simulation. Gem5 is a full-system simulator and most prominently models both x86 and ARM ISAs (further ISAs are supported, but deprecated), provides a modular system composition with different types of cores, system components, and various abstraction / speed levels. The core of gem5 is an event-driven simulator which fast-forwards simulation models to when events happen.

PTLsim [135], and its descendant Marss86 [253] focus on out-of-order cores and the x86 ISA. Both provide ways to fast-forward simulation to regions of interest and perform full-system simulation. While PTLsim uses the Xen hypervisor [72] as a hardware abstraction layer and supports switching between native execution and simulation, Marss86 relies on QEMU as the fast-forward architectural execution model and switches between QEMU and the detailed simulation core. Marss86 furthermore adds a much more detailed cache hierarchy and coherence protocol over the fully-private cache hierarchy in PTLsim.

Graphite [229] uses a direct execution mechanism that runs applications through a behavioural simulator and collects stats about executed code and feeds those asynchronously to a cycle-level timing model [46]. Over the original work, Graphite focusses on high core counts, user-space only simulation, and can distribute simulation across many physical host machines with lax timing synchronisation. Distribution, lax synchronisation, abstraction and the use of analytical models make Graphite a good tool to inspect large-scale scalability of applications, less so for understanding detailed pipeline and cache interactions.

Sniper [242] extends the concepts of Graphite, but adds more detailed modelling of stall behaviour and quickly executes non-stalling instruction windows, and discovers ILP / MLP of independent instructions if the head instruction is stalled. Sniper is implemented in the Graphite framework, but extends it in the following areas: overall improved memory hierarchy, MSI cache snooping behaviour, better branch prediction, and basic cost functions for thread sleep / wakeup.

More recently, ZSim [295] followed similar approaches by accelerating sequential simulation through the use of binary translation and direct execution, a two-stage parallelisation of the simulation, and finally a small abstraction layer to isolate the user-space only simulation from system-level specifics.

In recent related work, I have helped improve the field of simulation through: abstraction of the core pipeline by elastic memory traces [356], providing accurate power estimation capabilities [364, 375, 379], simulating distributed systems on top of distributed systems [368], and using a novel way of simulator cloning to deterministically model fused heterogeneous cores [378].

For my thesis, I have extended (and maintained) both PTLsim and Marss86. I have implemented a detailed hardware transactional memory mechanism with realistic pipeline and memory system integration, and appropriate instruction mnemonics and extensions to both ISA level and microarchitecture. Furthermore, I have repaired and enhanced both simulators: PTLsim originally only modelled private per-core cache hierarchies which I extended with a simple MSI coherence model; both simulators were not modelling logic for in-order execution of loads which I repaired; finally, Marss86's memory hierarchy is very detailed,but suffered from deadlocks (and other smaller bugs) that I repaired in this thesis [9].

2.6 Summary

2.6.1 HTM Summary

In broad strokes, the research and development of hardware transactional memory investigated mostly the following aspects:

Read Set Tracking, Conflict Detection and Handling Main developments are *signatures* [126, 134] that allow infinitely large working sets with reduced precision, eager vs lazy conflict detection, and differentiating between conflict detection and *handling* the conflict [196]. Further, several pieces of work improve conflict resolution policies to increase throughput in high contention cases [167, 187], and delegate conflict handling to software [105, 171]. Finally, some proposals do not perform conflict detection based on cache coherence messages, but instead use *value-based validation* [248, 265]. Generally, transactional conflict detection is piggy-backed on the existing coherence fabric, especially in eager conflict detection proposals. In cases where the base-line architecture does not provide coherent memory (e.g., in distributed systems [189, 231] or GPUs [248, 265, 291, 377]), or when transactional conflict detection actually *replaces* traditional coherence, a different protocol for explicit working set intersection between transactions is employed.

Data Versioning / Handling the Write Set Key developments in storing tentative transactional data values and the pre-transaction values are: storing either value in an out-of-band structure that can grow in size (SW / HW log) [86] allowing unlimited capacity and context switching of transactions; deciding between which version to store in the "proper" location and resulting faster commit / abort times; and proposals that can dynamically tune these locations and policies based on the workload [196, 228, 282]. Generally, these are more complex as they need to coordinate multi-step atomic memory operations in the memory system, and hold off concurrent accesses without inducing protocol deadlocks. Other proposals store both transactional and pre-transactional values in the same cache [293, 302].

Memory System Architecture and Commit Logic Several proposals optimise the use of directories and use them to accelerate transaction commit without having to broadcast; especially for lazy protocols that execute transactions in isolation and at the end have to perform address intersection with other transactions [102, 138, 171, 196]. Many proposals extend the cache coherence protocol, for example allowing "NACK" messages, timestamps to aid system-wide progress, or new states, state transitions and request types. In addition to the protocols, conflict detection schemes require additional look-ups in other structures, and versioning may provide new sources for data and complex data movement patterns. Effectively all industry proposals [188, 259, 270, 281, 287, 353, 367], however, aim to keep changes to the existing protocols and structures minimal. Most do not provide

[9] `https://bitbucket.org/stephand/marss86-asf`

progress or capacity guarantees (except IBM z-Series [271, 296]), some provide only small transactions [340], while others do not change the entire core, or L1 data cache, but instead implement HTM entirely in an advanced L2 cache with assorted benefits and challenges [267, 281, 345]. As a result, the behaviour of transactions can be erratic and hard to predict [201].

ISA Changes The ISA of baseline HTM looks deceptively simple: a handful of instructions (start, commit, abort) are enough to implement the key features. Some authors analyse the deeper semantics of these instructions [82, 113, 124, 171, 190], while others offer additional instructions to pause / resume transactions, and to optimise the interplay of transactions with other synchronisation primitives and system-level behaviours [124, 168]. Some proposals split out the separate mechanisms of HTM and make them available to software separately, or rely on software to perform some functionality itself [105, 119, 126, 153, 154]. Commercial HTMs offer various additional features: Intel TSX offers hardware-lock-elision [303], IBM Power supports suspend / resume [340], while IBM z-Series offers progress guarantees for limited transactions, and partial register checkpoints [271].

Evaluation Depth Most academic proposals evaluate their disruptive changes to the base-line architecture through the use of simulators; further, they often use a *very simple* core model (in-order, single IPC), and do not run a full system software stack. As a result, proposed changes are often challenging to implement under real-world constraints for system software, and CPU / memory hardware architecture. Notable exceptions are ATLAS implementing TCC [139], parallelised TCC [138], and EazyHTM [196].

2.6.2 Computer Architecture and Simulation

The field of computer architecture has advanced tremendously in the past four decades. Modern CPUs provide several orders of magnitude more performance due to advancements across the entire stack from materials to high-level architecture.

One key concept is *parallelism* which has been used to improve application performance through:

pipelining – shortening the work per stage, enabling higher clock frequencies,

instruction-level parallelism – executing multiple independent instructions at the same time,

memory-level parallelism – performing multiple memory operations at the same time to overlap their stall times,

data parallelism – wide vector units performing identical operations on multiple data items in parallel,

thread-level parallelism – executing several threads or programs at the same time on multiple hardware threads or cores.

Furthermore, *speculation* is used to reduce the impact of critical paths by speculating they will behave in a specific way thereby unlocking parallelism, and rectifying the speculation in case it was incorrect. Finally, using *locality*, the speed of the compute units increased mostly independently from the much slower improvements of the memory system: caches provide low-latency, high-bandwidth access to frequently used data.

The rising complexity in CPU architecture causes rising costs for manufacturing, but at the same time needs more careful tuning of the separate components to provide a *balanced* design. Simulation is widely used in academia and industry to predict performance of specific features in both typical usage and dedicated corner conditions (running stress tests, performing big scalability studies). In addition, simulation (and emulation) allow software and hardware co-design and enable software adaptation before silicon is available – enabling shorter time to market. In line with the complexities of the CPUs, simulators have become more complex too. Depending on the level of detail modelled, they cause significant slow-downs (10x - 100,000x) for the applications that are being analysed. Several techniques exist to accelerate simulation by extracting core regions of interest, sampling, or abstracting the actual simulation step.

In this thesis, I use detailed cycle-level simulation of a modern out-of-order CPU architecture with a realistic memory hierarchy to prototype a realistic HTM mechanism. Basically a mechanism to speculate through critical sections and executing them in parallel so that application critical paths are reduced. One key challenge is the coordination between the different existing parallelism and speculation mechanisms and those that are provided / required by HTM.

Chapter 3

Instruction-Set Architecture and High-Level Design of HTM

3.1 Introduction

Making hardware transactional memory available in a microprocessor involves integration of the required new instructions into the instruction set architecture (ISA) of the system where it is to be used. Before that, however, the architecture level properties of HTM need to be specified and the required new instructions need to be derived. In this chapter, I will describe the key integration points, architectural mechanisms, and desirable high-level properties of HTM on the example of my work on AMD's Advanced Synchronization Facility (ASF).

ASF is an experimental extension to the AMD64 instruction set [257], which in turn is the 64 bit extension to the widely used x86 ISA. Going from the description of generic HTM primitives and architectural functionality to detailed real-world challenges when integrating with a naturally grown ISA, this chapter will provide a summary of the elaborate architectural design process and resulting choices and implications of the chosen design points for ASF.

ASF is the concrete example, but many trade-offs can be transferred to generally integrating best-effort HTM (BeHTM) in other ways and in other base-line ISAs. In fact, with the manifestation of several other industry-grade BeHTM proposals ([281, 284, 340]), the different design choices can be examined in the "wild".

In this chapter, I will give as much background as I can for why specific choices for ASF have made in the way they are and explain reasoning about associated costs and concerns in an industrial setting. The ASF design effort culminated in an *official* AMD experimental ISA specification document for ASF with all instruction encodings and interactions. Since the specification has all the detail, I will only briefly introduce the instructions and refer the reader to the "Advanced Synchronization Facility – Proposed Architectural Specification" which I have attached in Appendix A.

In addition to providing context, summary, background, and reasoning, I will focus on changes (bug fixes, small tweaks and adaptations) and clarifications made after the publication of the specification. I will present major design reconsiderations and options in Chapter 6. The architectural design aspect of ASF has also been described in the following publications:

- "Evaluation of AMD's Advanced Synchronization Facility within a Complete Transactional Memory Stack" at EuroSys 2010 [213]
- "The Velox Transactional Memory Stack" at IEEE Micro Journal [210]

- "ASF: AMD64 Extension for Lock-free Data Structures and Transactional Memory" - in MICRO 2010 [214]
- "Implementing AMD's Advanced Synchronization Facility in an Out-of-Order x86 Core" - in TRANSACT 2010 [220]
- "Compilation of Thoughts about AMD Advanced Synchronization Facility and First-Generation Hardware Transactional Memory Support" - in TRANSACT 2010 [215]
- "From Lightweight Hardware Transactional Memory to Lightweight Lock Elision" - in TRANSACT 2011 [254]

The remainder of this chapter is organised as follows: the remainder of this section will summarise concepts required to integrate BeHTM (Sub-section 3.1.1), Section 3.2 will present the actual ISA extensions, and Section 3.3 will show a simple prototype for integrating the ISA extensions into C / C++. In Section 3.4, I will highlight incremental changes made to ASF to make it easier to use; and in Section 3.5, I will discuss ASF's capacity and progress guarantees. Finally, Section 3.6 will summarise this Chapter and ISA design for BeHTMs.

3.1.1 Architectural Concepts and Functionality of HTM

Before diving into the selection and design of the actual instructions, it is useful to rehash the key high-level concepts in transactional memory in this section, because the ISA specification needs to also talk about properties of the high-level constructions, not just the semantics of the singular instructions.

Transactional memory groups instructions / statements into transactions that operate concurrently on memory locations. From database transactions, transactional memory employs the *atomicity* feature: either all, or no modifications made by a transaction will appear in global memory. In addition, *isolation* ensures that despite concurrent execution, multiple transactions appear as if they were operating in serial order [11]. Most literature on transactional memory merges these two concepts and the fact that transactions order compatibly with their invocation times under *linearisability* [21] or *(strict) serializability*.

Due to the isolation property, transactional memory functions as an effective synchronisation mechanism and can be used to replace critical sections enforcing mutual exclusion [34, 58]. The atomicity property in itself can be exploited for failure atomicity [8], and speculative compiler optimisations [127].

In implementations and architectural discussions alike, three concepts are used for efficient execution of transactions:

Conflict detection tracks the transactional memory accesses and enforces the isolation property so that two concurrent transactions do not form data-dependency circles;

Speculation will attempt to *concurrently* execute transactions assuming conflict freedom, removing a potential sequential bottleneck of parallel code. If conflicts are detected speculation will then rectify such situations; and finally,

Data versioning provides atomicity, by separating pre-transactional values of transactionally modified memory locations and the new, transactional value.

One common mechanism to deal with conflicts are *transactional aborts* that will stop the speculative execution of a transaction and revert all transactional modifications to their pre-transactional values. A *contention manager* with associated policy will make decisions on which transaction is to be aborted and potentially restarted.

In transactional memory (as opposed to databases), the accessed and reasoned about objects are memory locations, it may be possible to access them from both *inside* and *outside* transactions. To make the distinction clear, memory accesses from inside a transaction will be named *in-tx* and accesses to memory that are not (dynamically) inside a transaction will be called *out-tx*. In addition, accesses inside a transaction may be marked to bypass the transactional mechanisms, a concept explored further on in this chapter, and then are called *non-tx* accesses in this work.

Useful idioms for mixing in-tx and out-tx accesses to the same memory locations exist; making it necessary to handle their interleaving on an architectural level: using transactional memory as a mechanism to privatise / publish data [164], interplay of transactions with classic synchronisation primitives, such as locks and mutexes; and finally, other unclassified (often benign data races) overlapping access patterns. *Strong isolation* effectively specifies that memory accesses outside of transactions will be treated as if they are encapsulated in single instruction transactions and thus participate in conflict detection and require proper isolation. *Weak isolation* schemes, on the other hand, avoid the additional handling of such accesses and thus have more subtle specifications / semantics that are outside the scope of this thesis [165, 172].

Finally, *progress* marks an important system-wide architectural level property of transactions. It specifies under what characteristics of transactions (conflicts, working set sizes, durations) the system can specify eventual successful execution for some / all transactions. Unbounded *wait-free* transactional systems will ensure that all (terminating) transactions will succeed in finite time; whereas at the other end of the spectrum, *best-effort* systems do not make any claims about transactional progress at all, but instead attempt to run transactions that fit into resource constraints and do not conflict as effectively as possible.

3.2 HTM Instruction Set Architecture Design

The previous section described the basic mechanisms used for transactional memory; in this section, I will introduce the actual instructions and detailed interactions between them on the example of AMD's Advanced Synchronization Facility (ASF). This section concentrates on backgrounds and high-level summaries, because the full detail is available in the official, experimental ISA extension spec from AMD ("Advanced Synchronization Facility – Proposed Architectural Specification"), which is also attached to this thesis in Appendix A.

3.2.1 ASF 1: A Precursor to Full BeHTM

ASF underwent a transition from two phase transactions to a simple single phase transactional scheme. ASF 1 has two phases, the first being a discovery phase that loads and loads with write-intent (essentially locking specific addresses in read / write mode), but cannot modify elements discovered [158, 159]. This first phase operates a lazy conflict detection scheme where conflicts in the first phase are only detected when transitioning into the second phase. In the second phase, predeclared locations may be modified transactionally, and conflicts cause immediate abort and roll-back.

The instructions used to declare memory locations are normal x86 / AMD64 `MOV` instructions, however prefixed with `LOCK` to signal that they perform special operations. The first section is started with the first declarator instruction and the `SPECULATE` instruction is used to switch to the second, speculative phase which is ended by `COMMIT`. Additional tracking / marking without reading from or writing to the memory location is available with the `LOCK PREFETCH` and `LOCK PREFETCHW` instructions.

In earlier work, I implemented ASF 1 and showed that it was possible to use it not only for small contained atomic primitives such as DCAS where the working set was known, but also for bigger lock-free data structures and for accelerating an STM [158, 159].

```
void f1 (...)  {
  discover_f1 ();
  modify_f1();
}

void f2 (...)  {
  discover_f2 ();
  modify_f2();
}

void compose_f1_f2(...) {
  tx { f1 ();  f2 ();  }
}
```

```
void compose_f1_f2_ASF1(...) {
  tx {
    // Collocated  read−only discovery  phases
    discover_f1 ();
    discover_f2 ();
    // Switch  to  modify phase
    tx_switch ();
    // Read−write non−extending parts
    modify_f1 ();
    modify_f2();
  }
}
```

Figure 3.1: Composition of two-phase transactions into larger transactions. While each transaction can be easily decomposed into a non-modifying working-set extending part and a non-extending, data-modifying part, the composition does not have an obvious easy equivalent transformation.

Despite several benefits (the transaction scheduling can easily be centralised due to full working set knowledge before modifications), the strict requirement to separate transactions into two phases was perceived as a strong limitation by us and software and library writers such as the TU Dresden group working on TM. In particular, composability of operations was impossible due to the required code transformations, as illustrated in Figure 3.1. Additionally, using transactions as a drop in-replacement for critical sections was not possible due to the tighter structural requirements of ASF 1 over free-form critical sections.

3.2.2 ASF 2: AMD's Hardware Transactional Memory Proposal

Because of the perceived weaknesses of ASF 1 identified in the previous section, we designed ASF 2 to allow full dynamic discovery and single-phase transactions with transactional modifications ahead of further discovery of transactional locations.

ASF 2 re-uses the `SPECULATE` and `COMMIT` instructions to start and end a transaction; and keeps the `LOCK MOV` and `LOCK PREFETCH(W)` instructions to perform transactional reads and writes. The concept of declarators (together with the transactional / non-transactional mixing rules explained below in Section 3.2.6) has changed over time. In the original ASF 2 specification, there still was a concept of declaring first (and potentially reading) with `LOCK PREFETCH(W)` and `LOCK MOV` and reading / writing later (using normal `MOV` instructions) with allowed working set extensions. Memory accesses to undeclared memory locations are also possible (*non-tx* accesses), and will bypass the transactional mechanisms, more on that in Section 3.2.5. For simplicity, however, this eventually changed to separating transactional and non-transactional accesses (drop the separate declaration step) and specific rules for mixed-transactionality accesses to a single location (compare, for example, to [214, 220]).

ASF makes the granularity of tracking transactional accesses architectural and fixes it at 64 byte. This means that both conflict detection and versioning happen on a 64 byte granularity. *False sharing* can therefore cause additional transactional aborts and data versioning on 64 byte granularity has complications which will be discussed in Section 3.2.5.

3.2.3 Conflict Detection and Transaction Aborts

In transactional memory, aborts occur if the transactional execution needs to be rolled back and restarted. One design option for HTM ISA extensions to consider is whether to expose transactional abort and restart to the applications or not. STMs often do not expose aborts, but instead hide them, rollback, contention

```
# DCAS Operation:
# IF ((mem1 = RAX) && (mem2 = RBX)) {
#   mem1 = RDI;     mem2 = RSI;    RCX = 0;
# } ELSE {
#   RAX = mem1;     RBX = mem2;    RCX = 1;
# }  // (R8, R9, R10 modified)
DCAS:
  mov       %rax, %r8
  mov       %rbx, %r9
retry:
  speculate                  # Speculative region begins
  jnz       retry            # Page fault, interrupt, or contention
  mov       $1, %rcx         # Default result, overwritten on success
  lock mov (mem1), %r10      # Specification begins
  lock mov (mem2), %rbx
  cmp       %r8,%r10         # DCAS semantics
  jnz       fail
  cmp       %r9,%rbx
  jnz       fail
  lock mov %rdi, (mem1)      # Update protected memory
  lock mov %rsi, (mem2)
  xor       %rcx,%rcx        # Success indication
fail:
  commit
  mov       %rax, %r10
```

Figure 3.2: ASF example: An implementation of a DCAS primitive using ASF.

policy, and restart behind a library interface [95]. For HTMs, it is desirable to make aborts *visible* to the software layer from a hardware / software interaction perspective. Visible aborts allow the application / run-time to schedule transactions (for example through randomised exponential back-off), or employ other modes of concurrency control (such as invoking an STM or global lock layer). Employing these modes transparently beneath the ISA is costly, as I will explore in Chapter 4. Instead, ASF informs software-policies with additional information about the abort reason and whether the abort was of a *transient* nature (data conflict, interrupt), or will likely occur again in a future reexecution (capacity exceeded, programming error).

In ASF, aborts are therefore visible to the application. To limit the amount of wasted transactional execution and non-transactional state modification, ASF aborts the conflicting transaction as soon as the conflict is detected (*eager* conflict detection); I will detail challenges in the interaction of both eager conflict detection and non-transactional accesses in Section 3.2.6. Conceptually, the eager, user-visible abort is a user-level exception. Since there is no precedent that could hint at the design in x86 / AMD64 for lightweight user-level exceptions, ASF was free to employ its own way to implement them. To avoid arbitrary control-flow jumps, the SPECULATE instruction functions as an anchor for aborts. Upon an abort, execution is directed to SPECULATE and the instruction seems to execute but produces an error signal ($rAX \neq 0$, and set FLAG.NZ) as opposed to the clean transaction start signal ($rAX = 0$, and $FLAG.NZ = 0$). This way, the application can employ a conditional branch and go to an abort handler. A simple example is to directly retry the transaction, as depicted in the DCAS example in Figure 3.2.

Conflict detection deals with at least two participants conflicting; usually classified in a *requester / holder* side depending on which transaction adds a specific data item to its working set first (being the holder) and the transaction causing the conflicting access later (the requester). On an architectural level, the decision which of the conflicting transactions is aborted can have a significant impact on the transactional system and its throughput guarantees under contention. Owing to the complexities of the mechanisms on which ASF bases its conflict detection capabilities (the cache coherence protocol, more detail in Chapter 4), ASF employs a simple local scheme that favours the requester (*requester-wins*) and always aborts the holder of conflicting transactional data.

ASF detects conflicts between transactions and concurrent overlapping memory accesses not executing inside a transaction (out-tx accesses); implementing *strong isolation*. Together with requester-wins, this gives priority to code not running inside transactions. The combined effect is not accidental, but instead a design feature. During ASF's design phase, it was deemed important to not give applications a way to indefinitely stall (privileged) software when accessing (user) memory. Instead, aborting the transaction and not stalling the out-tx memory access ensures that out-tx are never unduly delayed and the instruction stream executes in a wait-free fashion. In hardware implementations which use the cache coherence protocol, strong isolation is easy to implement (Chapter 4). ASF therefore does not suffer from the issues associated with weak isolation in STMs, and is inherently publication- and privatisation-safe.

3.2.4 Nesting of Transactions

One desirable feature of transactions is the ability to nest transactions. Various types of nesting exist, and are evaluated in particular in the world of language semantics and STMs: *open* nesting (where nested transactions can commit independently of the parent and use higher-level undo logic in case the parent aborts) [129], *closed* nesting (where child transactions can effectively checkpoint the parents transaction's progress and allow partial aborts of the parent back to the last successful child transaction) [85], and various forms of *parallel* nesting (where parent and child transactions can execute concurrently on different cores) [212]. These elaborate nesting schemes are costly to implement due to the required differentiation between working sets of differently nested child transactions. Instead, ASF supports simple *flat* nesting that flattens successive starts of transactions through executing `SPECULATE` when already in a transaction. These *nested* `SPECULATE` and `COMMIT` instructions will effectively become no-ops (and just signal successful transaction entry through the supplied register). Architecturally, they will in-/decrement a nesting counter that is available upon abort. The nesting depth is limited, mainly to provide a limit on the size of the nesting bit-field and counter implementations and transactions that nest too deeply will simply abort.

Due to the flattening, aborts will always jump back to the outermost `SPECULATE` instruction, discarding always all accumulated transactional state. Furthermore, the transactional state of all child transactions is live (and checked for conflicts) for the duration of the entire outermost transaction.

This simple nesting regime allows simple composition of small transactions into larger building blocks, but does not support support some of the more sophisticated nesting uses reported in the literature directly [156, 197, 211]. The accumulation of working sets also makes this mostly useful for combining small primitive transactions into somewhat larger primitives, for example combining multiple transactional N-CAS operations, or linked list removal and insertion into an atomic move operation.

I will look at more interesting nesting cases in Chapter 6, which show how ASF can be used to support various forms of parallel open / closed nesting, to increase performance and to bypass architectural limitations of ASF.

3.2.5 Non-transactional Memory Accesses

From a bottom-up perspective, the historical requirement in ASF to mark (or rather declare, see Section 3.2.1) transactional accesses naturally evokes the question about how unmarked memory accesses inside a transaction (non-tx) should behave.

Two important cases need consideration, here. First, the non-tx access is disjoint from the transactional working set; and second, there is an overlap between the non-tx access and the transactional working set. Historically, ASF separated the declaration (first phase in ASF 1) and actual access / modification (second phase in ASF 1) and thus treated the overlap case as the access / modification to a previously marked line.

With the increased flexibility of ASF 2 transactions, the split was eventually not mandated and thus ahead-of-time marking became a special case. Originally, unmarked accesses that would not intersect with the working set (marked cache lines), were simply not treated specially by ASF, but instead bypassed all transactional mechanisms.

Eventually, this implementation-driven feature led us to contemplate whether the feature was actually useful and not just a byproduct. Examining transactional accesses, we find that they require both conflict detection and versioning; non-transactional memory operations may weaken either or both properties. Since loads do not modify global state, they do not require versioning.

Non-transactional loads therefore could be specified to not participate in either or both “ends” of conflict detection: *requester* conflict detection where the load will send out conflict messages (coherence snoops) to other agents in the system; and *holder* conflict detection, where loads check for incoming conflict messages and abort the local transaction if a conflict is detected. Due to the strong isolation property, ASF sends out conflict messages for non-transactional out-tx accesses already (in fact they are part of the existing coherence protocol, as I will explain in Chapter 4). It seemed only natural to maintain this property for non-tx loads and so their participation in the requester side of conflict detection is necessary. This change also coincides with the mapping to the cache coherence protocol, effectively treating in-transaction non-transactional loads as simple out-of-transaction loads and not have them participate in the holder-side of conflict detection for them.

As a result, non-transactional loads will not abort the surrounding transaction and can therefore be used to monitor changing outside data from within a transaction. Interesting idioms are for example in lock elision, where non-transactional loads can be used to monitor a taken lock and only when the lock becomes free add it to the transaction’s working set.

Similar reasoning regarding the conflict detection applies also to non-transactional stores; participating in the requester-side of conflict detection ensures cache coherent operation. Data versioning is specific to (non-transactional) stores. For ASF 2, the non-transactional operations were largely thought of as a means to preserve precious transactional tracking resources, thus, employing data versioning for non-tx stores was not considered. Instead, non-transactional stores will not be rolled back upon transaction abort, making them *immediate* non-tx stores. Their visibility therefore depends on the their execution timing relative to transaction aborts, and on the existing memory ordering model. A specific usage for such non-transactional states is in debugging, for example in the ReachPoints work [202].

An alternative would be to have non-tx stores perform data versioning, but not participate in conflict detection. The written data would then only become visible when the enclosing transaction commits (*delayed* non-tx stores). This is somewhat similar to lazy conflict detection, where stores are checked for conflicts only at the end of a transaction; instead they are not checked for conflict at all here. We believe that the unreleased Sun Rock processor [188] may have supported this type of non-tx stores, but we are not aware of publicly available documentation to confirm or provide reasoning or use case examples.

3.2.6 Non-transactional Access Challenges

Accessing known thread-local data through non-transactional accesses (for example the *stack*) removes resource requirements and leaves more capacity to the critical synchronisation-specific code sequences, and allows controlled leakage of transactional state for diagnostic purposes. For that reason, the standard AMD64 stack manipulation instructions `PUSH`, `POP`, `CALL` and `RET` operate non-transactionally and are not

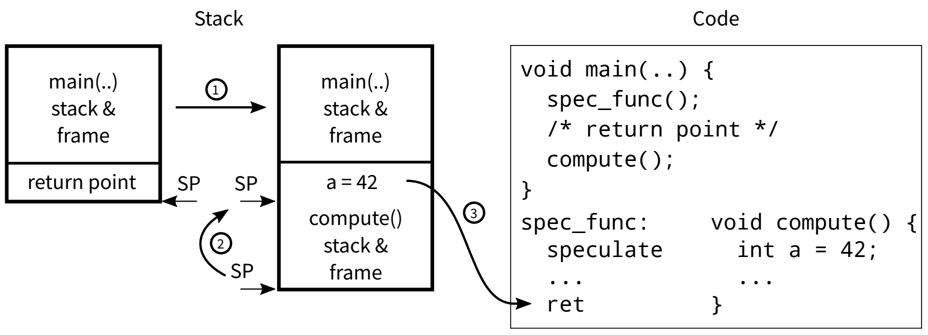


Figure 3.3: Non-transactional modifications of the call stack can overwrite call / return chain information outside of the transaction scope: just after `SPECULATE` is executed inside of the nested function `spec_func`, the stack looks as in the left picture. Inside the transaction, the `spec_func` returns using the return point saved on the stack, and eventually calls the `compute` function which puts a local variable (`a`) and other data on the stack (1). If the transaction is aborted at this point, the instruction and stack pointers are rolled back to just after the execution of `SPECULATE` (2). Now, however, the return point is overwritten on the stack and the `spec_func` returns to a random, wrong location (3).

(and cannot be) marked. During the ASF design process, this feature was deemed important, since in particular 32 bit x86 code has only few truly general purpose registers available (six), and therefore has to spill live registers to the stack frequently. Removing these accesses from the transactional working set meant conservation of precious transactional resource for the actual transactional functionality.

Two complications, however, arise even in the simple settings. First, non-transactional modifications of the call-stack could overwrite return addresses non-transactionally, as explained in Figure 3.3. The problem occurs if the `SPECULATE` instruction is called from within a function at a deeper call stack than where the abort occurs, or in general when starting a transaction through a nested function call.

In my transaction integration prototype (introduced in Section 3.3), this is remedied by enforcing inlining of the transaction start primitive, the TUD-developed code integration solves this in two steps: in the compiler, the instrumentation logic captures all escapes from a transactional block (similar to exception and destructor handling in `try` / `catch` blocks) and inserts the appropriate calls to the TM library for transaction entry and exit. The library itself carefully uses techniques similar to `setjmp` / `longjmp` and also copies part of the live stack as a transactional backup copy[1]

Multi-word non-tx stores The second issue with non-transactional stores is that they essentially race with a potential transaction abort. Similar to operating system kernel code and interrupts, the abort can happen at any time and may interrupt multi-instruction non-transactional (store) sequences at any time. Complications arise when trying to synthesise higher level cross transaction communication primitives, and I show how to solve these in [274] and Chapter 6.This is one of the benefits of using a suspend / resume mechanism to escape from transactions: the non-transactional code will run to completion before transaction aborts are handled after resuming the transaction.

With Michael Hohmuth, I worked on a *roll-forward* mode of ASF that avoided immediate aborts, but unfortunately, that work has not been published or fully implemented in the simulation tools. Instead, I created an alternative suggestion to briefly continue aborted transactions through the concept of *resurrecting* transactions, which has been applied successfully to handle arbitrarily long sequences of non-transactional code within transactions in the face of immediate aborts. This work is available as [289, 337], and Appendix B.7, and will be explained further in Chapter 6.

Overlapping accesses The final problem with non-transactional accesses arises from overlapping transactional and non-transactional accesses inside a *single* transaction as described earlier. Accessing a single

[1]After personal discussion with Torvald Riegel and Martin Nowack. Thanks!

data item (or cache line!) both transactionally and non-transactionally has complicated promotion rules. While a non-transactional access followed by a transactional access always transitions the data item into transactional (earlier non-transactional stores are made visible, first), the reverse combination is more complicated. The proper semantics of this mixing were a topic of intense discussion during the ISA design phase at AMD.

Transactional loads will “flip” the data item to being transactional, enforcing conflict detection from the point when the data has been loaded transactionally. Subsequent non-transactional loads will not remove the transactional handling (the `RELEASE` instruction does that). The high-level reasoning is that the “do not track” property is a performance optimisation and can thus easily be upgraded (“accidentally”) by overlapping transactional loads, which need to remain transactional to enforce atomicity / isolation. Incidentally, this means that the “peek then add to read set” idiom works well with non-transactional loads.

Handling (later) non-transactional stores that overwrite transactional data (produced by earlier transactional stores) caused discussion and changes during the design process. Clearly, not both immediate store availability and continued participation in transactional conflict detection and data versioning can be achieved (while also observing per-location access ordering). Since no clear consensus towards favouring immediate visibility over transactional semantics could be reached, the initial design caused this overwrite pattern to cause a general protection fault with transaction abort, indicating a programmer error [186, 214]. For experimentation with compiler integration[2], however, I added a mode of transactional stickiness, where non-transactional stores are also seen as a performance optimisation and can safely lose their non-transactional status; gracefully handling the case where the non-transactional accesses are caused by escape analysis identifying thread-local variables rather than a desire to communicate.

This shift in semantics is interesting in particular in light of the history of ASF, and the associated change in the programming model gives rise to this conflict. Initially, there was the definition of declarators, where one needs to declare data for transactional store and then all stores were transactional automatically, and transactional stores to undeclared items would constitute a programming error. This eventually weakened (strict separation in ASF 1, mixed declare-then-use and discover more in initial ASF 2) with the desire to lose the explicit declaration. Enforcing marking on all transactional stores to distinguish them from intentionally unmarked immediate non-tx stores then created the conflict of intention: ephemeral transactional item / cache line vs. later intent to make immediately visible.

The mixing behaviour of transactional and non-transactional accesses is particularly important when transactifying existing code without detailed control over data layout. In particular with ASF’s architectural tracking size of 64 byte, the “transactionality” is always tracked on a per-block (cache line) basis. Allowing reasonable promotion rules without inducing additional transaction aborts was therefore crucial for these use cases. For full ISA integration, I can envision a mode switch that enables the strong separation if desired, or toggles laxer promotion rules for reduced number of aborts, similar to exceptions tracking mis-aligned memory accesses.

3.2.7 Limited Register Checkpointing

In addition to memory memory modifications, the register state of the CPU is also changed during a transaction. To allow for rollback / restore of that state in case of an abort, register state versioning is desirable for example through taking a full register snapshot at the beginning of the transaction. In ASF, however, we have adopted a *limited snapshot* approach, where only a small set of registers is being saved on transaction entry. All other registers are not saved and not rolled back in case of an abort. The decision was largely made due to the expectation of short transactions and to avoid the overhead

[2]After discussion with developers from the TUD STM stack, Torvald Riegel and Martin Nowack.

of taking a full register checkpoint at each transaction start; also reducing implementation cost by not requiring a separate full register file and register write tracking logic.

In addition, the limited register checkpoint can be used for controlled state export from an aborted transaction, in a concept similar to ReachPoints. The minimal register checkpoint consists of the instruction pointer `rIP` which is saved at the beginning of a transaction to allow return to the `SPECULATE` instruction. To simplify save / restore of data that is live at transaction entry and changed during the transaction, the stack pointer `rSP` is also saved and restored. This allows simple stack-based spilling of registers needing save / restore.

In my prototype integration library (Section 3.3), I will show that the compiler can usually do a very good job of tracking live registers at the beginning of the transaction and perform save / restore actions only on them. This is particularly important and applicable for small transactions where the compiler can inspect all code. In larger environments where the compiler has limited control over / knowledge of the code running inside a transaction (for example in the TU Dresden DTMC stack), software will create a full register snapshot at transaction entry and abort (saving only the callee-saved registers when integrating with a function).

Revisiting a design decision that had largely been driven bottom-up by the urge to simplify hardware support–if software can easily emulate the required functionality–led to an interesting observation that having access to the *full* register state at the transaction abort site opens up for interesting high-level use-cases, such as introspection, debugging, and transaction resurrection which I will discuss in Chapter 6.

3.2.8 Interactions with the Memory Model

Two transactional memory transactions must appear linearisable with respect to each other's concurrent execution (this is the *isolation* property in the classic ACID transaction model). Thanks to strong isolation, transactions also linearise with out-of-transaction memory accesses.

For the memory model, this means that all transactional memory operations inside a single transaction become globally visible at once and all observers agree on the order of transaction commits (*multi-copy atomicity* [29]).

In ASF, in-transaction memory accesses are split into two groups, transactional and non-transactional. ASF's in-transaction non-transactional accesses do not need to become globally visible at the same linearisation point as the surrounding transaction. In fact, enforcing such ordering would remove a large number of both architectural and microarchitectural benefits of those instructions: making stores visible with the linearisation point conflicts with the concept of immediacy of the non-tx stores and also forces them to become visible at a *single* point in execution, which breaks any use case where an external observer needs to observe non-tx accesses sequentially (for example in communicating transactions). In the microarchitecture, the need to buffer non-tx stores will require buffering resources and thus complicate hardware design.

Transactional accesses follow the linearisation model so they will linearise at a single point during the lifetime of the transaction, and non-transactional accesses follow the existing AMD64 memory model [209]. Non-transactional stores can become visible at any point after their execution and can reorder with transactional stores, but not with non-transactional stores (AMD64 mandates a TSO-like memory model [193]). A memory fence before transaction commit can enforce all non-transactional stores to become visible before the transaction commits.

The original ASF 2 specification did not fully define other interactions between transactional and non-transactional accesses. In particular, it did not establish order between non-tx and tx loads; all existing implementations, however, honour data and control dependencies between transactional and non-transactional loads (effectively ordering tx and non-tx loads by treating them both as out-tx loads).

Finally, after discussion with Torvald Riegel, et.al., for their recent hybrid TM publication [255], it became clear that they needed stronger properties between visibility, abort and state of the transaction working set for their hybrid TM. The property in particular is that in a program ordered sequence such as

$$\texttt{SPECULATE} \ldots \texttt{ld}_{\texttt{tx}}; \ldots \texttt{mem}_{\texttt{nontx}}; \ldots \texttt{COMMIT}$$

conflict detection for a transactional load ld_{tx} detects conflicts before a non-transactional memory instruction mem_{nontx} can make non-transactional changes (and even before its load is globally visible) to memory. Finally, if mem_{nontx} is an atomic RMW instruction (or ordered through a fence), its modifications will become globally visible before the conflict detection for ld_{tx} is completed and the transaction commits (and all transactional modifications become globally visible). In the proposed HyTM scheme, this property is required to ensure that a transactions working set and the modification of a non-transactional synchronisation mechanism (a generation count / lock) occur at overlapping time intervals and thus are linearisable together.

3.2.9 Instruction Support and Non-Conflict Sources of Aborts

Several instructions and modifications of architectural state may be hard to support in HTM. Three main causes are responsible for these complications:

Complex state that is hard to save / restore and rarely changed, for example segment registers, execution state (kernel vs. user-space), page-table modifications, I/O, and device state modifications

Non-transactional modifications in code that is called from a transaction, but is unaware that it might be aborted and then may leave inconsistent global state behind; I will describe the usage of transactional resurrection to remedy this in Chapter 6

Entangling independent code for example when time-sharing the transactional facility between two applications on the same core, or between the application and running OS kernel code; this either requires additional logic to allow tracking of multiple in-flight transactions, silent aborts, or entangles the progress of otherwise independent code sequences in unpredictable ways

The complications arise usually at context switch / protection level switches and when executing specific instructions. As a universal remedy, ASF therefore will abort transactions upon encounter of an unsupported instruction (including calls to the OS), any interrupt and exception.

One noteworthy special feature in ASF, is, however, that ASF will forward the exception / interrupt to the OS. This is somewhat expected for interrupts, but for page fault exceptions, this means that a page fault may propagate out of an aborted transaction and originate not from a memory instruction. We were careful to reflect the abort in the register state of the process *before* transferring control to the OS so that all abort information is already stored in the general purpose registers (`rAX`, `rIP`) so that the OS can be completely *unaware* of ongoing transactions and their execution state and does not have to save / restore this information as part of a context switch. This makes ASF essentially backwards compatible with old, ASF-unaware operating systems.

The strong isolation and immediate asynchronous aborts in case of conflicts reduce the chances for seeing inconsistent data (as opposed to STMs where issues of late aborts can make transactions temporarily consume inconsistent data and thus more rigorous sandboxing is required), and we therefore think that avoiding the case where a transaction is repeatedly aborted due to demand paging (and the causing page fault causing a transaction abort, the restarting transaction running into the same exception) warrants such a deviation from non-leaking properties to improve transactional progress. Other practitioners were aware of this challenge, but gave higher priority to not changing the "exception signature" of applications running transactions (causing page faults and other exceptions from an aborted transaction state) due to compatibility concerns[3]. Our inspection and experimentation with the Linux kernel showed this to not be a problem, instead, the kernel resolves the demand paging request and the restarted transaction will continue successfully.

Exceptions are caused by the application (and the OS software stack) and we therefore expect them to have a predictable effect on transactions; in particular with demand paging eventually stabilising (assuming a reasonable OS that maps enough pages for the process). Interrupts, on the other hand, are usually not controllable from user space processes. (The OS kernel can use the `cli` / `sti` instructions to control interrupts around ASF transactions.) In particular in high-I/O-load scenarios (40GbE network card, for example), the resulting interrupt frequency may cause long transactions to fail due to the interrupt-induced abort of the transactions.

One option that we discussed for the ASF design is the option to *delay interrupts from user-space transactions* for a specific maximum number of cycles. By limiting the total number of cycles such an interrupt-disable would be in effect (and potentially adding rate limitations of how often such an instruction will be honoured), it is possible to reduce the chances of interrupts repeatedly aborting a transaction, thereby ensuring a minimal transaction length that can be executed without aborts induced by interrupts.

3.3 Language Integration Prototype

Proper language integration of HTM which needs *explicit* marking for transactional accesses requires integration into a compiler to catch all accesses to (shared) objects and properly instrument them. Manually identifying all necessary accesses by the programmer and wrapping them in functions is tedious and error prone. The concept of *atomic regions* allows marking larger blocks of code and all called functions, and the compiler will instrument all memory accesses to shared memory. In our joint publication [213], we have integrated the transactional compiler stack developed at TU Dresden using LLVM with a separate pass to identify memory accesses and turn them into inlined calls to the transactional memory barriers (accessor functions). These are similar to STM barriers (functions that perform fine-grained locking according to the STM mechanism) but largely consist only of a single `LOCK MOV` instruction. The link-time optimisation and inlining of these functions performed in LLVM thus is crucial for performance.

The DTMC compiler stack allows generic transactions and provides a flexible backend structure to generate code for multiple TM implementations. The support for ASF is one of these backends. As such, the ASF integration benefits from the excellent support for transactional language features, but also has to rely on optimisations to reduce overheads, and still has considerable overheads for transaction entry due to checkpoint creation.

The ASF backend in DTMC uses hand-crafted assembly and additional LLVM code to: (1) create a `setjmp`-like anchor to return to in case of abort and (2) copy all live local variables to a secure location, because the compiler does not instrument writes to these and they might be captured in registers.

For simple examples and fine-grained control (used for example in the linked list benchmarks), the transactional primitives can be integrated without full compiler knowledge through a few inline assembly snippets. Figure 3.4 shows the implemented slim wrappers with small overheads and no additional compiler requirements. Figure 3.5 and 3.6 present a simple application example and the resulting compiled code. The idea of wrapping the transactional start and end code into a for-loop originally appeared in [130], allowing the syntactically clean usage of transactional basic blocks of the following form: `asf_atomic { int x = asf_load32(&foo); asf_store32(&foo, x + 1); }`

[3]Ravi Rajwar, Intel, personal discussion.

```
#define asf_speculate(fail, state_save) \
    asm volatile("mov %%rbp, %1" \
        ASF_SPECULATE \
        :"=a"(fail),"=m"(state_save))

#define asf_speculate_fail_clobber( fail , state_save) \
    asm volatile("mov %2, %0 \n\t" \
        "mov %1, %%rbp" \
        : "=a"(fail)  \
        :"m"(state_save), "r"(fail)
        :"memory","rbx","rcx","rdx","rsi", "rdi",
        \ "r8", "r9","r10","r11","r12","r13", "r14","r15" )

static inline long asf_commit() {
    long res;
    asm volatile(ASF_COMMIT :"=a"(res) : :"memory");
    return res;
}

static inline uint32_t asf_lock_load32(const volatile uin32_t* m) {
    uint32_t t;
    asm (ASF_LOCK "movl %1,%0"
         :"=r"(t):"m"(*m));
    return t;
}
static inline void asf_lock_store32(volatile uint32_t* m, uint32_t d) {
    asm (ASF_LOCK "movl %0,%1"
         :: "r"(d),"m"(*m));
}

static inline int enter_asf() {
    int fail ;
    unsigned long clobber;
retry:
    asf_speculate( fail , clobber);
    if unlikely ( fail ) {
        asf_speculate_fail_clobber ( fail , clobber);
        goto retry;
    }
    return 0;
}
static inline void leave_asf() {
    asf_commit();
}

#define asf_atomic \
    for (int ___asf___ = enter_asf();\
        !___asf___ ;\
        ___asf___=1, leave_asf())
```

Figure 3.4: Example code for prototyping ASF wrappers without full compiler support.

```
int DCAS(uint32_t *m1, uint32_t *m2,
    uint32_t cmp1, uint32_t cmp2,
    uint32_t new1, uint32_t new2)
{
    int eq = 0;
    asf_atomic {
        uint32_t d1, d2;
        d1 = asf_lock_load32(m1);
        d2 = asf_lock_load32(m2);
        if ((d1 == cmp1) && (d2 == cmp2)) {
            eq = 1;
            asf_lock_store32(m1, new1);
            asf_lock_store32(m2, new2);
        }
    }
    return eq;
}
```

Figure 3.5: Using the transactional prototyping extensions from Figure 3.4 with a small application example of double compare-and-swap.

```
  mov      %rdi, %rbp
  push     %rbx
  mov      %rsi,−0x28(%rsp)
  mov      %edx,−0x14(%rsp)
  mov      %ecx,−0x18(%rsp)
  mov      %r8d,−0x20(%rsp)
  mov      %r9d,−0x1c(%rsp)
  mov      %rbp,−0x8(%rsp)
  speculate
  test     %eax, %eax
  jne      abort_handler
entry:
  lock mov 0x0(%rbp),%edx
  mov      −0x28(%rsp),%rax
  lock mov (%rax),%eax
  cmp      %eax,−0x18(%rsp)
  jne      wrong_compare
  cmp      %edx,−0x14(%rsp)
  jne      wrong_compare
  mov      −0x20(%rsp),%eax
  lock mov %eax,0x0(%rbp)
  mov      −0x28(%rsp),%rax
  mov      −0x1c(%rsp),%ebx
  lock mov %ebx,(%rax)
  mov      $0x1,%edx
done:
  commit
wrong_compare:
  xor      %edx, %edx
  jmp      done
  ...
abort_handler:
  mov      %eax, %eax
  mov      −0x8(%rsp),%rbp
  mov      %rbp,−0x8(%rsp)
  speculate
  test     %eax, %eax
  je       entry
  jmp      abort_handler
```

Figure 3.6: Compiler output of the DCAS example from Figure 3.5 in AMD64 assembly code; compiled with GCC 4.8 and optimisation level -O3. The transaction code folds the recovery from register modifications for aborts into the fast-path code. That reduces overall code size at a small overhead in execution time. Note how the compiler also restores registers / variables that were not written to in the transaction.

```
50   mov      %rdi,%rbp
51   push     %rbx
52   mov      %rsi,-0x30(%rsp)
53   mov      %edx,-0x28(%rsp)
54   mov      %ecx,-0x24(%rsp)
55   mov      %r8d,-0x20(%rsp)
56   mov      %r9d,-0x1c(%rsp)
57   mov      %rbp,-0x18(%rsp)
58   speculate
59   test     %eax, %eax
60   jne      abort_handler
61   lock mov (%rdi),%eax
62   lock mov (%rsi),%edx
63   cmp      %edx,-0x24(%rsp)
64   jne      wrong_compare
65   cmp      %eax,-0x28(%rsp)
66   jne      wrong_compare
67   mov      -0x20(%rsp),%eax
68   lock mov %eax,(%rdi)
69   mov      -0x1c(%rsp),%eax
70   lock mov %eax,(%rsi)
71   mov      $0x1,%edx
72 done:
73   commit
74 abort_handler:
75   mov      %eax,%eax
76   mov      -0x18(%rsp),%rbp
77   mov      %rbp,-0x18(%rsp)
78   speculate
79   test     %eax, %eax
80   jne      abort_handler
81   lock mov 0x0(%rbp),%eax
82   mov      -0x30(%rsp),%rcx
83   lock mov (%rcx),%edx
84   cmp      %edx,-0x24(%rsp)
85   jne      wrong_compare
86   cmp      %eax,-0x28(%rsp)
87   jne      wrong_compare
88   mov      -0x20(%rsp),%eax
89   lock mov %eax,0x0(%rbp)
90   mov      -0x1c(%rsp),%eax
91   lock mov %eax,(%rcx)
92   mov      $0x1,%edx
93   jmp      done
94 wrong_compare:
95   xor      %edx,%edx
96   jmp      done
```

Figure 3.7: Compiler output of the *unrolled* DCAS example from Figure 3.5 in AMD64 assembly code; compiled with GCC 4.8 and optimisation level `-O3`. In this example, the compiler generates a separate fast-path of the transaction content that has no overhead for superflous register restoration.

The compiler may generate the resulting code in various ways, depending on optimisation targets. The two examples in Figures 3.6 and 3.7 show two possible outcomes. The second example has the nice property that there is no restore clobber section in the uncontended case. That means that while the compiler needs to save the live registers to handle a potential abort (lines 1 – 8, 50 – 57), the compiler does not have to *restore* the registers in the fast-path code, as is demonstrated in Figure 3.7, lines 58 – 73, where the live registers are used as is. The cost is a code size increase, as the restore code or a second version of the entire transaction need to be added (lines 78 – 93).

In Figure 3.6, the compiler was clearly optimising for code size and assumed that multiple retries would be the common case and collapsed the first fast path and the retry path into a single code path that restores / uses the live registers from the saved location on the stack (lines 13, 14, 16, 18, 20, 22, and 23). However, thanks to AMD64's flexible operand specification modes (can directly operate on memory operands instead of needing a separate load / store), the restore can be interleaved with the actual logic.

In general, the code is very lean and does not induce additional function calls, and also saves / restores only registers which are actually live. There are limitations with this approach: despite testing with several transactional payloads and compiler versions, the code is not suitable for production systems, as the compiler barriers are hand-tuned to deal with saving / restoring `rBP`, the frame pointer, which can not be clobbered in an inline assembly clobber list (lines 8, 34, 35). Manual inspection (and many simulated cycles) of all tests, however, confirms that the compiler properly handles the save and restore. Testing with multiple versions of GCC (4.6 - 4.8) shows this to be robust accross compiler versions.

Furthermore, the compiler needs to restore all live registers, even those that are *never written* to in the transaction (`rDI`, `rSI`) because the register clobber statement conservativley discards *all* register state at transaction abort. A proper compiler integration that is aware of all writes (and the register allocation) inside the transaction could identify those live variables (registers) that are never changed in the transaction and remove the need for saving / restoring them. This could reduce overhead in particular of small transactions, but the required compiler changes are outside of the scopeof this thesis. Similarly, the generated code is not very elegant, the explicit propagation of the abort code is required to ensure that register `rAX` is indeed used for the propagation (line 33, 75). Unfortunately, the inline assembler code has no way of efficiently tying into the control flow, or export the flags register. Therefore, an additional `cmp` or `test` instruction is required to check for an abort condition (lines 10, 37, 59, 79), even though the `SPECULATE` instruction already sets the flags properly allowing a conditional branch to follow directly.

Full compiler support could improve the generation of faster small transactions by steering the optimisation towards a non-restore from clobbered uncontended case, as has been done manually, here.

3.4 Incremental Adaptations of ASF

So far, this chapter presented the basic ISA design decisions for ASF, and a small prototypical C language wrapper that enables small transactions from C. I will now describe two incremental changes that we made to the design of ASF; I will describe larger repurposing changes in Chapter 6.

3.4.1 Inverted Transactional Semantics

ASF has been designed from the ground up to be an addition to the AMD64 ISA. Therefore, it does not change the meaning of normal loads and stores, but instead prefixes instructions that should behave specially with the `LOCK` prefix. The original idea was that this would simplify the instruction decoder and followed from the concept of the ASF 1 declarators, because there is no context information required to understand if a specific memory instruction is transactional or not. Instead, additional work that is required for the transactional accesses can easily be added in the decoder front-end because instructions are directly flagged.

Marking transactional accesses relies heavily on a toolchain that detects and converts shared memory accesses inside language-level transactions into the right `LOCK`ed machine instruction. Since the original use-cases for ASF were small, contained transactions, such as DCAS, the working set was available in advance and / or memory accesses were easily marked by the programmer. Early discussions with AMD's micro-architects suggested that the marking of transactional instructions would be beneficial for a hardware implementation. As outlined above, the idea is that early detection of transactional accesses in the decoder would remove the need to track a transactional context to decide whether a load / store was transactional or not.

Despite explicit marking, ASF needs some context awareness (is a specific memory access inside / outside of a transaction) in the front-end of the CPU (see Chapter 4), in order to properly track whether transactional `LOCK`ed instructions are placed inside an ASF transaction (they constitute an illegal instruction if they are not). Naturally, we looked to exploit the transactional context tracking in hardware, and into benefits of changing the "polarity" of the `LOCK` prefix in ASF: unmarked, basic memory instructions would exhibit transactional semantics, while marked instructions would be *non-transactional* inside a transaction. These new *inverted* transactions are started with the `SPECULATE_INV` instruction, while the existing mode is still available through `SPECULATE`.

This relatively simple change creates *significant* simplifications for the programming model, and is therefore the standard mode of operation of all commercially released HTM architectures [353, 367]. Notably, they were all released *after* our initial ASF design had been published. Most importantly, it is now possible to use ASF for eliding locks of critical sections and call into binary library code from transactions, while keeping the critical sections themselves in their original binary form. This is possible, because all memory modifications made by the called code will be undone on transaction abort, and unsupported instructions (that could cause unrecoverable state modifications, such as system calls) would still abort the transaction. The partial execution of such binary code therefore cannot leave any inconsistent state behind.

Together with `LD_PRELOAD` tricks that replace the pthread locking library with one that performs ASF transactions instead of acquiring / releasing pthread mutexes or locks, we have shown to support transparent transactional lock elision with normal transactional memory [254]. The paper is attached in Appendix B.4.

With the described simple compiler wrappers (Section 3.3), the inverted mode integrates ASF almost fully into the compiler, thanks to the inverted semantics and no need to mark accesses at all. One problem of the non-inverted ASF was that stack accesses (as being unmarked) would occur in a non-transactional fashion. It is therefore critical to not overwrite pre-transactional call stack (for example by starting the transaction and then returning to the transactional code) in normal ASF. Inverted ASF, however, treats all unmarked memory accesses as transactional and thus call-stack overwrites become speculative and will be undone upon transaction abort; at the expense of always adding stack accesses to the transactional working set.

One problem arises with mixing normal and inverted mode transactions: upon a commit, the outer transaction needs to know how to treat the polarity of the lock prefix. This cannot, however, be solved with a `COMMIT_INV` instruction, as that would need knowledge of the outer transaction type and not the one just being committed. One possible option is to keep a bit-stack in the CPU and memorise the normal / inverted state whenever a new, nested transaction is started. For simplicity, however, we restrict nesting to same-type nesting initially (nesting only inverted transactions in inverted transactions, and non-inverted in non-inverted transactions).

3.4.2 Signalling Transactional Problems

The original ASF 2 specification [186] (Appendix A) was very strict on flagging transactional errors that indicated a programmer problem. Executing wrong instructions, improper mixing of transactional and non-transactional accesses (see Section 3.2.6) and other issues would not only cause an abort of the transaction, but also cause a general protection (`#GP`) or undefined instruction (`#UD`) exception. The reasoning behind the strong flagging of these errors was the assumption that transactions would be written for ASF by a programmer and in accordance with ASF rules for illegal instructions and memory accesses. A large use case for transactional memory, however, is that of replacing critical sections with transactions in existing code–often without recompiling / extensive restructuring of the code in the transaction / critical section.

After practical experience with a full runtime and transactional compiler (using the TU Dresden stack [213], we realised that the generation of exceptions in those cases was rather draconian. In particular when transactions may call into the OS to allocate memory, a `#GP` exception is a drastic measure, as these will usually kill the application straight away. Instead, these issues can be worked around by following the backup path, which for example uses software transactional memory or a single global lock.

Installing a signal handler for these cases seemed a too heavy-weight solution and would require careful distinction between in-transaction exceptions caused by the transactional context around legal code that could be fixed by simple restart and possibly using a different synchronisation mechanism, and those that are genuine programming errors (for example calling a kernel mode instruction in user mode).

We therefore changed the specification and implementation of ASF such that these events would be notified to the transaction by aborting and setting a special bit in the abort code and not generate an exception separately.

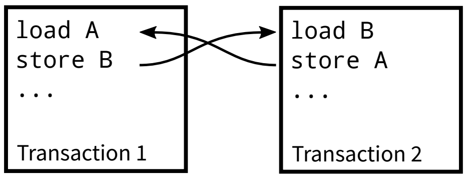


Figure 3.8: Two transactions with overlapping, conflicting access patterns.

3.5 Capacity and Architectural Progress

Transactions may abort in response to various events: contention on shared data (Section 3.2.3), resource depletion, unsupported instructions, interrupts and exceptions (Section 3.2.9). If such aborts occur repeatedly, transactional progress may be reduced to nil. From a programmer perspective, strong properties (guarantees) about the progress of transactions of varying size, length and composition are desirable, because they may reduce the need for complex fall-back paths. Fall-back paths can reduce overall performance since they may be overly serialising (global lock fall-back), be hard to verify due to subtle interplay between different synchronisation mechanisms, and generally contain more bugs due to little utilisation. Furthermore, strong progress guarantees in the face of contention can directly improve performance by limiting the amount of wasted execution in aborted transactions and reduce software contention management complexity and runtime overheads.

From the hardware, the opposite is true. Making any assertions about the transactional properties ties very closely into the underlying freedom to change the microarchitecture of the CPU. Instead, transactions should be allowed to abort for *any* unforeseen interaction that makes it infeasible to continue transactional execution. As an example, standard (non-real-time[4]) CPUs do not guarantee a certain size or replacement policy for their caches and therefore, no guarantee is given for the state or location of a specific data item in the cache. Instead, CPUs significantly rely on the beneficial effects of caches but continue to execute instructions if caches are too small (albeit at lower throughput).

In contrast, mechanisms for buffering / tracking transactional data often have limited sizes and thus cannot support transactions of arbitrary size. No forward progress will be made if an application repeatedly executes a transaction that is too large. Specifying a size guarantees for transactions therefore would unduly restrict the freedom of future microarchitectures that for example wish to employ a smaller, but faster cache.

In addition, certain progress guarantees require potentially complex system-wide reasoning, for example guaranteeing progress for multiple conflicting transactions. Figure 3.8 shows two small transactions. Two transactions that abort each other after one retries need some form of centralised contention control to give strong progress properties. On top of that, the order of memory accesses that may be crucial to determine conflict circles may change due to the processor employing out-of-order execution and prefetching (see Chapter 4).

In order to not impede progress and to allow simple, localised reasoning, ASF employs a simple requester-wins conflict resolution policy and does not make any assertions for contending transactions. Expectations are that non-progressing pathological cases are rare and should be handled with a combination of (randomised, exponential) back-off in software, and a progress guaranteeing fall-back path. For lock elision, ASF's worst case progress properties and performance depend on the algorithm when to switch to grabbing the global lock.

ASF does not fully refrain from giving progress guarantees; it give a limited guarantee which is close to *obstruction freedom* [67]: if transactions have a working set smaller or equal to the architectural

[4]Hard real-time CPUs such as ARM Cortex R-series CPUs with scratchpads are a notable exception.

minimal capacity, do not cause exceptions, and do not face contention or interrupts, they will *eventually* succeed. The architectural minimal capacity is four 64 byte cache lines, but specific implementations may specify (for example through a CPUID-field) that they have a larger minimal capacity that gives the same obstruction-free progress property.

This progress property does not *require* that all application transactions have to meet these constraints; instead, larger transactions are executed on a best-effort basis and are expected to succeed often (but no explicit progress guarantee is given).

Obstruction-freedom under capacity limits is a weaker property than unbounded obstruction freedom, lock freedom, and wait freedom (bounded / unbounded, but in the face of contention). It is, however, a stronger assertion than plain best-effort mode: pure best-effort implementations are free to not actually implement useful transactional memory, but may instead choose to abort *every* transaction. ASF is not at liberty to do so.

The benefits for the programmer / software layer of obstruction-freedom have been explained earlier, also the complications of providing progress guarantees involving contention. The capacity constraint is derived to the worst-case size of the data structures used for transactional conflict detection and versioning. While a typical level-one data cache has a larger capacity than the minimal guarantee (several tens of kB vs.4×64 bytes), unfortunate address patterns may cause set collisions: all cache content is mapped to a single set and the cache capacity is effectively the cache associativity. Further reasons that complicate specifying the associativity of the cache as the worst case capacity are described in more detail in Chapter 4.

Even though four cache-lines may not cover big transactions, important data structure operations, such DCAS, linked list and tree modification can be covered by atomically modifying and monitoring two to four memory locations. It is these use cases that benefit most from the described guarantee.

3.6 Summary

This chapter introduced the "Advanced Synchronization Facility" (ASF) which is a best-effort HTM and was the first published industrial HTM ISA extension. Several design decisions in ASF 2 were influenced by experiences and choices of the preceding design of ASF 1 (that I have worked on for my Diplom thesis). Several points distinguish ASF from other similar BeHTM proposals: the treatment of exceptions inside transactions, the ability for the programmer to bypass the transactional mechanisms and specify non-transactional memory accesses, the lack of a full register checkpoint at transaction entry, and the specification of a limited guarantee for how transactions must be executed.

In retrospect, several decisions have been revisited in more recent commercial implementations:

- full register checkpointing simplifies compiler integration and reduces software overheads (but also increases demands on hardware); some proposals provide programmer control over which registers will be checkpointed [270]
- non-transactional accesses are only available in suspend / resume mode in IBM Power with similar, unclear semantics of sharing data between transaction and suspended region; and the additional complication of possible aborts of the "underlying" transaction [287]
- IBM z-Series gives stronger progress guarantees for stricter restrictions on transaction structure and size that our proposal [270]; all other proposals are strictly best-effort

Chapter 4

Microarchitectural Implementation Details of HTM

This chapter presents challenges and solutions for implementations of BeHTM in modern out-of-order microprocessors. ASF, introduced in the previous chapter, is not only used to illustrate typical critical elements of any BeHTM implementation, but I will also present the implementation details and cost of the features specific to ASF and not present in other BeHTMs. This chapter contains material that originally appeared at the TRANSACT workshop [220], MICRO conference [214] and at a keynote at the VELOX project meeting in Champery [237]. The material has been extended, rearranged and updated.

In addition to the added material here, the largest change is the added implementation of ASF in the Marss86 simulator [253]. This simulator is an offspring of the original simulator baseline of PTL-sim [135], but adds a much more detailed memory subsystem with enhanced modelling of directories, buses and coherency messages [208]. In Chapter 5, I will discuss detailed trade-offs, challenges and solutions from the simulator implementation perspective.

Viewed abstractly, Transactional Memory is a simple programming construct and should be easy to implement, as well [34]. Taking a more detailed look at modern processor pipeline implementations, however, many real-world challenges and important *corner cases* are lurking in shadows of reality. To identify these areas, one must actually go and implement the primitive in a realistic substrate, not just in a simple, in-order simulator (frequently done in related work such as [102, 126]). To that end, I will present a selection of concrete examples of challenges and experiences. While they might not apply down to the last detail to all other microarchitectures, I expect that the areas of impact and general solution strategies can be adopted widely. Similarly, a large part of the details presented will apply for HTM implementations other than ASF.

The remainder of this chapter is organised as follows: terminology is discussed in the next paragraph; Section 4.1 will give a high-level overview of the design and implementation questions, which will be substantiated in extended copies of our TRANSACT 2010 publication [220] in Section 4.4 and our publication at MICRO 2010 [214] in Section 4.5. I will briefly mention other implementation variants that we considered in Section 4.6. In Section 4.7, I will analyse the interactions between the microarchitectural level and programmer visible behaviour, and how the features specific to ASF interact with the microarchitecture.

4.1 Introduction

To implement the HTM mechanism outlined in Chapter 3, several areas in the microarchitecture of a modern processor core need to be changed. For the baseline processor microarchitecture, there is a significant variety in the design space, and a large number of publications exist on the matter; Hennessy and Patterson provide an extensive overview [123].

This work extends modern high-performance CPUs, which find instruction-level parallelism (ILP) in sequential instruction sequences. To achieve this unserialisation, they employ *out-of-order execution* (OoO) [1, 10], *register renaming* and *branch prediction* in addition to *pipelining* to extract instruction-level-parallelism (ILP) from sequential execution streams. Together, these techniques will shuffle independent instructions and only sequentialise true producer-consumer instruction pairs and hoist execution past unresolved branches. The next sections will introduce these mechanism in more detail to understand complexities encountered when implementing BeHTM.

4.1.1 Processor Microarchitecture

Although the exact implementation details of contemporary out-of-order CPUs vary, the generic functionality and structure are very similar (see also Figure 4.1): a *front end* fetches and decodes instructions, the *execution units* perform the operations, and finally the *back end* will remove completed instructions from the core.

Front End The core *fetches* instructions in the native AMD64 instruction-set architecture (ISA) from memory (the instruction cache) and decodes the instructions and operand information. A number of the instructions are not executed directly in the core, but are split up into multiple smaller instructions, so called microoperations (uops), instead. These flow through the pipeline independently and also *retire* in sequence.

Conditional branches make the code sequence dependent on data, which usually is produced only near the branch instruction and may be subject to long-latency operations, such as complex arithmetic and cache misses. To maintain a sufficiently large look-ahead instruction window, modern microprocessors employ *branch prediction* to forecast the instruction stream; decoupling code-flow from data and allowing further look-ahead. If a conditional branch is predicted the wrong way (predicted taken vs. resolved not taken, and vice versa), instructions on the wrong branch have been executed. These instructions have to be removed from the core and their effects have to be undone, or annulled, and architectural state needs to be restored to a previous, known-good configuration.

Other predictions, such as predicting intra-thread data dependencies (or their absence) for pairs of stores and loads with unresolved addresses (store-load aliases), or optimistic assumptions for scheduling conflicts and late resource shortages, may also cause re-execution of instructions.

A central data structure called the *reorder buffer* (ROB) keeps track of in-flight instructions, their states, and required input operands. Dependencies among instructions are formed through producer-consumer relationships between instructions: operands required by one instruction are produced as results by an earlier one and are usually conveyed through registers.

Because architecturally visible registers may be used by multiple independent in-flight instruction pairs, *register renaming* is used to separate these aliases. Register renaming happens early in the pipeline and maps a small set of *architectural registers* to a larger number of entries in a *physical register file*. Mapping happens so that anti-dependencies (write-after-read hazard) and overwrites (write-after-write hazards) are dealt with by mapping writes to different physical registers than the one currently mapped to the architectural register they are overwriting.

Execution Once an instruction has all input dependencies fulfilled, it is considered for execution and is eventually *issued* on one of the functional units of the core. Executing these instructions is not dependent on program order at this point anymore, but can proceed out of order: later instructions with fulfilled dependencies may execute before earlier instructions with unmet dependencies. Once the instructions complete execution, they forward the results to dependent in-flight instructions. The final pipeline step *retires* instructions from the core. In contrast to previous pipeline stages, this stage processes completed in-flight instructions strictly in program order and thus maintains the sequential semantics of the code.

One source for long-latency operations are load instructions that access memory: memory latency has not kept up with CPU clock frequency scaling and so a memory access takes on the order of hundreds of clock cycles. A hierarchy of *caches* exploits locality of access patterns and stores parts of the working set in faster SRAM-memory cells on the CPU die. Generally, caches closer to the CPU core will be smaller, but offer shorter access latencies (two to twenty clock cycles), while larger last-level caches on die aim to minimise off-chip traffic and offer several megabytes of capacity at higher access latency (forty cycles and more). At the closest level to the core (L1), the caches distinguish between instructions (L1i cache) and data (L1d cache, DC), due to different access patterns and spatial layout. Despite the cache hierarchy, memory accesses may miss in the data cache(s). OoO execution helps because the core can issue multiple independent cache-missing loads at once and execute independent other instructions (such as arithmetic), thereby effectively overlapping the latencies for the loads and the computation. Several data structures keep track of in-flight memory operations: The *load and store queue(s)* (LSQ) of the core handle single load and store instructions before they retire. An additional miss buffer keeps track of the pending cache-lines, which may be referenced by multiple in-flight memory operations.

Executing memory instructions out of order interferes with the global order of memory accesses in multiprocessor systems, impacting memory consistency guarantees. To free the application programmer from reasoning over the actual complex interactions, our baseline core maintains stronger (simpler to reason about) guarantees by locally checking for consistency violations and selectively replaying memory instructions [27]. Other commercial implementations take varying positions on whether to employ these and provide strong [209, 367] memory model with stronger ISA-level rules, or a weak memory model [351, 353] for performance and / or energy reasons.

Back End In the back end of the core, instructions that executed out of order are serialised again; mainly by *retiring* (or *committing*) them from the ROB in order when they have completed. At this point, the architectural state of the register file is reconstructed (either by directly updating a separate architectural register file, or a separate remapping table). Futhermore, completed instructions are checked for exceptions (such as permission errors or missing mappings for memory instructions, and arithmetic exceptions); these require a clean architectural state and must not be actioned while they might be happening on a mis-speculated branch. Because the back end retires instructions in-order, branches must be fully resolved before subsequent instructions can be retired. That way, checking for exceptions in the back end is an easy way to ensure they are indeed part of the actual application instruction stream.

Similarly, other instructions that need to wait for preceeding instructions to finish (sometimes called pipeline serialising) will wait until the retire stage to take effect; fences, `CPUID`, privilege level changes, and modifactions to the segment registers. Another class of instructions that typically waits until the back end are stores. In most ISAs, speculative stores must not be visible to the memory system. Sending them out only at the retire stage ensures that; often, they are then held in a separate write buffer, or post-retire store queue; in some designs, a part of the unified LSQ is used for that purpose.

Types of Speculation In total, modern out-of-order microprocessors execute instructions *speculatively* and need to have mechanisms to deal with wrong speculation and reset the processor state to a known

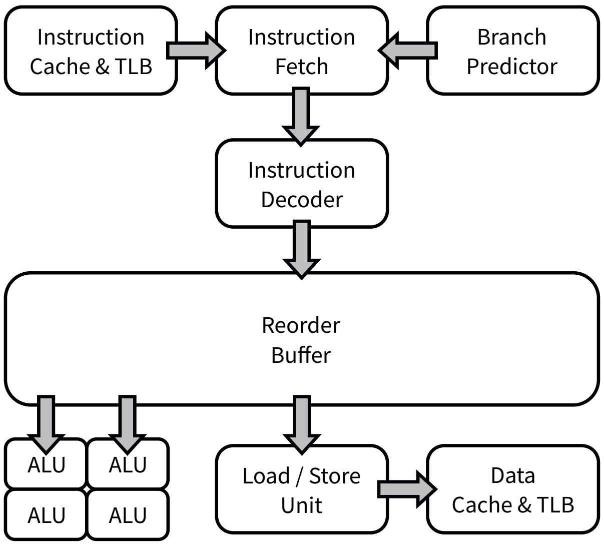


Figure 4.1: Abstract pipeline diagram of an out-of-order microprocessor.

good configuration–a valid architectural state. During speculation, however, speculative state is advanced and only promoted to architectural state once all predictions have been validated successfully (at an in-order retirement stage), or remedied.

In this thesis, I will refer to this collection of speculation as employed by current OoO microprocessors as *out-of-order speculation* (OoO-spec). In contrast, I will refer to speculation used in the transactional context (for example when entering an ASF speculative region) as *ASF speculation* (ASF-spec).

These speculation levels do not have to be distinct, their scope is, however, different. Out-of-order speculation speculates on the instructions inside the reorder- / instruction-window which contains tens to hundreds of instructions[1], and supports a large number of different speculation mechanisms and failure scenarios. Transactional speculation is a more high-level concept, conceptually executing local transactions sequentially, but speculating on (lack of) interference from remote memory accesses. The horizon of transactional speculation is the size of the transactions.

Adding support for transactional memory into an existing microarchitecture needs changes in a large fraction of the units: clearly, the new instructions required to start and end a transaction necessitate changes in the decoder; while the actual transactional memory functionality of conflict detection and versioning require changes to the load-store unit and memory subsystem. The following sections outline the technical details, looking at the features one by one.

4.1.2 Recovery From Out-Of-Order Misspeculation

Discovering instruction level parallelism gets a boost from OoO speculation due to seeing past unresolved (data dependent) branches and scheduling of loads and stores. In many other parts of the microarchitecture, speculation is employed as well to accelerate the fast path of execution. Examples are way-prediction, assuming conflict free scheduling of instructions etc.

If these predictions fail, their effects need to be rectified. In some cases, a simple *pipeline flush* provides the cleanest restoration of a known good architectural state and is a “heavy hammer”: it discards accumulated state and instruction progress of the entire instruction window (tens to hundreds of instructions). Often, selective *annulment* or *squashing* of single instructions is better suited to discard badly speculated paths behind a wrongly predicted branch. The associated tracking logic, however, may make

[1]Even though proposals exist for larger windows, or run-ahead execution [73, 88].

fast-path execution slower and may also slow down the selective repair mechanism. Instruction *replay* will simply retry an instruction on a local pipeline where it was facing resource issues. Finally, *redispatch* will reexecute an instruction as well, but will usually take place after the instruction has already executed and had its wrong result forwarded to consuming instructions. Selectively reexecuting the entire dependency tree of the broken instruction is necessary in those cases. In most designs, redispatch will be approximated by a full pipeline flush.

4.1.3 Coherency Protocol Basics

Coherency protocols are a vast area of both academic and industrial research [15, 22, 31, 33, 256, 279, 297, 313]. The essential function of a coherency protocol is to provide cache coherence in the system, ensuring that writes become visible to other cores in the system and reads always see a coherent view of memory. More formally, coherence ensures that per memory location there is a single global order over writes that all cores observe the same way. In systems with cache coherence, the programmer does not have to manually push data from one cache to another, but instead can rely on the fact that data will eventually become visible at the consumer end.

The coherency protocol works by sending out messages to other cores / caches in the system on reads and writes. These are often called snoop (or probe) messages due the heritage of having a single shared bus that connected all caches and cores and the caches could just snoop that bus. In todays complex memory hierarchies, there is not a single shared bus that can easily be snooped; instead explicit messages need to be sent.

For the sake of this thesis, we assume a simple coherency protocol that sends out snoop messages on cache misses to all holders of a cache line. Reads that miss in the cache will send out *non-invalidating* snoop messages, while writes that miss in the cache (either the cache line is not present or not in an exclusive state) will send out *invalidating* messages.

Holders of a cache line will usually acknowledge non-invalidating snoops (potentially forwarding the data directly to the requester, and noting the lost exclusivity), and upon receiving an invalidating snoop message, all caches will invalidate their copy of the cache line so that the requesting core / cache will subsequently be the sole owner of the cache line and can modify it accordingly.

Again, for simplicity, we assume that the core will send out snoops to *all* (potential) locations before completing the memory instruction. For loads this means they will be sent before making the loaded data available to consuming instructions and retiring the load. Stores we assume will send out invalidating snoop requests before they actually modify the cache line at the point of coherence, or before they commit.

4.2 Key Microarchitectural HTM Mechanisms

Transactional memory is a composition of four main mechanisms working together: data versioning, conflict detection, multi-word atomic stores, and additional transactional state rollback. The following subsections provide insight into their functionality and interaction with existing processor features.

4.2.1 Data Versioning

Transaction semantics require that transactional stores are made visible to other cores only when the enclosing transaction commits successfully. In the case that the transaction does not commit successfully, the transactional stores need to be discarded; that is, locations stored to need to return to the value they had before the transaction. For the duration of the transaction it is therefore necessary that both the pre-transactional version and the speculative, transactional version of the memory location need to exist in the system. The speculative copy will become the authoritative copy upon commit, while the pre-transactional version is the fall-back in case of a transaction abort.

Data versioning describes this existence of two versions of the same memory location and usually *buffering* is employed to keep one of the two possible versions of the data available in addition to the other one which is usually stored at the normal place in the memory hierarchy.

The first decision is whether to reuse / augment an existing, similar structure (LSQ, DC) for buffering, or whether to create a dedicated, new buffering structure. Reusing an existing structure for data versioning usually restricts flexibility in sizing and organisation, may complicate existing interfaces, and slow-down timing on critical paths. Furthermore, the extended unit may need a complete re-validation pass during the design of the CPU, even though transactions are not used. On the positive side, however, may be the smaller overall cost in silicon real estate due to reuse of similar logic and storage cells; and a higher performance due to closer proximity of the backup storage to the unit storing the definite copy of the data / performing the store.

Creating a new buffering structure, on the other hand, may permit better tuning to the needs of transactional code and leave exiting interfaces and control paths unencumbered. On the negative side, the new buffer does need to interact with the existing components and thus needs careful integration into existing flows, and verification effort to ensure that non-transactional operation remains correct. An additional hurdle may be challenges in placing the new structure in an existing, tight floor plan without impeding timing and routing of signals. A dedicated structure will therefore be usually smaller and may have higher latencies to access.

Regardless of the choice, the data versioning mechanism must be able to supply the updated transactional values to loads inside the ongoing transaction, while hiding them from global visibility and allowing to revert to pre-transactional values. Reusing a structure that is queried on local loads (LSQ, DC) has therefore the advantage of automatically forwarding transactional writes to loads inside the transaction. Using a new structure to store the transactionally written data, on the other hand, requires additional checking of upon loads inside the transaction. Together with the idea of optimising for transaction commit, dedicated structures for data versioning should therefore track the old version of the modified data for restore upon abort.

The concept of *eager / lazy* versioning [102] (and similarly, undo / redo logging) is a concept from STM systems, but does not adequately address the different trade-offs and complexities in hardware systems. In particular, it fails to distinguish between the mechanism of data versioning and the location of global visibility, and the time when store probes are sent to the system. For a better analysis, three components need to be considered: (1) is transactional data stored in the local source for loads (i.e. the local L1 DC, the LSQ); (2) is transactional data stored in (or beyond) the global point of visibility and finally, (3) is transactional data snooped at the time of the write, or at the end of the transaction.

Eager versioning for stores would likely refer to a system that stored transactional data in the coherent and globally visible L1 cache, and keep pre-transactional values either in a separate undo log / buffer, or more easily in lower levels (L2, DRAM) of the memory hierarchy (see below). Lazy versioning, on the other hand, could be implemented in exactly the same way in a system that had the L2 cache as the point of visibility and thus would require transactional stores to be pushed out to the L2 at the end of the transaction. Note that in this lazy versioning scheme the usual cost for transactional loads in STMs does not occur, there is no additional buffer that the transactional loads need to consider to be able to read from earlier transactional stores. Instead, the L1 will already provide such functionality.

One particularly light-weight choice for data versioning uses an outer-level cache (e.g. L2 cache) as a storage for pre-transactional data and holds transactionally modified data in the local L1 cache.

Upon transaction commit, the copy in the L2 cache is made unauthoritative, either by invalidating / updating it, or by making sure that the L1 copy (that often is consulted in parallel with the L2) responds to remote read requests. Transaction aborts will invalidate the transactionally written data in the local L1 cache and later instructions read the pre-transactional state from the L2. Care has to be taken that the outer cache / main memory contains the correct pre-transactional copy. For example, in most modern (MSI, MESI, MOESI) cache coherence protocols [136] the most recent store (non-transactional or from a committed transaction) may not be present in main memory, but instead live only in a single cache or cache hierarchy. A transactional store to such a line will require writing-back the pre-transactional data to the L2 cache. Additionally, exclusive cache hierarchies explicitly forbid duplicate cache lines in different levels of a single hierarchy. In these cases, a transactional store may still need to write back prior to the modification of the transactional copy, temporarily bypassing the exclusive regime.

Chosen Implementation For my thesis, I implemented multiple variants of data versioning. The first uses a special buffer (the *locked line buffer*) that buffers pre-transactional data next to the L1 data cache; the second one uses lower levels of the cache hierarchy for storing pre-transactional data as discussed here; and finally, we also implemented a version that provides additional buffering logic in the LSQ of the core to work around pathological capacity cases where indexed data structures can exhibit very low usable capacity due to index trashing.

4.2.2 Conflict Detection

Transactions need to detect conflicts to ensure that they remain *linearisable*. Conflict detection usually involves monitoring the local transaction's working set and remote memory operations on addresses in this set. In cache coherent multiprocessor systems, the *coherence protocol* is the obvious candidate to exchange information about local and remote memory operations. Transactional conflict detection therefore ties in with the specifics of the protocol. Changing the coherence protocol to cater for transactional memory is challenging, because design and validation of a coherent multi-processor interconnect and associated message protocol are hard. An easier choice is therefore to implement conflict detection local to a core, without changes to the interconnect or its protocol. Fortunately, this is possible and is the route we have taken during our work on ASF and also what I will present in this thesis. Despite the benefits customised coherency protocols may provide [104], the cost of changing the protocol usually significantly exceeds the commercial value of the (not yet established) transactional memory programming paradigm whose value is hard to determine. Therefore, *incremental* BeHTM implementations will reuse the coherency protocol. For a fully local implementation that permits concurrent execution of non-conflicting transactions, the addresses of the transaction's read and write set need to be intersected with the snoop messages arriving from the interconnect. This requires a tracking structure to keep these addresses available during execution of the transaction. The fundamental trade-offs are similar to buffering transactional stores for data versioning, but two things are different: (1) tracking requires only read / written information per address, and (2) tracking the working set for conflict detection does not need to be precise, since it can be seen as a mere performance optimisation. Pessimistically aborting too often (for example on *any* message from the interconnect) does not break the safety of the mechanism, but may thwart progress / liveness guarantees and *performance*!

Chosen Implementation In my work, I have implemented three different options for conflict detection in ASF. The locked-line buffer (LLB) introduced earlier for data versioning can also be used for conflict detection. In that case, only an address and read / written information is stored per-LLB entry. The cost of LLB entries (they need to also function as data versioning containers) does limit the size of the read set due to the limited conflict detection capacity (I will show analyses for 256 and 8 entry LLBs).

Transactions tend to read more data than they write [205], so using a dedicated less costly mechanism for read-set tracking is beneficial. The two other proposed implementations therefore use the L1 data cache with additional state bits for transactional conflict detection. One implementation uses only the cache for conflict detection, while the other employs both LLB and cache for conflict detection. Finally we extended conflict detection to additional structures in the core in Section 4.5.

Air-tight Conflict Detection Regardless of the mechanism used to hold the live addresses for conflict detection, another complication is the timing between the actual detection of a conflict, its propagation to the control overseeing the live-cycle of the transaction, the time of the actual abort, and a potential commit or conflicting access of a concurrent transaction. All ASF implementations maintain conflict freedom from transaction start / addition of each element in the working set to the end of the transaction with `COMMIT`.

For successful commit, the entire transaction must have been conflict-free at least a single point in time before the commit–the *linearisation point*. In addition, at this particular point the stores need to become visible to the system, too.

Due to the complex timing constraints on conflict detection, and the usage of the data versioning mechanism also for conflict detection, it is often the case that multiple hardware blocks are responsible for ensuring overall conflict freedom. One such example is using a dedicated buffer for data versioning *and* conflict detection of transactional stores, while using the larger data cache for conflict detection of loads (as they do not require additional buffering).

Similar to the notes on data versioning regarding the notion of eager / lazy versioning, these attributes do not sufficiently describe the behaviour of a conflict detection mechanism. Instead, the timing of sending out probes, and how incoming, conflicting probes are handled by the core executing a transaction require a more detailed specification and can be independent.

Piggy-backing conflict detection on top of normal coherence messages, for example, requires that loads that miss in the cache need to send out probes eagerly. Stores, however, may wait until the send out invalidation messages until transaction end. Conversely, transactions receiving a conflicting probe may decide to abort as soon as possible, abort at commit-time, or may decide to try to fix-up the conflict instead of aborting altogether.

4.2.3 Multi-Word Atomic Store Visibility

Finally, one issue of transactional memory is often overlooked or (implicitly) conflated in the literature: making all the transactional stores visible to the entire system atomically when the transaction commits. The problem arises because complex memory hierarchies have complicated rules for store visibility; there is usually one *point of global visibility* in the system meaning that a store reaching this point is visible to all other cores[2] in the system. *Multi-copy atomic* stores are either only visible to the local core that executed them, or globally visible to *all* other agents. Such systems are the typical *total store order* (TSO) systems, such as SPARC and x86 / AMD64. Systems do exist that are not multi-copy atomic, the store may become visible to some cores while not visible to others, yet. Such behaviour can be observed in non-TSO systems, such as IBM's Power [353], and in the ARMv7 architecture [351]. Figure 4.2 shows various points of global visibility of stores.

For a transaction to commit its stores atomically, all its stores need to be "revealed" at the global visibility point at the same time. Complications arise if this point is not collocated with the structures used for data versioning and conflict detection. The point of global visibility may, instead, be relatively far away from the core and be situated deeper in the memory hierarchy. The distance can complicate

[2]Ignoring other agents, such as GPU and I/O.

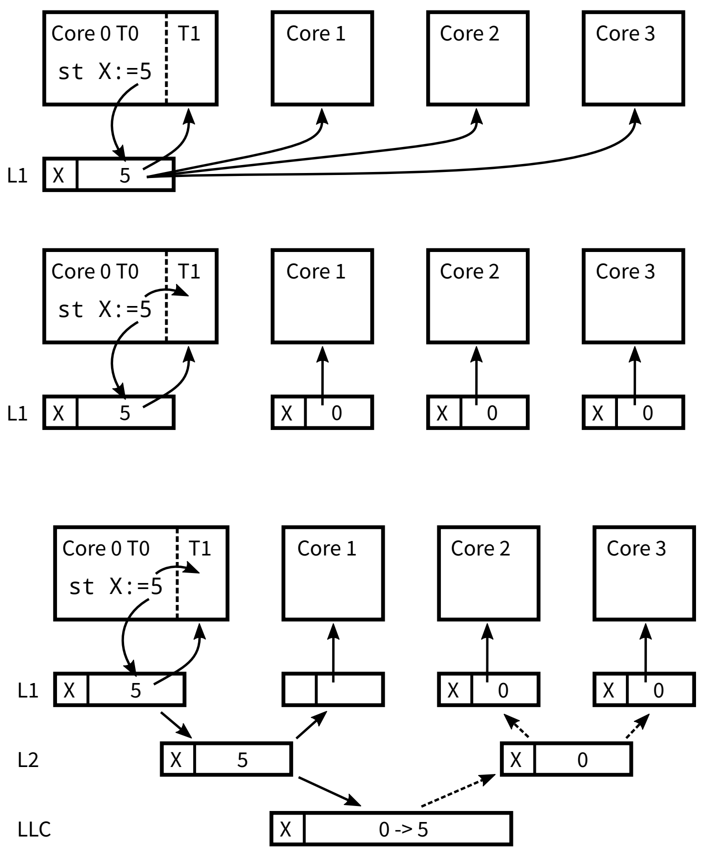


Figure 4.2: Processor cores with multiple options for visibility of stores: close global visibility, distant global visibility and intermediate visibility points. Top: multi-copy atomic system, all agents see the updated value at the same time. Middle: non-multi-copy atomic system, Core 0 Thread 1 can see the update from Thread 0, but other agents still see the old value. Bottom: propagation of updates in a non-multi-copy atomic system, eventually making the update globally visible.

the handshake between the core sending transactional stores and the memory hierarchy down to that point. Extending the conflict detection mechanism to also include the point of global visibility may require consolidated or additional conflict detection logic at the global visibility point, with detrimental effects on latency to and from the core (for transactional marking of entries, reporting conflicts and handling aborts). Not collocating the data versioning unit and the point of global visibility requires an atomic transfer of the transactional stores from the former to the latter at transaction commit. Figure 4.3 pictures the interaction. One example of a transactional memory system with global visibility available only further away from the core is IBM's Blue Gene/Q system with PowerPC A2 cores [281], where the transactional memory functionality resides in the L2 data cache. The resulting trade-offs are analysed in [345].

As established already for data versioning and conflict detection, simple STM-derived eager / lazy reasoning does not capture all freedom in hardware design. In software transactional memory, there is effectively only a single point of visibility per address: that location in the single memory address space. The two (or more) levels of local vs. global visibility are thus not properly captured in the simple eager / lazy characterisation. In addition, the policy for acquiring store permission (see Section 4.1.3, a store sends snoops to the system before it can commit and become globally visible) to a line has multiple steps and options on hardware, these may be done eagerly during transaction execution, while the actual transmission of the values happens at transaction commit.

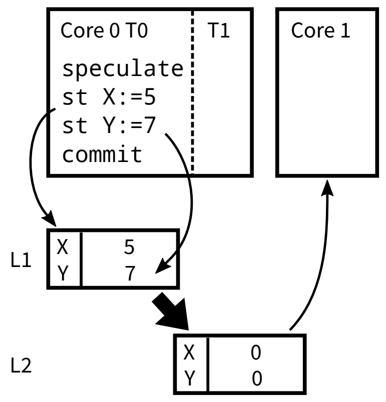


Figure 4.3: Transactional stores must move as a unit from the versioning unit to the point of global visibility.

ASF Design Choices Store visibility varies between the different ASF implementation variants explored in this thesis. All variants eagerly acquire write permissions for the cache line, but some track stores at the point of coherence while others will track stores before that point and thus require a mechanism for transferring stores to the point of coherency atomically after transaction commit.

4.2.4 Rollback of Transactional State

As outlined in Chapter 3, the transactional state that needs to be rolled back may not only comprise the transactional memory modifications. Transactional stores (to memory) are rolled back through data versioning, modifications to the register state are not handled by that mechanism, though. ASF does not require a full register checkpoint and rollback (see Section 3.2.7), instead it only checkpoints the instruction and stack pointer at the beginning of the transaction. Not rolling back all transactional modifications opens interesting use-cases which I will explore in Chapter 6.

If register versioning is desired, various options exist to create a register snapshot / rollback mechanism. The simplest is an explicit register copy and restore mechanism that copies all registers upon transaction entry to a scratch-pad memory / shadow register storage. When the transaction needs to abort, the copies are restored and provide the pre-transactional state. While simple, full snapshot requires time linear in the size of the architectural (integer[3]) register file (16 64 bit registers on AMD64). Depending on the speed of the storage (usually one 64 bit store per cycle to the L1 cache), this may slow down transaction entry (by at least 16 cycles in the example).

In particular for small transactions where overheads matter, only a small subset of the register file is (1) live and (2) modified in the transaction. Tracking liveness is complicated, but register modifications are easily detected in the core. An alternative incremental implementation can log the old register value only on register modifications, adding a total cost only linear in the number of *modified* registers inside a transaction. Care has to be taken that every register is only backed up once, possibly complicating the logic for register overwriting.

Instead of a special memory location, the physical register file itself may hold the checkpointed register values: either copying explicitly inside the register file, or by incrementally backing up with a modified register renaming logic. Figure 4.4 shows such a mechanism. At transaction entry, the architectural register renaming table is checkpointed and the physical registers currently holding architectural state are marked as not to be recycled during the transaction. The net effect of both variants is a reduction of the number of physical registers available for hazard avoidance (ultimately limiting discoverable ILP),

[3]The floating-point and vector registers are usually excluded due to their large footprint.

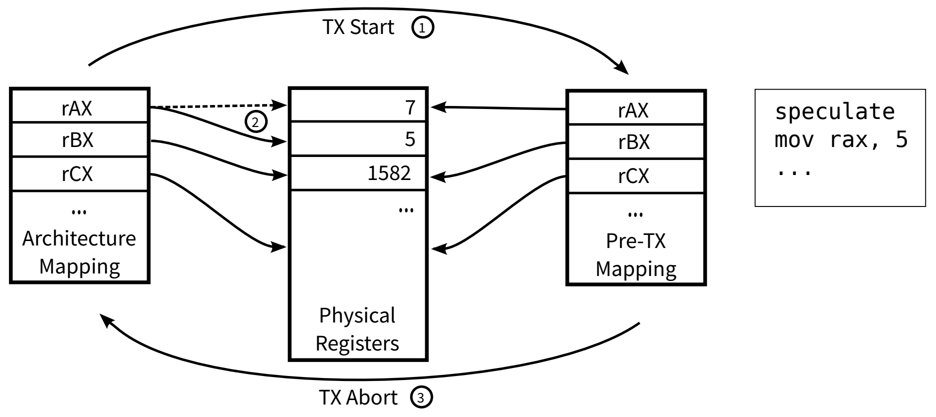


Figure 4.4: Using the register renaming logic to create a backup copy of architectural registers to allow unrolling of register modifications inside a transaction. A starting transaction saves the mapping of architectural registers (1). Updated registers are allocated a new physical register (2), allowing simple undo of the transaction by restoring the pre-tx mapping (3).

but no extra effort during the instruction execution inside the critical section is necessary. The copy of the mapping table itself can be created eagerly, or also only when registers (RAX in the example) are modified. Some microarchitectures provide a similar mechanism to track committed architectural state and OoO-spec architectural state register mappings.

Due to the additional overhead, some HTM proposals, such as ASF, do not undo register modifications inside the transaction. Instead, simple transactions rely on the compiler (or programmer for short assembly snippets) to recreate overwritten, live values (Section 3.3). In addition, several use-cases exist for deliberately not restoring the register snapshot, explored in Chapter 6.

Other options IBM z-Series [259, 270] provides fine-grained control over which registers should be saved and restored on transaction abort and which can be clobbered freely, or even used to transmit information deliberately from the aborted transaction. As a result, that implementation has depending, variable latency for transaction entry due to saving the selected, backed-up registers to scratch memory.

4.3 Basic Implementation Variants

We designed ASF such that a CPU design can implement ASF in various ways. The minimal capacity requirements for an ASF implementation (four transactional cache lines) are deliberately low so existing CPU designs can support simple ASF applications, such as lock-free algorithms or small transactions, with very low additional cost. On the other side of the implementation spectrum, an ASF implementation can support even large transactions efficiently. In this section, we present three basic implementation variants.

4.3.1 Speculation Mechanism Reuse

Given that many cores already have mechanisms for keeping speculative program state private, such as the store queue and an OoO-speculation mechanism, it is tempting to reuse these mechanisms for ASF speculation.

To illustrate, we consider the Rock processor [188], which relies on existing microarchitectural features to implement transactions: Rock uses the hardware's register checkpointing mechanism for keeping and restoring the register-file contents before starting speculation, and it keeps speculative memory updates in the core's store queue. Consequently, Rock needs to abort transactions when the capacity of either hardware resource is exhausted. Furthermore, Rock employs only a single level of speculation; the resolution of branch mispredictions, TLB misses, and other exceptional conditions also abort an ongoing transaction. For these and other reasons, Rock does not give any guarantees on transaction success even in the absence of contention and interrupts.

In contrast, ASF does give an architectural forward-progress guarantee (in the absence of contention) and a minimum-capacity guarantee for speculative regions. Microarchitectural conditions such as a TLB miss or a store-queue overflow must not prevent a speculative region from eventually succeeding. Because it would be impossible to provide these guarantees (including the weaker guarantee of eventual forward progress) based on the OoO microarchitecture, we chose to implement the ASF mechanisms separately from (and complementary to) the OoO mechanisms.

4.3.2 Cache-based Implementation

A first non-core variant is to keep the transactional data in each CPU core's L1 cache and use the regular cache-coherence protocol for monitoring the transactional data set. Each cache line needs two additional bits, a speculative-read and a speculative-write bit, which are used to mark protected cache lines that have been read or written by a speculative region, respectively. These bits are cleared when the speculative region ends. In case the speculative region is aborted, the cache also invalidates all cache lines that have the speculative-write bit set. This implementation has the advantage that potentially the complete L1 cache capacity is at disposal for transactional data. However, the capacity is limited by the cache's associativity. Additionally, an implementation that wants to provide the (associativity-independent) minimum capacity guarantee of four memory lines using the L1 needs to ensure that each cache index can hold at least four cache transactional lines that cannot be evicted by nontransactional data refills.

4.3.3 LLB-based Implementation

An alternative ASF implementation variant is to introduce a new CPU data structure called the locked line buffer (LLB). The LLB holds the addresses of protected memory locations as well as backup copies of speculatively modified memory lines. It snoops remote memory requests, and if an incompatible probe request is received, it aborts the speculative region and writes back the backup copies before the probe is answered. The advantage of an LLB-based implementation is that the cache hierarchy does not have to be modified. Speculatively modified cache lines can even be evicted to another cache level or to main memory. (We assume the LLB can snoop probes independently from the caches and is not affected by cache-line evictions.) Because the LLB is a fully associative structure, it is not bound by the L1 cache's associativity and can guarantee a larger number of protected memory locations. However, since fully associative structures are more costly, the total capacity typically would be much smaller than the L1 size.

4.4 Basic Pipeline and Out-of-Order Core integration

4.4.1 Sequential ASF Semantics

ASF has sequential programming semantics, which a core must preserve whether it employs OoO-speculative execution or not. For example, a transactional memory access occurring inside a speculative region must not be reordered to occur before the beginning of a speculative region. In this section, we discuss two aspects of executing ASF-speculative memory accesses on an OoO core. We begin with issues raised by executing protected memory accesses out of order in Section 4.4.2. Section 4.4.3 discusses how

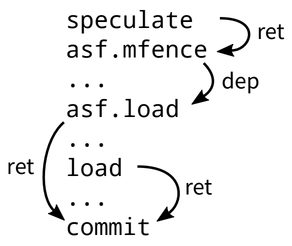


Figure 4.5: Using light-weight fence μops to order ASF instructions at the entry into a transaction.

ASF resources reserved for ASF-speculative data can be managed when the instructions referencing this data are OoO-speculative and later annulled.

4.4.2 Speculative-region Flow

The execution order of instructions in an OoO core is largely determined[4] through data dependencies. Regular instructions without dependencies can generally execute in arbitrary order. For ASF speculative regions, we need to decide whether particular memory accesses are executed inside or outside a speculative region to decide whether it should be handled transactionally (flagged access / implicitly flagged access inside a transaction), constitutes a programming error (flagged transactional access outside a transaction) or is a non-transactional access. Hence, these instructions have to be ordered with respect to the marker instructions that begin / end such a containing region (`SPECULATE` and `COMMIT`). We add special ASF memory-fence μops during decode of the `SPECULATE` instruction to attain this goal, as shown in Figure 4.5. These fences operate mostly like normal fences in that they create an artificial dependency on any later memory instruction that is only resolved once the fence has retired from the core, and do not require special handling / ordering outside the local core (no special messages on the bus). Later memory instructions can therefore only issue after the fence has retired. The fences are ASF-specific in that they only affect ASF-spec memory instructions (`LOCK MOV`s) and not regular memory references. Memory instructions ($asf.memop$) are then ordered by the following ordering rules (with $\rightarrow$ being similar to Lamport's happened-before relation [5] and S(X) denoting the event of instruction X being in pipeline stage S):

$$issue(asf.\texttt{SPECULATE}) \rightarrow retire(asf.\texttt{SPECULATE}) \tag{4.1}$$

$$retire(asf.\texttt{SPECULATE}) \rightarrow retire(asf.mfence) \tag{4.2}$$

$$retire(asf.mfence) \rightarrow issue(asf.memop) \tag{4.3}$$

$$issue(asf.memop) \rightarrow retire(asf.memop) \tag{4.4}$$

$$retire(asf.memop) \rightarrow retire(asf.commit) \tag{4.5}$$

Ensuring that $issue(asf.\texttt{SPECULATE}) \rightarrow issue(asf.memop) \rightarrow retire(asf.memop) \rightarrow retire(asf.commit)$

Rules 4.1 and 4.4 trivially follow from the regular pipeline flow; Rules 4.2 and 4.5 are ensured by retiring all instructions in order. Rule 4.3 is enforced through the functionality of fences: the $asf.memop$ has an artificial register dependency on the (dummy) result that the $asf.mfence1$ instruction produces once it hits the *retire* stage (as opposed to the out-of-order completion of the execute stage for normal instructions). This dependency is created in the ROB and dependency tracking data structures.

In Section 4.5, we present another method that moves the tracking of being inside a transactional context into the core's front-end decode unit.

[4]Additional ordering constraints exist through address overlaps, such as store-to-load forwarding.

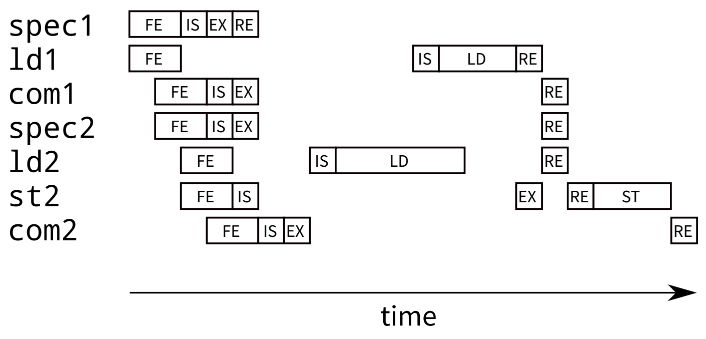


Figure 4.6: Short transactions may execute concurrently in the same processor core due to out-of-order execution and pipelining.

Speculative-region Overlap Short speculative regions that execute neck-to-neck, such as in Figure 4.6, can be in flight in the core simultaneously because of the reorder window.

Keeping track of the state of multiple simultaneous speculative regions is complicated. Conventional state containers–registers–are renamed to track simultaneous usage of shared resources, but making all of ASF's state renameable is complex, because it is large and distributed: not only the information of the speculative region state, the abort instruction and stack pointer, but also the entirety of the bits used to track the read/write sets.

Figure 4.7 shows different options for handling transactions in close proximity in the core.

While there may be safe approximations to full renaming (such as merging read/write sets), we have chosen a more straightforward approach by serializing the execution of consecutive speculative regions in the core. Not renaming close overlapping transactions, but instead merging them will lose the tracking between which transaction had which working set entries forcing group commit / abort. Between the two transactions might be non-transactional code and that must not be visibly executed if the first transaction has not yet committed, and must be executed even though the second transaction might abort. The heavy pipeline-serialization mechanisms, such as flushes and stalls, have a large performance impact because of the time needed to drain and fill the pipeline, decreasing performance especially for frequent, small speculative regions that would execute in few cycles.

To avoid this performance decrease, we chose to implement the serialization through the existing dependency rules, ASF memory barriers, and by not changing the state of the speculative region until `SPECULATE` and `COMMIT` hit the pipeline's retire stage. The serialization of two consecutive speculative regions (comprised of $asf.spec_1$, $asf.memop_1$, $asf.commit_1$ and $asf.spec_2$, $asf.memop_2$, $asf.commit_2$ respectively) is then ensured through the following dependency chain:

$$\begin{aligned} issue(asf.memop_1) \rightarrow retire(asf.memop_1) \rightarrow retire(asf.commit_1) \\ \rightarrow retire(asf.spec_2) \rightarrow retire(asf.mfence_2) \rightarrow issue(asf.memop_2) \end{aligned} \tag{4.6}$$

The worst-case latency due to the additional serialisation is depicted in Figure 4.7. Each mechanism causes additional cycles spent in the second transaction waiting for successful commit of the first transaction. In practice (see Chapter 5), this is a pathological case, because there are usually non-transactional instructions around the transactions which can execute in the gap between the consecutive transactions, thanks to the proposed selective scheme.

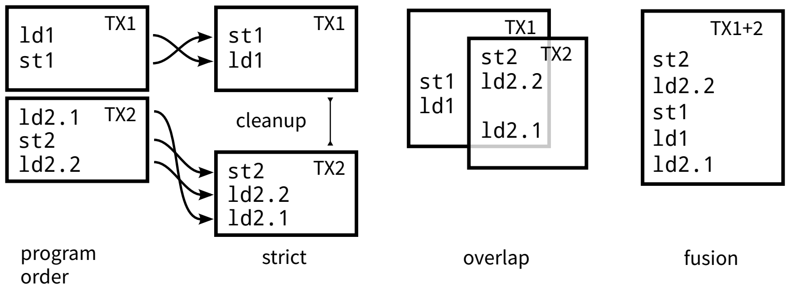


Figure 4.7: Two transactions in close proximity can be executed with various mechanisms experiencing different amounts of overall latency.

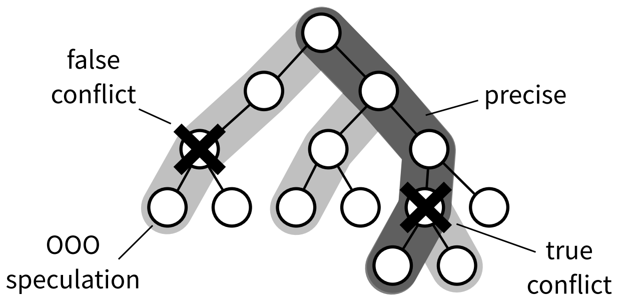


Figure 4.8: Transactional traversal of a linked tree data-structure may increase the working set size due to branch mis-speculation causing traversal of the wrong child pointer. As a result, the conflict surface with respect to concurrent memory accesses may be increased.

4.4.3 Misspeculation

Section 4.1.1 introduced multiple instances for speculation in the OoO core and how they could fail. ASF-speculative load and store instructions are also subject to these mechanisms and this has caused several challenges for our implementation, because of the complex interactions imposed by release and redistribution of resources due to misspeculation.

Precise ASF working-set tracking Because of OoO speculation, the core may overestimate ASF's working set: misspeculated memory instructions can add spurious ASF-spec entries to the LLB or cache before the misspeculation is detected and the corresponding memory instructions are annulled. Linked tree data structures (see Figure 4.8) can exhibit this issue, for example. If the branch for the test at each node is wrongly predicted, speculative execution might traverse the wrong path / multiple wrong paths of the tree.

The overestimation does not impact correctness of the execution conceptually (all lines that need protection are protected), but has performance implications, since the additional lines artificially increase contention and also put additional pressure on the limited capacity.

It is thus desirable to detect and remove spurious entries in ASF's working sets. However, recomputing the actual ASF-spec state of a cache line when annulling an ASF-spec memory access is challenging. The state depends not only on in-flight memory instructions, but also has to take into account retired ASF-spec memory instructions of the current speculative region that have referenced the cache line.

Our LLB-based ASF design supports reference counting for that particular purpose and thus can track read/write sets precisely. Adding reference-counting mechanisms to the existing L1 cache would be expensive; thus, the L1-based ASF implementation currently may *overestimate* the read set. Another option to avoid misspeculated entries in the transactional read/write sets is to mark entries in the cache only after all control flow speculations have been resolved with a mechanism similar to what we describe for misses in the next section; for example at the retire time of the load. The additional cost of an addi-

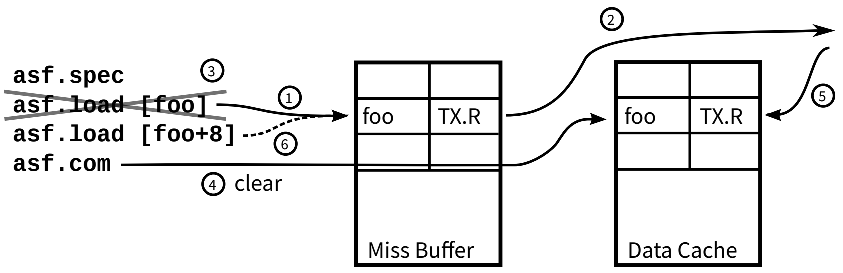


Figure 4.9: Complications when ASF instructions get squashed during a cache miss, and proper identification of the associated cache line is required: a speculative `LOCK MOV` misses in the cache and creates an ASF-marked miss buffer entry (1), and subsequently sends a cache miss request to the memory hierarchy (2). While the request is in flight, the load instruction is squashed (3). If the transaction commits at this point (4) and clears all speculative bits in the cache, the returning response for the miss will then set up the line with ASF marking (5). Simply resetting the ASF flag of either cache line or miss buffer entry upon load squash (3) is not correct, because there might exist an aliasing load that requires the ASF-ness to be preserved (6).

tional access to the data cache may, however, cause additional latency for the instruction, and consume bandwidth of the cache interface.

Orphan cache entries Not tracking ASF's working set precisely can lead to orphan ASF-spec entries in the cache in an L1-based ASF implementation under specific timing conditions; even though it is safe in principle. These orphan entries remain even though the originating speculative region has already successfully committed or aborted; see Figure 4.9.

To illustrate, consider the following sequence of events: an ASF-spec load misses in the cache and sets up an ASF-spec miss-buffer entry to track the cache miss. The load eventually is annulled because it is on a wrongly predicted branch. The cache-miss handling cannot be aborted at this time. Eventually, the speculative region commits by successfully retiring the `COMMIT` instruction (the original dependency on the cache-missing load is not present anymore, since that load has been annulled). The cache line is eventually filled into the cache and gets its spec-read bit enabled because the corresponding miss-buffer entry was tagged as ASF-spec, leading to an orphan spec-read cache line.

Note that simply resetting the cache line's spec-read bit on annulment of referencing ASF-spec loads would be incorrect, because multiple in-flight loads (ASF-spec and non-ASF-spec) may still reference the miss-buffer entry. Similarly, the miss-buffer entry's ASF-spec state cannot be simply reset because it may still be referenced by other in-flight ASF-spec loads.

A simplified version of the recomputation introduced previously solves this issue (Figure 4.10): we reuse the existing reference from a miss-buffer entry to its associated in-flight loads and count the ASF-spec-load references (or rather track the ASF-ness of each referencing memory instruction referencing the missbuffer entry). We observe that no retired load can contribute to the ASF-spec state of the miss-buffer entry because loads can only retire once their cache misses have been resolved. Therefore, the number of ASF-spec loads referencing the miss-buffer entry can always be computed online by counting all non-retired (in-flight) loads with such a reference, allowing miss-buffer entries to precisely track their ASF-spec state and eliminating the need for dedicated reference counting in the L1 cache. In result, no modification to the L1 cache is necessary, and we readily implemented this mechanism to prevent orphan spec-read cache entries in our ASF prototype.

In a simpler design, it may be viable to wait for the missbuffers to be completely empty before start / commit of the transaction, but doing so increases the entry latency and also commit latency in the presence of non-transactional accesses inside the transaction.

Flash clearing all ASF-spec bits (of miss-buffer *and* cache entries) at the end of a speculative region

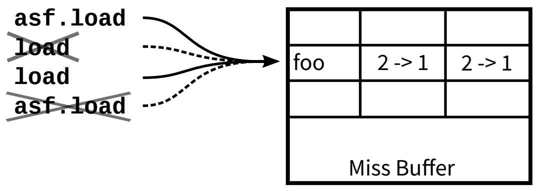


Figure 4.10: Recomputation of the ASF-ness of a specific miss buffer entry that is being reused by multiple in-flight ASF-spec / non-ASF-spec accesses. The mechanism performs reference counting for each of the two categories in the miss-buffer.

(retirement of the `COMMIT` instruction) would also work around the orphan-cache-entries issue. However, our recomputation approach tracks ASF's working set more closely (for misses) and thus reduces the likelihood of contention.

4.4.4 Abort Semantics

ASF has eager conflict detection and provides early-abort semantics: it defines that no side effects (e.g., memory modifications or page faults) ever become visible caused by ASF misspeculation (i.e., further execution of a speculative region after is has been aborted)[5]. The rationale is that no ASF-speculative state should be able to leak unintentionally from an aborted speculative region. In particular with non-transactional stores, timely aborts are crucial, because one may want to reason about the conflict freedom of a speculative region and the non-transactional stores that it managed to make visible up to a specific point.

This section discusses how our ASF implementation realizes early abort semantics. Section 4.4.5 explains that, to receive timely abort information, cores need to track access conflicts with protected data in more CPU data structures than just the cache or LLB because of the asynchronous nature of memory accesses in OoO processors. In Section 4.4.6 we describe how a core recovers when it has received an abort signal.

4.4.5 Conflict Detection Handshake

The global linearizability of ASF transactions and consistency of the read and write sets is ensured through eager conflict detection. Conflict detection has to start when or before the value of the load is bound [27] or the load is performed [17]. Usually, some limited form of conflict detection and additional ordering is already employed in current multiprocessor systems with strong memory semantics[6] to provide suitable memory-consistency semantics to the application. To keep changes to this very sensitive area of microarchitecture small, it is advisable to reuse the existing mechanisms and extend coverage of the conflict observation until the speculative region commits. However, extending the monitoring period of the legacy mechanisms is difficult, because it involves touching sensitive hardware and furthermore may not be possible due to design decisions, such as reliance on bounded delay for certain operations, or serialization of monitoring requests. Finally, the capacity of the legacy mechanisms is usually small, because they only need to support a bounded instruction scheduling window instead of a transaction and increasing their size may carry a prohibitive area, power and timing cost.

Therefore, the responsibility for monitoring ASF-spec data has to transition eventually from the legacy mechanisms (such as the miss buffer or LDQ) to ASF's monitoring facility (such as the LLB or the augmented L1 cache). During the transition, it has to be ensured that the data element is never without

[5]Very related to the "opacity" safety metric for TM discovered later.
[6]Sometimes referred to as "Frey's Rule Snooping" [29].

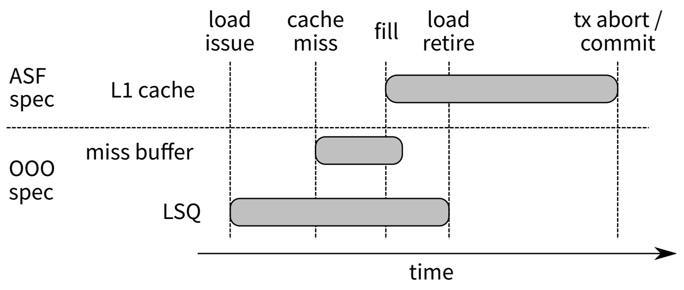


Figure 4.11: Responsibility of devices for tracking conflicts of transactional memory accesses.

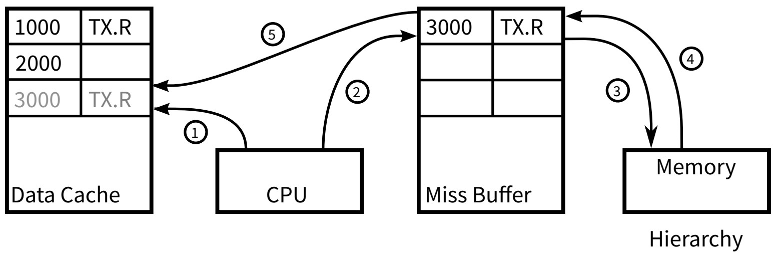


Figure 4.12: Flow of setting ASF bits for loads hitting / missing in the cache.

conflict observance, necessitating atomic transitions or overlapping intervals of conflict-detection responsibility, as depicted in Figure 4.11.

For our prototype, we reuse the existing miss buffers and flag cache lines as soon as they are initially probed (for cache hits) or when they are delivered to L1 (for cache misses) with the according ASF-spec bits. In the case of cache misses, the miss buffer is augmented to know that the cache line it is currently tracking and requesting from the memory hierarchy will need to be flagged with appropriate ASF state. In Figure 4.12, the CPU access the cache for a transactional line (1), if the access is a miss, the miss buffer will be instructed (2) to fetch the cache line from the memory hierarchy (e.g., L2 cache, main memory) (3). Once the line arrives (4), the miss buffer checks and identifies the line as transactional and adds it to the cache with the right transactional bits set (5). For hits, the cache flags the line transactionally at the first access (1), and steps (2) - (5) are not necessary.

Our LLB-based implementation similarly allocates entries as soon as possible, too. This design saves an additional cache lookup at a later point in time (to set the respective ASF-spec bits) and ensures overlapping contention monitoring. The timing between the hand-over from one conflict detection mechanism to another (in particular to the enhanced L1 cache) has been a source of a lot of complexity. For example, one issue we encountered was caused by store-to-load forwarding, in which a load receives the data directly from an earlier store to the same address in the same thread. These loads effectively bypass the caches, circumventing any conflict-detection mechanism implemented in the cache, in particular, when the store itself did not create a conflict detecting entry. This may happen if the store was placed outside the transition, or is a non-transactional store inside the transaction, as can be seen in Figure 4.13.

The transactional begin fence does not really help for two reasons: (1) abstractly, because it only prevents in-TX things to get out, roach motel of older stores applies and the issue of non-tx stores to load forwarding remains; (2) in the implementation, the issue is that PTLsim / Marss86 assume that stores are globally visible as soon as they retire; therefore they do not model such a case and even a full fence would not guarantee that the cache has been queried. This issue was solved by forcing cache access and creating entries in the L1 cache for transactional loads that would get their data through store-to-load-forwarding to ensure proper conflict monitoring.

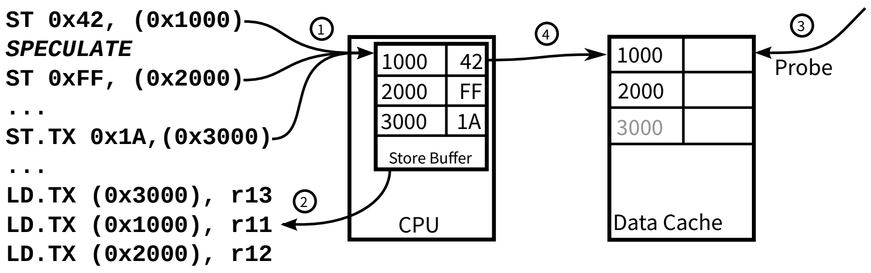


Figure 4.13: Store-to-load forwarding may cause transactional loads to not access the cache providing a challenge for transactional conflict detection. Multiple transactional and non-transactional stores may have an entry in the CPU's store queue or the post-retire store buffer (1). Loads inside a transaction may then directly get their value forwarded from such a store (2) without accessing the data cache. The cache therefore *may not detect* transactional conflicts (3). The data will *eventually* leave the store buffer, but that might be after transaction commit (4).

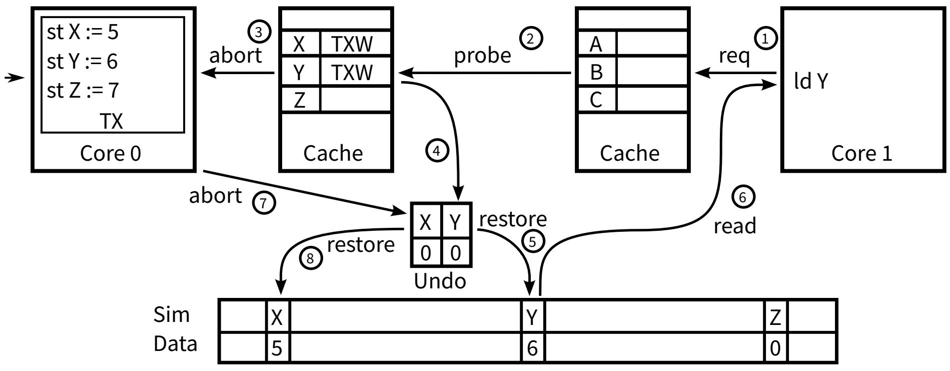


Figure 4.14: Incoming conflicting memory probes abort an ongoing transaction. Their reply, however, does not need to wait for completion of the abort mechanism. Detail: Core 1 sends a load request (1) that misses in the local cache and therefore probes other caches and hits on a transactionally written line (2). This causes an abort signal to be sent to Core 0 (3). The load then checks with the abstract versioning layer (4) which in turn restores the overlapping access to the single global memory view (5), from which the requester reads (6). Eventually, Core 0 acts on the abort (7) and in turn triggers the restore of all its transactionally written locations (8).

4.4.6 Abort Mechanics and Implications

In ASF, speculative regions abort whenever a conflicting concurrent data access is detected (requester-wins conflict resolution policy), which may happen asynchronously to other core timing. As outlined previously, we use the existing cache-coherence mechanisms to detect these conflicting memory accesses. Whenever an ASF-speculatively modified line is read by another core, it must be ensured that the requesting core receives the backup copy with the probe answer, and not the updated data. Chapter 6 will describe techniques around this basic principle, which allow a transaction to continue executing despite the conflict.

The timing between probes, replies, and the rollback operation is crucial for correct operation. To reduce the delay between the arrival of the conflicting probe and the final probe answer, we introduce *partial rollbacks*. These rollbacks undo modifications only for the requested line, deliver the probe answer, and then signal the core for further abort handling. This approach decouples the response to a requester that caused an abort from the actual duration of the abort, reducing risks of deadlocks in the coherency protocol (due to the removed dependency) and allows non-atomic aborts. Figure 4.14 visualises the flow.

Aborting the core (Figure 4.15) can then proceed independently of probe handling, at the core's discretion. The core checks for detected conflicts every cycle. If found, the core triggers the full rollback,

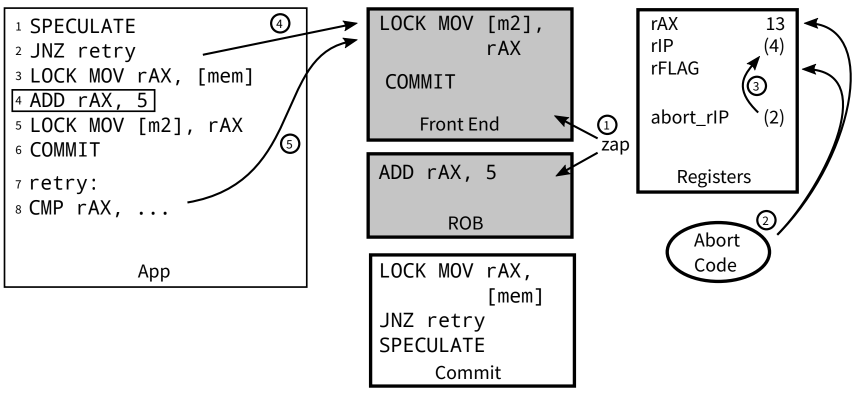


Figure 4.15: The processor core handles transaction aborts like branch mispredictions: by squashing in-flight instructions (1), filling the abort information for the application (2), and resetting execution to a specified anchor (just behind SPECULATE in the case of ASF) (3). Eventually, the core will start fetching and executing instructions from the application's abort handler (4,5). In addition, transactional stores are rolled back through the transactional versioning mechanisms (not shown).

encodes the abort reason into the rAX register, sets the flags register accordingly, and resets the instruction and stack pointer to the values right after transaction entry (SPECULATE, see Chapter 3). Finally, a pipeline flush and reset of the instruction fetcher (similar to the resolution of a mispredicted conditional branch) completes the abort.

Although checking for abort conditions every cycle seems sufficient on the surface, we had to address two subtleties of modern cores, which we describe in the remainder of this section.

Intra-cycle parallelism It is possible for a specifically timed in-flight in-tx store operation to the already rolled-back line to retire in the same cycle in which the abort condition was detected, but before the pipeline flush, essentially proceeding in parallel to the ongoing abort. If the speculative tracking bits of the cache line have been already reset by the abort (due to self-abort / rollback in the cache), and the retirement logic does not reset them, the store would be able to make ASF-speculative modifications permanent (despite the abort of the enclosing speculative region). Proposals with decoupled versioning and conflict detection mechanisms are particularly prone to this and similar errors. Therefore, it is important to avoid disabling write-set tracking too early.

M**op splitting** As described in Section 4.1, native-ISA (AMD64) instructions do not have to proceed atomically through the core. Instead, they may be split up into smaller μops. These μops flow through the pipeline independently and also retire in sequence, which creates another subtlety with respect to the asynchronous nature of aborts: an abort may trigger when only a subset of the μops comprising an instruction have retired and updated the architectural state. Together with non-transactional memory accesses, and only partial register snapshots, the resulting register / memory state after a transaction abort may be inconsistent and not correspond to any execution of single instructions and aborts *between* them.

The most critical instructions regarding this are CALL and RET (in Figure 4.16), because they both access the stack pointer, the instruction pointer and memory non-transactionally. Their partial retirement is, however, contained by ASF, because the abort resets both registers to a consistent value (and no guarantees for stack values below the stack pointer are given). Chapter 6 describes extensions that require consistent register snapshots at abort. We therefore add additional logic that will delay abort handling to between AMD64 / x86 instructions, carefully making sure that the post-abort-retired μops do not expose state (register state or non-transactional memory accesses) after the abort.

```
SPECULATE       COMMIT          CALL <dest>      RET
 asf.spec        asf.com         add tr           ld  tr=[rsp]
 mfence.asf      mfence.asf      st  [rsp]:=tr    add rsp
                                 sub rsp          jmp tr
                                 col flags
                                 bru <dest>
```

Figure 4.16: AMD64 (and other ISAs) may split instructions into simpler μops in the processor pipeline. Aborts that occur when only a fraction of these μops is retired can be challenging, because the instruction may have taken effect, partially.

```
  SPECULATE
  JNZ       abort_handler
traverse:
  LOCK MOV  rdx, [rsi + val]    ; Load val
  CMP       rdx, rdi            ; Element found?
  JE        found
  LOCK MOV  rsi, [rsi + next]   ; Load next pointer
  TEST      rsi, rsi            ; End of  list?
  JNZ       traverse
  COMMIT                        ; Element not found
   ...
found:
  COMMIT                        ; Element found
```

Figure 4.17: A small linked-list traversal loop searching for a particular element, illustrating potential inflation of speculative working set because of mispredicted branches: The loop continuing `JNZ traverse` instruction may be mispredicted and subsequent iterations use up ASF resources needed for maintaining the capacity guarantee of ASF.

4.4.7 Capacity Guarantees

The ASF specification mandates that implementations support a minimal number of read/write set entries (four cache lines), regardless of address layout and other aspects (such as TLB misses, branch misprediction, etc.).

Supporting such a guarantee under the OoO execution regime is complicated by several interactions. As described previously (Section 4.4.3), ASF-speculative memory instructions may flag cache lines as speculative optimistically, and they can artificially increase the speculative region's working set and reduce the number of available cache entries / tracking capacity that an application can really use. In particular, ASF loads behind unresolved and mispredicted branches, such as mispredicted pointer traversal loops (linked-list traversal in Figure 4.17, tree structures in Figure 4.8), can cause this behavior.

Furthermore, loads may be issued out of order and may also fill missing cache lines in arbitrary order, depending on their residence in the underlying memory hierarchy (e.g., line present in L2 cache vs. line fetched from remote main memory). Therefore, even determining precisely if and when the capacity limit is reached is complex, if execution is not to be overly serialised in the common case.

Non-ASF-spec memory instructions may also compete for space in the employed conflict detection device, in particular if an existing structure, such as the L1 cache, is reused for that purpose. It may be possible that non-ASF-spec entries displace ASF-spec entries, thwarting any possible capacity guarantee.

Finally, the organization of the speculative storage and tracking device heavily impacts the feasible minimal guarantee. Set-associative caches have a small worst-case minimal capacity (their associativity) because all requested addresses may alias into the same cache index. Other devices such as Bloom filters [3] may allow tracking of an arbitrary number of elements (with decreasing precision), but do not provide space for backup copies to support ASF-spec stores.

In summary, a naive implementation does not even guarantee the worst-case capacity of the storage container (i.e., the associativity of the L1 cache for a cache-based implementation). Additional ordering and priority mechanisms are necessary to give such a guarantee, for example by carefully ordering accesses to capacity-critical parts of the storage device. However, strictly serializing all memory accesses would reduce overall performance and complicate core design.

For our LLB-based implementation, we have therefore crafted a staged buffer (see Figure 4.18) that has a (small) first stage where cache lines are held as long as they are only referenced by OoO-speculative, in-tx, in-flight memory instructions. Whenever one of these instructions retires, the line in the LLB transitions to the non-OoO-spec second buffer stage. The minimal guarantee is then provided predictably by the non-OoO-spec second buffer stage, while the first OoO-spec stage basically controls how much (OoO-)speculation can go on. This design allows us to carefully trade performance (through higher discoverable ILP) for additional buffer space (for the additional first stage buffer). Memory instructions have to wait until a free entry in the first stage is available before they can issue. To avoid deadlocks through OoO fill-up of the speculative buffer stage, we carefully replay later memory instructions (further down in the program flow) that have already been granted an entry to make room for the earlier ones waiting for a free entry. In the right part of Figure 4.18, LD_4 tries to allocate an entry in the OoO-spec part of the buffer, but cannot do so, because the buffer is filled up with older (ST_3) and younger (LD_5, LD_9) memory accesses (1). In order to avoid deadlocks when allocating into the buffer, if such a condition is detected by the core, either of LD_5 or LD_9, or both, need to replay and free their entry to make room for LD_4 (2).

Transitioning a memory instruction to the non-OoO-spec side requires an additional access from the core to the buffer at the instruction retire stage. As such, the split dedicated buffer is similar to (or the generic concept of) an implementation that uses both the LSQ and the data-cache after for tracking : the LSQ will perform OoO-spec conflict detection / versioning, and the data-cache is accessed (again) after the memory instruction has become non-OoO-speculative (for example at retire). In the next section, I will outline such a design.

Our simple cache-based implementation currently lacks these features, because it aims at reusing most of the existing cache implementation. Hence, it does not yet meet ASF's required minimal capacity guarantee under certain circumstances. In the next chapter, we will look at ways to remedy the situation with cache-based implementations.

4.5 Enhanced Pipeline Integration

In the previous section, a simple baseline processor microarchitecture was used and extended mainly on the memory subsystem facing part of the processor core. In this section, we propose changes that target the core of the instruction flow by modifying the instruction decoder and the reorder buffer, and also extensions that address shortcomings of the purely cache-based implementation of Section 4.4 and focus more closely on optimising the pipeline integration. This section has previously appeared in our MICRO 2010 publication [214] and has been edited and extended.

4.5.1 Overview

While the main design challenges to the ASF implementation come from putting the long-lived ASF speculative hardware context and the short-lived out-of-order execution context together, at a high level, our extended design in Figure 4.19 is close to a cache-based HTM design [74, 102]. In addition to the L1 cache, it buffers speculative data in the *load/store queues*. The L1 cache supports relatively large speculative regions for transactional programming. The load/store queues are used to provide a higher

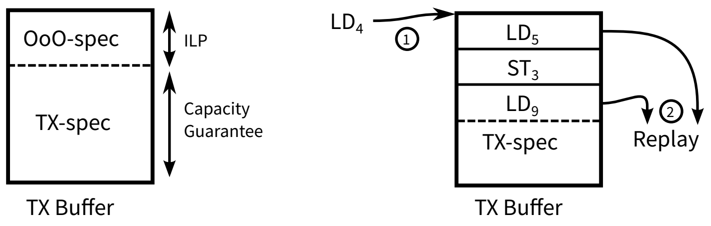


Figure 4.18: Split buffer allowing discrimination between in-flight and retired memory instructions. Left: Logical buffer split with parts for OoO-spec memory accesses that alow ILP, whereas retired transactional accesses are tracked in the TX-spec part of the buffer, which provides the effective (minimal) transactional capacity. Right: Allocation of memory instructions into the OoO-spec buffer part when an earlier LD_4 cannot find space (1), and will displace younger accesses LD_5, and LD_9 (2), but *not* abort the transaction.

minimum capacity guarantee for lock-free programming since the minimum capacity guarantee with a 4-way set-associative L1 cache is limited to only four distinct memory words.

The design adds one bit per load/store queue entry and two bits per cache line to mark speculative data. During a transaction, the AMD Coherent HyperTransport (cHT) protocol [122] is used to detect conflicting speculative/non-speculative memory accesses from other cores by checking incoming cache coherence messages against the additional bits. If a conflict is detected, the conflicted speculative region is aborted by discarding the speculatively modified data and resetting the bits. If `COMMIT` is reached without conflicts, the buffered speculative data are committed by gang-clearing the bits and sending the stores buffered in the store queue to the L1 cache. Figure 4.19 shows the hardware structures for the ASF implementation. In the figure, the components changed or added for ASF are shown in grey. The rest of the section explains the ASF implementation details.

4.5.2 Extended Transactional Instruction Implementation

Beginning a Speculative Region The `SPECULATE` instruction starts a speculative region and is microcoded in the microcode ROM. On detecting a `SPECULATE`, the extended instruction decoder sets the InSP (in speculation)[7] bit to remember the beginning of a speculative region. Making the decoder aware of the boundaries of the speculative region allows earlier marking of speculative accesses at a point where the instruction stream is still processed *in-order*, see the complications described in Section 4.4.3. Similar to the baseline proposal, the decoder signals the instruction dispatcher to read the `SPECULATE` microcode. The microcode (1) computes the next rIP so that rIP is restored to point to the instruction following the `SPECULATE` at an abort, (2) saves the next rIP and the current rSP in the shadow register file, and (3) executes an mfence (memory fence) micro-op. The mfence generates a dependency between `SPECULATE`

[7]More precisely a nesting depth counter to support nested transactions.

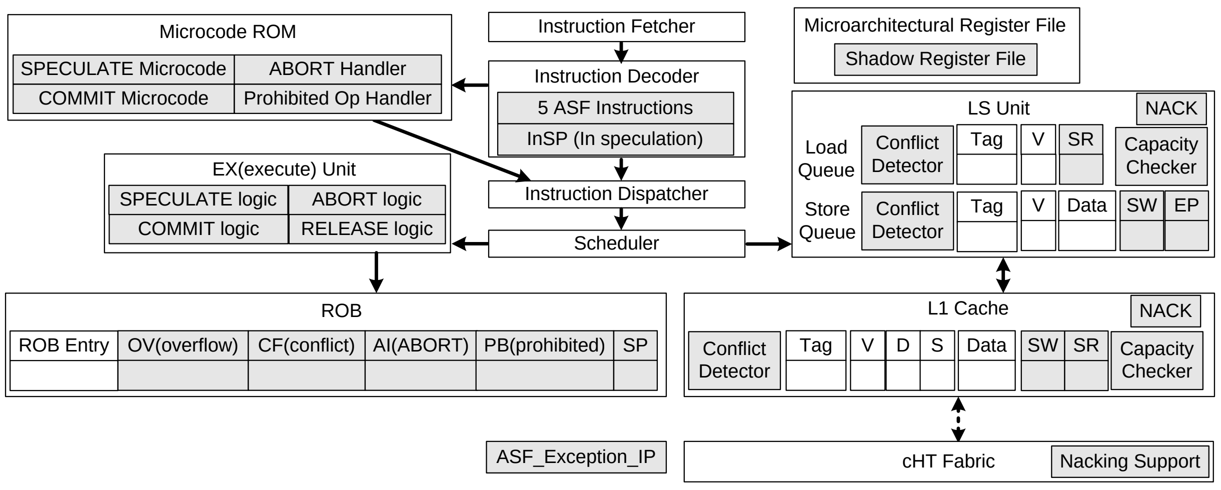


Figure 4.19: Extending an out-of-order core for transactional memory with focus on changing the flow of instructions.

and later `LOCK MOV`s, which prevents the `LOCK MOV`s from being executed ahead of the `SPECULATE` in the out-of-order execution stage (Section 4.4.2). The shadow register file is carved out of the existing micro-architectural register file used only by micro-ops.

Speculative Accesses To track speculative accesses, two bits are added per cache line: the SW (speculative write) bit for speculative stores and the SR (speculative read) bit for speculative loads as shown in Figure 4.19. The SW bit is also added per store queue entry, and the SR bit per load queue entry. A transactional access is issued to the LS (load/store) unit and sets the SW bit of the store queue entry for a store operation and the SR bit of the load queue entry for a load operation. The data movement operation executes in the same way as a normal access. The AMD64 TLB refill hardware allows a speculative region to survive a possible TLB miss during address translation by handling the miss in hardware. When the speculative access retires, the SR bit of the load queue entry is cleared, and the corresponding SR bit in the L1 cache is set, carefully observing the handover principles established in Section 4.4.5. Although sending another access to the L1 data cache at retire time can cause additional delay, it reduces transactional overmarking and allows for less complicated logic in the miss-buffer handling logic. The SW bit of the store queue entry is cleared when the speculative data are transferred from the store queue to the L1 cache along with setting the SW bit in the L1 cache. If speculative data are written to a cache line that contains non-speculative dirty data (i.e., the D (Dirty) bit is set, but the SW bit is not set), the cache line is written back first to make sure that the last committed data are preserved in the L2/L3 caches or the main memory.

Increased Worst-Case Capacity As a fall-back, the LS unit can assist buffering speculative data post-retire, if the L1 cache is out of capacity (for this particular index). More precisely, the transfer of the SW/SR bits from the load/store queues to the L1 cache needs to meet two conditions: (1) the access misses in the cache (i.e., no cache line to retain the bits) and (2) all cache lines of the indexed cache set have their SW and/or SR bits set (i.e., no cache line to evict without triggering a capacity overflow). In this case, the entry in the load/store queue is not deallocated, even though the associated instruction has retired. While the total capacity increase is small, this scheme helps to handle unfavourable access patterns that exceed the capacity of a few cache indices. Figure 4.20 depicts such an interaction.

If a non-speculative access meets the two conditions above, the L1 cache handles it as if the access is of uncacheable type (through a dedicated buffer outside the L1) to avoid a capacity overflow, and the L2 cache handles it directly. In order to hold as much speculative data as possible, the L1 cache eviction

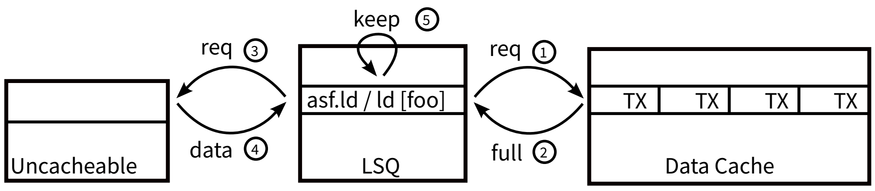


Figure 4.20: The load/store queue can be used as an additional storage container when a cache index is filled with transactional tracking data. If a request accesses the cache (1) and the corresponding set is full with transactional data (2), the LSQ will use the uncacheable path for getting the data (3) and (4), and keep the access present for conflict detection if it was transactional (5).

policy evicts cache lines without the speculative bits set first. Similarly, a cache line prefetched by a hardware prefetcher is inserted into the L1 cache only when it does not cause eviction of transactional data.

Extending Data Versioning Capacity Using the store queue to buffer speculative stores instead of the L1 cache needs careful integration with the logic for store visibility. Stores in the store queue are only locally visible in the AMD64 memory model [209]. Therefore, a store is visible to the rest of the system only after it is transferred to the L1 cache. To broadcast the existence of the buffered speculative store that cannot be transferred to the L1 cache without triggering a capacity overflow, an exclusive permission request for the store is sent directly to the coherent interconnect when the store retires in the store queue. This enables the other cores to detect a conflict against the store.

Once the exclusive permission is acquired, the store queue entry remembers the acquisition through an exclusive permission (EP) bit. The `COMMIT` instruction later checks the EP bit to make sure that the store has been seen by the rest of the system for conflict detection before starting the commit procedure. This is an example of the complication introduced in Section 4.2.3, and we will investigate details in the following Section 4.5.3.

Capacity Overflow An overflow exception is triggered when the load/store queues do not have an available entry for an incoming transactional memory instruction (i.e., the SW/SR bits of all entries are set in the queue the instruction needs to go to). A tricky problem is that non-speculative accesses should always make forward progress regardless of the number of the speculative accesses executed in a speculative region. Our design reserves one entry per queue for the non-speculative accesses to be able to execute them even when the rest of the load/store queue entries are filled with speculative data. As outlined earlier, OoO misspeculation may trigger a false overflow exception. To address this problem, we add an additional tracking bit per ROB entry: the overflow (OV) bit is set for an speculate access when it would cause a capacity overflow. If the speculative access is on a mis-speculative path, the ROB entry and the hardware resources associated with the entry will be discarded by the existing branch misprediction recovery mechanism. A true capacity overflow at that time will be serviced when the instruction becomes the oldest in the reorder buffer (the ROB entry reaches the head of the ROB) and triggers an overflow exception (assuming no other abort conditions exist before the speculative access). One last chance is given to such an instruction and it tries to allocate again in the load/store queue. The reasoning is that previous, OoO-*speculative* entries could have used up a slot that was later freed due to the instructions being on a wrongly speculated branch, and thus impacted an instruction on the correct path. Such misspeculation will be rectified once the OV-marked instruction has reached the head the ROB.

The combination of marking transactional lines in the cache only at the instruction's retire time (which is bound to have resolved all earlier misspeculations) and the freeing of load/store queue entries of instructions on any misspeculated branch allows for precise capacity tracking at the expense of an additional round-trip to the cache and additional logic in the OoO execution supporting structures (ROB and

LSQ, here). The solution outlined in the previous section is, in contrast, designed to minimise changes to the processor's core mechanisms.

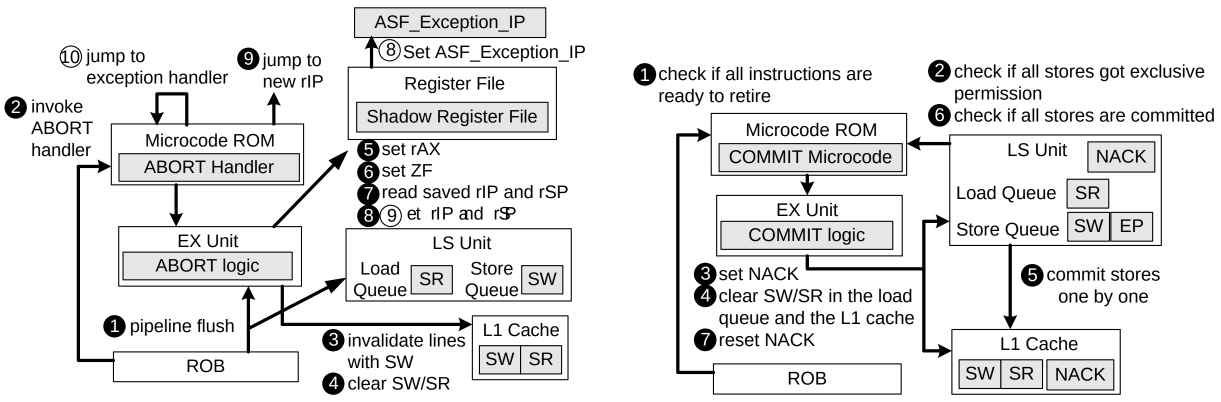
4.5.3 Enhanced Transactional Flows

Conflict Detection A conflict with another core is detected by checking the SW/SR bits in the LS unit and the L1 cache against incoming cache coherence messages. An invalidating message (for stores) conflicts with entries of the same address and set SW or SR bit. A non-invalidating message (for loads) conflicts only with entries with set SW bit. The baseline AMD processor already has the necessary content-addressable-memory (CAM) logic in the LS unit and the L1 cache for conflict detection. The LS unit has the CAM logic to check the address tags of all loads and retired stores for different purposes. The L1 cache has the CAM logic for address tags. These CAM logics are extended to read out the SW/SR bits for conflict detection.

Unlike a store, a load is problematic for conflict detection since a conflict against the load has to be detected from the moment the loaded value is bound to a register. Since the value is bound before the load retires, there can be a false conflict due to in-flight speculative loads (i.e., those that have not retired yet) on a mispredicted execution path. Similar to the handling of the speculative overflows, we add a conflict (CF) bit per ROB entry, to circumvent false conflicts. A conflict with a SR bit in the load queue sets the CF bit of the ROB entry of the conflicted load if the load is in-flight. If the conflicted load is on a mispredicted execution path, its ROB entry with the false conflict information will be discarded before reaching the head of the ROB. If not, the ROB invokes the abort procedure when the ROB entry of the conflicted load reaches the head of the ROB. A conflict with other speculative accesses (i.e., retired loads and stores either in the LS unit or in the L1 cache) is immediately reported to the ROB as a new interrupt, `ASF Conflict`, since the retired accesses are free of branch misprediction. On detecting the ASF Conflict interrupt, the ROB invokes the abort procedure for the conflict. The conflicted core replies to the conflicting core pretending that it could not find a matching cache line for the cHT protocol. Since the backup copy of the data is held in the L2 cache or main memory, requesters will access the pre-transactional version of the data.

In addition to maintaining precise tracking of the working set's size, delaying conflict detection (or rather acting upon observed conflicts) to after a speculative memory instruction is guaranteed to be off a mispredicted path avoids the problem of increased conflict surface due to transactional overmarking.

Aborting a Speculative Region As explained previously, a speculative region can be aborted for various reasons. An abort is triggered when the ROB detects any of the overflow, or conflict bits set for the retiring instruction, or the instruction is an abort or prohibited instruction and is the oldest ROB entry, or when the core receives an ASF Conflict interrupt (from the cache tracking the working set). By checking for abort / prohibited instructions only as oldest entries in the ROB, our design eliminates false aborts on a mispredicted execution path. The abort procedure shown in Figure 4.21(a) is very similar to the normal interrupt handling procedure and is used in all abort cases. The ROB first (1) initiates the pipeline flush that invalidates all ROB entries and load/store queue entries and (2) signals the microcode ROM with one of the ASF abort status codes. Then, the uninterruptable abort handler in the microcode ROM conducts the following procedure. It (3) invalidates the L1 cache lines with the SW bits, (4) clears the SW/SR bits in the L1 cache, (5) sets rAX with the signaled abort code, (6) updates the flags register, and (7) reads the saved rIP and rSP values from the shadow register file. At this point, the abort procedure bifurcates. If the abort is due to exceptions or interrupts, the microcode (8) sets the register ASF Exception IP with the current rIP (i.e., the one that triggered the exception), (9) sets rIP and rSP with the saved rIP and rSP values, and (10) jumps to the existing exception handler in the microcode ROM. The exception handler



(a) Abort flow. (b) Commit flow.

Figure 4.21: Aborting and committing speculative regions, showing interaction between functional units.

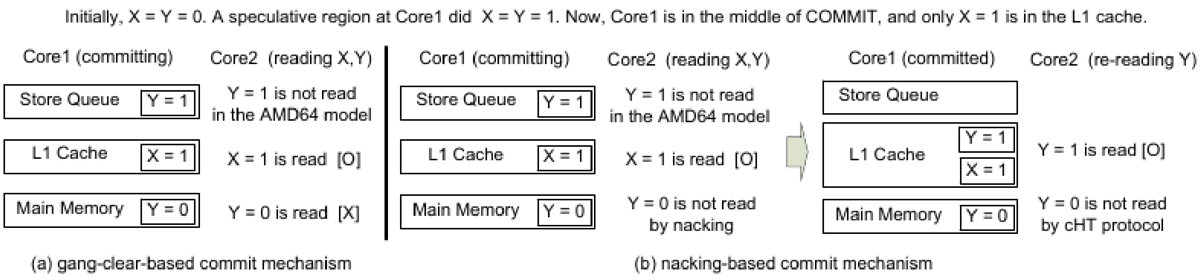

Initially, X = Y = 0. A speculative region at Core1 did X = Y = 1. Now, Core1 is in the middle of COMMIT, and only X = 1 is in the L1 cache.
Core1 (committing)
Core2 (reading X,Y)
Store Queue Y = 1
Y = 1 is not read in the AMD64 model
L1 Cache X = 1
X = 1 is read [O]
Main Memory Y = 0
Y = 0 is read [X]
(a) gang-clear-based commit mechanism
Core1 (committing)
Core2 (reading X,Y)
Store Queue Y = 1
Y = 1 is not read in the AMD64 model
L1 Cache X = 1
X = 1 is read [O]
Main Memory Y = 0
Y = 0 is not read by nacking
Core1 (committed)
Core2 (re-reading Y)
Store Queue
L1 Cache Y = 1 X = 1
Y = 1 is read [O]
Main Memory Y = 0
Y = 0 is not read by cHT protocol
(b) nacking-based commit mechanism

Figure 4.22: Illustrating non-atomic commits when tracking transactional stores not at the point of global visibility.

thus perceives the exception as being caused by the instruction following the SPECULATE, but can still access the original abort site.

All other abort reasons do not update the ASF Exception IP register, but instead directly update rIP and rSP (9), and then execute a jump micro-op to redirect the instruction fetcher (10) to the saved rIP (typically JNZ as explained in Section 3.2.2).

Committing a Speculative Region As outlined earlier in Section 4.2.3, the main challenge to committing a speculative region is that the simple gang-clear of the SW/SR bits in the LS unit and the L1 cache does not guarantee the atomicity of a committing speculative region under the AMD64 memory model [209]. The problem is that the LS unit is not usually the point of global visibility, and thus the transactional stores buffered in it need to traverse to the visibility point at commit atomically, together with exposing the contents buffered in the L1 cache[8]. Section 4.2.3 introduced the issue in general terms.

This problem is illustrated in Figure 4.22(a). In the figure, Core 1 ran a speculative region that wrote to X, Y, and other variables speculatively. The new X value ($X = 1$) happened to be buffered in the L1 cache, and the new Y value (Y=1) in the store queue. Now, Core 1 is about to commit the speculative region. If Core 1 just gang-clears the SW/SR bits to commit the speculative region, the new Y value will stay in the store queue. The problem is that if Core 2 reads X and Y at this point, it will read the new X value ($X = 1$) in the L1 cache and the old Y value ($Y = 0$) in the main memory.

This is because any value in the store queue of Core 1 cannot be read by Core 2 according to the AMD64 memory model (because the store queue is considered as the local write buffer of Core 1). This breaks the atomicity of the speculative region because Core 2 reads the mix of the new value ($X = 1$)

[8]We assume the L1 cache is the point of global visibility.

and the old value ($Y = 0$). To eliminate this problem, our design uses a nacking[9] mechanism that detects a conflicting cache coherence message from another core and nacks the message during the commit procedure. The core receiving the nack message retries the nacked cache coherence operation later.

In Figure 4.22(b), the commit procedure starts the nacking mechanism first and then gang-clears the SW/SR bits in the load queue and the L1 cache. Such behaviour is similar to commit-time locking in STM implementations. At this point, Core 2 can read the new X value ($X = 1$) in the L1 cache but cannot read any Y value since Core 1 detects the conflicting read from Core 2 by checking the SW bit set for Y in the store queue and nacks the read operation. On receiving the nack message, Core 2 retries the read operation and obtains the new Y value ($Y = 1$) after Core 1 completes transferring the new Y value to the L1 cache. In this way, the nacking-based commit procedure guarantees the atomicity of the committing speculative region by preventing the other cores from reading the old values of memory addresses (e.g., $Y = 0$ in this example) if the store queue of the committing core has new values to commit to the memory addresses (e.g., $Y = 1$). It is impossible for these cores to read the old value from their own caches, because those entries have been invalidated eagerly through invalidating snoops sent before the store updated the cache line.

Figure 4.21(b) shows the overall commit procedure based on the nacking mechanism. The `COMMIT` instruction is microcoded. On detecting a `COMMIT`, the instruction decoder resets the InSP bit and signals the dispatcher to read the `COMMIT` microcode. The microcode ROM has a feature to stall dispatching micro-ops until a wait condition specified in the microcode is satisfied. A new wait condition is added for `COMMIT` that checks (1) if all instructions in the ROB are ready to retire without exceptions and (2) if all retired stores in the store queue have obtained exclusive permissions (i.e., their EP bits are set). Once the condition is satisfied, the `COMMIT` microcode (3) signals the L1 cache and the LS unit to set their NACK bits. Once the bits are set, conflicting cache coherence messages are nacked instead of aborting the current speculative region. There is no deadlock due to the nacking mechanism because the committing speculative region holds all necessary exclusive permissions to complete the `COMMIT`. Then, the commit handler (4) clears the SW/SR bits in the L1 cache and the load queue to first commit the speculative data in them. This (5) naturally enables the store queue to resume transferring the speculative data to the L1 cache since the SW/SR bits in the L1 cache are now cleared. Pushing the data out from the store queue into the L1 will displace other lines that were transactionally accessed during this transaction. The data transferred from the store queue do not set the SW bits in the L1 cache (done by checking the NACK bit) since the completion of the data transfer at this point means that the data are committed and made visible to the rest of the system. Due to the eager snooping, the sequentially published transactional stores will logically still become visible at one point in time (compare that with the sequential unlocking of commit locks in STM), and all pending loads that observe them occur after transaction commit. The microcode ROM stalls on another wait condition that (6) checks if no store queue entry has the SW bit set. By the time this condition is met, all new values in the store queue are transferred to the L1 cache. Finally, the commit logic (7) signals the L1 cache and the LS unit to reset their NACK bits, and the commit procedure completes here. Nothing is done to the shadow register file since it will be overwritten by the next `SPECULATE` . Since the ASF hardware context in this design can support a single speculative region, a speculative region should not enter the out-of-order execution stage before the previous speculative region commits. This case is naturally handled in our design since the microcode ROM prevents the instructions behind a `COMMIT` in the program order from being dispatched until the `COMMIT` completes.

Reliance on Eager Coherence This commit procedure relies on eager acquisition of write permissions for stores, and adds a NACK message to the protocol. Eager store acquisition is often found in most coherency protocols, but may be weakened in particular for systems with weaker memory consistency

[9]from NACK; negative acknowledgement

semantics (non-multi-copy atomic systems, IBM and ARM systems, for example). In non-eager systems, writes may not send out snoops to all (potential) shares before writing, or may decide not to wait for all acknowledgements. Instead, writes could “diffuse” through the system or be grouped and made visible to nearby cores, first.

Alternatives to “NACK” Messages Together with the early write permission acquisition for transactional stores buffered in the store queue, and depending on the exact details of the coherence probing mechanisms, it might be sufficient to delay sending an acknowledgment message in response to the read probe (for Core 2’s initial read of Y) in the example, instead of explicitly sending a “NACK”. We observe that the read *must check core 1*, at least up to the point of global visibility, and usually such a probe message is answered with either an empty acknowledgement, or a reply with the updated data, and ultimately, Core 2 needs to wait for all such replies (we are considering only eager / strong memory subsystems here). If the interface from the L1 cache to the store queue (1) sends probes arriving in the L1 cache to the store queue and (2) has a return channel, no new “NACK” messages may be required, stalling on ACK reply suffices instead. Deadlock needs to be considered, but again due to the early write permission acquisition for transactional entries in the store queue and tracking of remote write permission acquisitions during the transactions life time, write permissions do not need to be reacquired for the transactional stores in Core 1’s store queue. These can therefore be propagated to the L1 cache without causing deadlock-inducing waits for responses.

Handling Branch Misprediction Mispredicted branches before or in the middle of a speculative region are troublesome since the speculative region can be fetched and executed along a mispredicted execution path. Our design uses the existing branch misprediction recovery mechanism to restore the ASF hardware resources occupied by the mispredicted ASF instructions of the speculative region. When the misprediction is detected, the recovery mechanism naturally discards the ROB entries and the load/store queue entries occupied by the mispredicted instructions. Nothing is to be done to the L1 cache since the SW/SR bits are transferred to the L1 cache only for retired instructions, and the mispredicted instructions never retire. The NACK bits in the L1 cache and the LS unit are not set at this point since a mispredicted `COMMIT` instruction will never start the commit procedure. This condition is true because the mispredicted branch will flush all younger instructions (including the `COMMIT`) when the branch resolves. This means that the `COMMIT` instruction needs to become the oldest in the ROB first and then transaction commit can commence:

$$execute(branch) \rightarrow mispred(branch) \rightarrow retire(asf.commit) \tag{4.7}$$

$$retire(asf.commit) \rightarrow enable(NACK) \tag{4.8}$$

The InSP bit is requires careful handling since it has to be restored to the value at the moment the mispredicted branch was decoded. To restore the InSP bit properly, the instruction decoder tags the up-to-date InSP bit values along with the decoded instructions. The InSP bit is added per ROB entry to record the tagged InSP bit value of each instruction (similar to Power [340]). When a branch is found to be mispredicted, the SP bit of the branch’s ROB entry is used to restore the InSP bit. If the mispredicted branch is before the speculative region, its SP bit is 0 and the InSP bit is reset. If it is in the middle of the speculative region, its SP bit is 1 and the InSP bit is set. Simple flat nesting of transactions replaces the simple InSP bit with a counter that tracks transactional depth per in-flight speculative instruction and in the core’s front-end.

Exceptions, Interrupts and Prohibited Instructions Exceptions and interrupts in a speculative region are handled first by the ASF abort handler to abort the speculative region and then by the existing exception handler in the same way as they are handled without speculative regions. If an instruction before a `SPECULATE` in the program order triggers an exception after the `SPECULATE` has entered the execution stage, the ASF hardware context is restored in the same way as a mispredicted branch before a `SPECULATE` is handled.

Since the prohibited instructions should not be allowed to enter the execution stage and modify non-speculative resources such as segment registers, our design detects them early at the decoding stage when the InSP bit is set. On detecting the instructions, the instruction decoder signals the microcode ROM to jump to the prohibited op handler. The handler (1) executes a micro-op that sets the PB (prohibited) bit of its own ROB entry, and (2) waits for the entry to reach the head of the ROB. Then, the ROB picks up the exception and initiates the abort procedure. This way, the prohibited instructions in a speculative region never enter the execution stage.

4.5.4 Early Release of Transactional Data

The `RELEASE` instruction is complicated to implement since it builds dependencies with `LOCK MOV`s around it. The problem is that the core will attempt to coalesce multiple memory accesses in an instruction window to a single cache line. The `RELEASE` instruction must not release the cache line too early, that is before all preceding accesses to the same cache line have completed, and also not too late, not after a later memory instruction has re-added the memory location to the transactions working set.

A simple implementation might want to add memory fences before *and* after the `RELEASE` instruction, but we have opted for an implementation with less constraints on ordering: We use the head execution feature of the ROB to implement `RELEASE`. Once dispatched, a `RELEASE` does nothing until its ROB entry reaches the head of the ROB. When the entry reaches the head, the ROB signals the `RELEASE` execution logic to search for the SR bit to be reset by the `RELEASE` only in the L1 cache and in the portion of the load queue that contains retired loads. After resetting the matching SR bit, the `RELEASE` logic signals the ROB for its completion. This way, our design abides by the dependency among the `RELEASE` and the `LOCK MOV`s around it since the loads behind the `RELEASE` have *not retired* yet and their SR bits are not examined by the `RELEASE` logic. These loads will then reset the SR bin the cache once they retire.

4.6 Alternative Microarchitectural Implementation Variants

In addition to the two implementation variants presented in detail in the previous sections, several other variants may be appropriate for specific system structures. The following two implementation sketches show how the structure of the cache hierarchy and a desire to target a small set of changes can be realised.

4.6.1 AMD Bulldozer Microarchitecture Integration Sketch

AMD's "Bulldozer" (BD) microarchitecture [136] has been developed as a revolutionary deviation from the previous succession of K7 / K8 / Greyhound (sometimes referred to as "K10" in non-AMD material) microarchitectures and therefore also provides a new design concept with cores sharing FPUs, front-ends and more importantly, a new memory hierarchy. I will briefly highlight how this affects the implementation of ASF and complications and extensions for such a design.

Aside from changes to the core microarchitecture, the most significant changes to the Bulldozer architecture are changes to the cache hierarchy: BD employs write-through L1 data caches, inclusive with the L2. Writes update the L1 upon retire and proceed to write-through to the L2. The core-side write-port of the L2 has a *write coalescing cache* (WCC) which is a specially organized sub-cache inside the L2 that allows high sustained write-bandwidth to sink the traffic caused by the write-through organisation. While the L1 data caches are dedicated per core, the L2 is shared between two cores / L1 data caches (sometimes called a “module” or “compute unit”).

In BD, stores are only visible globally once they are written-through to the L2. This means that transactional memory implementations that use the L1 data cache for conflict detection and data versioning will need to take the distance between the versioning mechanism and point of global visibility into account (Section 4.2.3). In essence, the implementation will need to send the transactional updates from the L1 / core to the L2 upon transaction commit, without allowing interference from other cores, and without re-requesting written cache lines from the system; similar to the mechanism used for versioning in the LSQ in Section 4.5.3. One positive side-effect of the inclusive, write-through nature of the L1 / L2 cache arrangement is that the L2 can easily hold the pre-transactional copy to roll back transactional stores. However, the L1 data cache is not designed to perform write-back operations, and thus the transactional commit cannot perform the delayed write-through from the L1 data cache into the L2. If the L2 will not be involved in the versioning of transactional data, a separate mechanism needs to be used to buffer the transactional writes between the core and the L2 cache. The size of that component will dictate the maximum write set size of the transaction. One option is to use the WCC as such a component.

Another option would be to perform data versioning (and conflict detection) in the L2, but the higher latency from the core to the mechanism used for data versioning / conflict detection is known to negatively affect performance [270]. In addition, the L2 is shared by two cores, requiring dedicated tracking bits for each of the cores.

4.6.2 Pipeline-only Implementation

Given the speculative substrate of an out-of-order microprocessor, a low-cost implementation alternative may be to reuse those mechanisms for implementing transactional memory. Transaction begin is essentially handled as an unresolved branch that will only be resolved once the entire transaction is ready to commit. The existing snooping logic is repurposed as conflict detection logic with the addition of detecting remote reads to entries in the store queue and the store queue buffers transactional stores.

One consequence of such a design is that the size of transactions is limited by the size of the buffers used for out-of-order speculation. Those buffers are usually sized as small as possible (due to area and energy constraints) and hence will allow only small transactions. The reorder buffer (50 – 200 entries) limits the number of total instructions per transaction; and the load / store queue will constrain the number of memory operations per transaction.

A second consequence is that events disrupting the base speculation mechanism (such as branch mispredicts) will cause aborts of the transactions, too. Finally, while the reuse of the OoO speculation mechanisms may save total area (transistors, wires), the mixing of the components and deep integration has significant impact on complexity, and may negatively impact timing. We therefore chose not pursue this path in our work at AMD, instead we point to the challenges the Sun Rock processor with a similar approach was facing and its ultimate failure [188].

4.7 High-level Interaction Between Microarchitecture and HTM

After looking at several microarchitectural implementation options, I will summarise the effect of uarch features on high-level properties of the implemented HTM.

4.7.1 Visible Microarchitecture Artifacts

Several aspects of the microarchitecture will be visible to the programmer using the HTM functionality:

Capacity Constraints Most, if not all BeHTMs provide no, or only a soft, perspective on the size of the transactions they support. That size will be determined by the read / write set sizes and how well they fit into the chosen tracking structures for conflict detection and versioning. There is unfortunately a performance cliff that applications will observe. Transactions either fit into the cache (or other tracking structure), or they don't. Some transient aspects and bin-packing variations exist, but for the largest part, applications will experience significant slowdowns immediately when they exceed the provided structures. Imprecise structures that trade large / infinite capacity for tracking precision loss will generally degrade more gracefully; yet do not offer versioning, and can complicate progress analysis.

Transactional Overmarking As a result of the interplay between OoO-speculation and transactional implementation choices, the resulting capacity available / used can be larger than what a stepwise execution of the program would have required. As a net result, capacity constraints become harder to predict and also progress can be affected (due to additional conflict "surface").

Performance Depending on the implementation choices of the HTM, transactions can affect the performance of the code they are running; in addition to the concern of transaction aborts and progress. Several implementation choices require work at the entry and exit of the transaction (to stash away registers) and may serialise the instruction stream around them (implicit barriers / fences). Furthermore, some variants can reduce ILP inside the transaction by reducing the number of OoO speculation resources available (by reserving physical registers, LSQ entries). Finally, if transactional accesses need to perform additional work in the memory hierarchy (multiple trips to the cache for conflict detection, writing back a pre-transactional copy), that work can either take longer, or the additional bandwidth requirements can slow down overall execution (such as reducing effective cache interface bandwidth).

4.7.2 Progress

Overall, guaranteeing progress in hardware transactions is complex, for three main reasons: (1) spurious aborts, even without contention, (2) resource depletion of transactional resources, and (3) guaranteeing global progress with local policy.

In the first category fall all the transaction aborts that are caused by imperfections of the underlying microarchitecture. Speculation itself is a best-effort feature and microprocessors will fall-back to non-speculative execution in extraordinary (and hopefully non-performance critical) circumstances. Depending on the coupling between OoO speculation and TX speculation, failure in OoO speculation may cause failure of TX speculation. The Sun Rock processor [188], for example, will abort transactions on branch mispredicts and TLB misses. Notably, these events are not visible at the architectural level, i.e., applications cannot directly observe them happening.

Even on microarchitectures that decouple OoO and HTM mechanisms, for example the ones described previously in this chapter, architecturally invisible events can still obstruct transaction progress, for example scheduling issues or corner cases in the coherence protocol. Several examples were presented as implementation challenges / bugs in Section 4.4.3 and 4.4.6; their workarounds could also have involved aborting the transaction at detection of such a corner case. It is conceivable that real designs will err on the side of aborting too many transactions rather than putting transactional safety at risk.

Events that have a large, visible effect on the architectural state may also abort transactions, for example calling into the operating system / hypervisor explicitly or implicitly due to an exception (page fault) or interrupt.

Broadly, the second category of resource depletion will encompass cases when a transaction requires more resources than are available in the transactional facilities. Depending on the implementation, transactions may be limited in instruction count (size of the reorder buffer), number of loads / stores (size of load / store queue), amount of transactional data used (size of the tracking structure, e.g., the L1 data cache), and related metrics, such as address patterns in indexed associative tracking structures. The latter is particularly important, because the worst-case capacity of a cache-based implementation may be limited by the associativity of the cache. While there are academic proposals even to bypass the capacity constraints of caches, they are *firmly* outside of what is deemed practically implementable as of today [80, 86].

Finally, one of the paramount principles outlined in this work, is to not adversely affect the original microarchitectural substrate and protocols, but instead have a minimally-invasive HTM implementation. Adverse conflict patterns between two or more concurrent transactions may cause progress issues. Simple overlapping access patterns of two transactions with reverse access order in the second may cause livelock, when both transactions conflict on their second access with the first access of each other and simply retry (in lockstep).

Unfortunately, other, less-obvious effects can complicate the conflict scenario. False sharing of different data in identical cache lines can hide address patterns from programmers. Furthermore, conflict-inducing coherence messages may be exchanged for reasons unknown / invisible to the application. Hardware prefetches will speculatively pull in data in order to reduce latency on future accesses. These prefetches may cause conflicts with concurrent transactions' read and write sets. While most stream prefetchers will prefetch data for reading, processors may use exclusive prefetchers for OoO speculative stores, thus coupling local misspeculation (on a branch containing stores) with remote spurious transaction failure.

Additionally, specific coherence messages may convert resource limitations elsewhere into transaction aborts, for example enforced evicts due to limited snoop filter / directory capacity.

In summary, there exists a large number of reasons for progress-hindering aborts of transactions. In the common case, these are expected to be rare, due to sufficient warm-up and expected to go away after a small number of retries (predictors warmed up, working set paged in); but persistent corner cases may exist, among them resource limitations and byzantine actual conflict patterns.

For all these reasons, it is hard to give general progress guarantees, such as wait-free / lock-free execution. Best-effort HTM systems are thus favoured due to the fewer constraints they impose on the underlying microarchitecture, speculation mechanisms and misspeculation recovery. These fewer constraints, however, do not make the feature trivial to implement; BeHTM implementations are hard as the observed bugs in my designs have shown and also the issues Intel has had with its first commercially available HTM implementations [365, 366].

In ASF, the architecture attempts to give a very limited progress guarantee; essentially obstruction-freedom when the number of transactional cache lines is smaller or equal to four. In this case, exceptions and interrupt events are also treated as obstructions. Thus, the guarantee means that small transactions that do not fault and execute with disabled interrupts will succeed *eventually*. In real life, the reasonable expectation is that page faults will not persist indefinitely (due to the OS page tables holding all required data) and interrupt rates will allow short transactions to complete between two interrupt events.

For the microarchitectural implementation, even such a weak guarantee will severely restrict the freedom, as described earlier. Even though the microarchitectures inside the simulators are relatively regular and cannot expose all quirks and corner cases present in product microarchitectures, the challenges introduced and solved in previous sections are indicative of the class of problems expected in real CPUs, but will very well present only the tip of the iceberg.

4.7.3 Influence of ASF-specific Features

Some of ASF's features influence and complicate hardware design. Because ASF provides a limited capacity guarantee (effectively obstruction freedom for transactions with up to four cachelines worth of read / write set), the hardware needs to provide that. That means that hardware cannot arbitrarily abort the transactions, but must attempt to run small transactions in earnest. The guarantee does not guarantee (progress) under contention, and also does not protect against external abort causes such as interrupts. Instead, the core must carefully handle branch prediction and how it adds entries to the transactional working set, especially with an associativity limited tracking structure. In our chosen implementations, different structures effectively permit speculation and the limiting structures only track working set when the accesses are not speculative anymore. Furthermore, our designs support cache and TLB misses while staying in transactional mode; and generally do not contain any unnecessary aborts.

Another influence on the microarchitecture are ASF's features of non-speculative accesses and only partial register checkpoints. Both of these allow state to leak from the transaction; the first challenge was to specify what behaviour programmers could expect to see of these accesses. Secondly, hardware of course must carefully adhere to the specification and not leak speculative transactional data through (falsely) aliasing non-transactional stores, and also ensure that the register file state at the abort time corresponds an appropriate position in the code.

Non-transactional accesses must of course be distinguished from transactional ones, complicating the flow and verification space of the load / store path of the core.

Together, however, ASF's limited amount of register checkpointing reduces the need for complex register stashing or renaming schemes; allows fast transaction entry and abort; and does not reduce ILP inside the transaction. Furthermore, non-transactional accesses allow more careful selective annotation and so a smaller tracking structure can be useful to more code that performs many operations on private memory.

Similar to overlapping tx and non-tx accesses, the `RELEASE` instruction requires *careful* sequencing in the instruction stream in order to not release later accesses that were hoisted before it due to out-of-order execution.

4.8 Summary

In this chapter, I have presented various options for implementing a BeHTM ISA extension with the example of AMD ASF. While conceptually simple on the ISA layer, implementations are complex because of the cross-cutting responsibility of different hardware mechanisms to coordinate transaction execution safely. Key challenges for any HTM implementation are the integration with other present layers of speculation, most prominently out-of-order execution in high-performance cores, and the integration with the specifics of the cache hierarchy and memory system. There, ensuring that conflicts are tracked through the life-time of transactions in spatially separate structures without any windows of vulnerability due to transition between mechanisms is crucial for correct transaction execution. Furthermore, choosing the right location for data versioning and making all stores of a successful transaction visible to the system at once while still watching for conflicts can be challenging depending on the layout and strength of the underlying memory substrate.

Additionally, there is a strong relationship between the chosen microarchitecture and easily observable application characteristics; in some cases resulting in a step-wise change for small input perturbations. On the other hand, the features that distinguish ASF from other "run-of-the mill" BeHTMs also need careful consideration in the microarchitectural realisation.

Overall, however, it is possible to implement BeHTM with no, or very simple, changes to the overall memory system architecture and cache coherence protocol. The required changes to the baseline CPU core and caches are, however, still complex for a relatively “vanilla” BeHTM; therefore several of the proposals from the literature that require more invasive modifications seem prohibitive due to the required complexity and verification cost – especially for first generation systems.

Chapter 5

Applications and Evaluation of HTM

5.1 Introduction

Transactional memory, like any other microprocessor feature, does not live in a vacuum, but instead both ISA design and microarchitectural implementation characteristics need to work well with the applications that use them. One of the first steps in design and implementation of such a processor extension is the analysis of workloads and understanding requirements and characteristics of potential use of the feature.

Therefore, after introducing the ISA of a BeHTM (Chapter 3) and micro-architectural implementation options and details (Chapter 4) in the previous chapters, this chapter will highlight use cases for BeHTM mechanisms and performance results of our BeHTM implementations described. After understanding requirements and usage, we build architectural and implementation prototypes to study interactions on both architectural and microarchitectural levels.

The architectural level analysis leads to an understanding of usability of the feature and allows testing of prototype software with necessary changes to support the proposed feature. In the case of BeHTM, the architectural component is important, because of new control flow interactions (transaction aborts), new instructions (transaction start / end, marked / unmarked memory accesses), and as a vehicle to test compiler backends, transactional memory libraries and hand-crafted use of BeHTM in concurrent data-structures and higher-level primitives (such as DCAS).

During my work on ASF, I was very fortunate to collaborate with our our partners in the VELOX EU-funded project who contributed compiler, language, and library support on top of ASF, and used this thesis' simulation infrastructure for testing and performance evaluation of new compiler techniques. This collaboration resulted in multiple joint publications [158, 210, 213, 214, 220, 254, 274, 289, 337]; some of which (workshop papers without formal proceedings) I have attached to this thesis (Appendix B) and will use / paraphrase here for illustration.

Building an actual microprocessor for this thesis work was infeasible, so I implemented all described microarchitectural mechanisms inside detailed simulators (PTLsim [135] and Marss86 [253]). In fact, only close to the end of my thesis have products with enabled BeHTM actually started shipping (Intel - Q3-2013 [278, 286], IBM - Q3-2012 [270, 281, 340]) and been disabled again due to implementation bugs (Intel - Haswell, Broadwell, Skylake [366]). Through my exposure to actual proprietary microarchitectures (mainly at AMD 2006 - 2012), I have tried to keep the simulator implementations as realistic as possible, without sharing proprietary microarchitectural implementation details.

Despite a faithful simulator implementation, several fundamental differences remain between simulators and actual implementations in RTL / silicon. Since I have not published about these, I will use some space in this chapter (Section 5.4) to generally introduce high-level differences / characteristics of simulation environments (Section 5.4.1), show where this simplified my implementation (Section 5.4.3), and finally present cases where the simulator actually complicated the design significantly (Section 5.4.4).

The remainder of this chapter is organised as follows: Section 5.2 will briefly introduce usage scenarios for BeHTM and summarise characteristics of the applications studied. Section 5.3 will present highlights of the numerous performance studies undertaken with my ASF implementation and will discuss additional background information, not present in the selected publications.

Section 5.4 will present backgrounds and detailed analysis of the actual simulator implementations, simplifications and complications; and I will summarise and conclude this chapter in Section 5.5.

5.2 Applications of HTM

Broadly speaking, transactional memory can be used in either a software visible or invisible fashion. Software visible uses require changes to applications in either high-level programming languages, such as C / C++ or Java, or direct usage of the primitives / instructions in assembly or through compiler intrinsics.

Invisible uses of HTM integrate the mechanism "under the hood" of either existing synchronisation mechanisms (e.g. Java's `synchronized` blocks, pthread mutex / locking primitives), into language runtimes (e.g. the Python interpreter) or invisibly into the CPU to accelerate existing locking primitives.

Concurrently with my work on the ISA and hardware implementation of ASF, the popular C / C++ programming languages gained language support for threads, concurrency, memory models and synchronisation in the C++11 standard [268]. The experimental extension for `tm_atomic` blocks is not yet part of the standard, but is being discussed officially for eventual language integration [292]. Atomic blocks group instructions and effectively provide transactional memory semantics directly at the C++ language level. The work undertaken in the VELOX project including the work by Red Hat, University of Neuchatel, and TU Dresden on the compiler and runtime, and AMD's work with significant contributions by me for providing a realistic testbed for the semantics have had and continue to have an impact on the design of these primitives.

Since the work on C++ extensions had been concurrent with my work on ASF, not many code examples exist in C / C++ that use concurrency and locking extensively. Therefore, parts of our analysis uses STAMP [166], a benchmark suite for transactional memory, handcrafted data structure benchmarks, and also prototype integration into other languages, such as Python and Java. Finally, we identified a way for *semi-transparent* lock elision through the use of a pthread library wrapper. The following sections will present an overview of each of these use cases.

5.2.1 Direct HTM Usage

Microbenchmarks Several concurrent data structures lend themselves easily to usage of HTM to simplify and accelerate the algorithm. In my thesis work, I have used, extended and evaluated concurrent integer set implementations that are based on various data structures: linked lists, skip lists, RB-trees, and hash sets.

On a high-level, there exist three ways how HTM can be employed in concurrent data structures. Direct usage wraps the entire traversal and update of the underlying data structure into a single transaction. This approach ensures that all data structure accesses are properly synchronised by providing linearisability through the hardware transactions' strict serialisability property.

Especially for linear data structures, this can cause performance problems, because every transaction accumulates a read-set with size $O(N)$ for an N entry structure. Therefore, for some data structures I have split the data structure operation into a non-transactional scan part with subsequent transactional

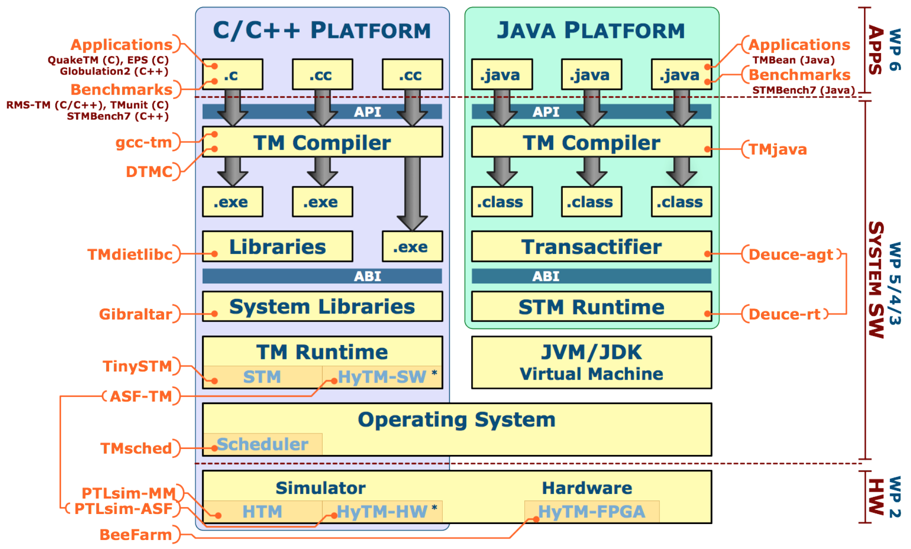


Figure 5.1: The VELOX stack showing the different components created and modified for evaluating Hardware Transactional Memory.

modification. Scanning traverses the linked data structure without acquiring locks or using transactions; traversing until the region that will be modified by the operation is found. The modification is achieved as follows: `SPECULATE`; check data structure consistency, e.g., elements still present and linked; perform modifications; `COMMIT`.

This algorithm borrows from conventional fine-grained locking approaches. One difference is that the scan phase will abort concurrent data structure modifications if they have already performed writes in their transactions, but have not yet committed. Therefore, the readers do not need to check for concurrent modifications (through scanning for locks, acquiring traversal locks or read-lock per-node read-write locks), which improves their performance.

While generally improving performance, this algorithm does not easily compose, despite using HTM, if the composition maintains the performance-preserving non-transactional scanning phase. Such composition is possible with ASF, where the scanning phase can be located inside a transaction (the one used for composing the two data structure operations), but still be performed non-transactionally (with unmarked loads in non-inverted ASF).

Finally, a third option to use HTM in concurrent data structures is to use a lower-level primitive synthesised from HTM, such as DCAS, and use that in a lock-free mechanism that will update the data structure. In the linked-list case, DCAS can be used to monitor the to be deleted element's next pointer and at the same time swing around the previous element's next pointer.

Large Applications - STAMP Microbenchmarks are a great way of debugging and testing performance of an HTM implementation. They also reflect a usage of transactions in small scale, mostly homogeneous settings. The STAMP benchmark suite [166] is designed to be characteristic of large-scale TM usage, and therefore presents larger transactions with a more diverse set of operations. To enable direct usage of transactional memory, the global memory accesses and beginning / end of transactions have been identified by hand and therefore allows direct usage of transactional primitives.

Language- and Library-Level Integration of HTM While hand annotation for transactional memory applications is possible for small benchmarks, the manual effort required to instrument all the right memory accesses and check for escaping function calls and other exits is significant. This work should ideally performed by a compiler that understands programming language-level transactional constructs and adds the required transactional memory accesses for data and also inserts instructions to start / end a transaction.

The most typical way for describing transactions in C-like programming languages is a special basic block `tm_atomic { ... }` wrapping an arbitrary sequence of code into a transaction.

During the course of this work, multiple transaction aware compilers were created by our partners in the VELOX research project: Deuce for Java [226], and DTMC (the Dresden TM Compiler) based on LLVM for C++ [150, 210], and finally gcc-tm – a fork of GCC which adds support for atomic blocks (now part of mainline GCC)[1]

Figure 5.1 shows the entire VELOX stack with different applications, languages, compilers, and libraries for the evaluation of transactional memory.

Conceptually, these compilers work in a very similar fashion. They instrument entry and exit paths of the basic block and add calls to a transactional-memory library. In addition, the compiler invokes read / write barrier functions of the TM library for every access to (potentially) shared memory. Finally, the compiler instruments called functions and checks that they do not cause problems, such as I/O, system calls etc., which can escape the transactional memory mechanisms, and adds additional code required for the control flow of transaction abort.

The transactional memory functionality is provided by the TM library, which can implement various TM algorithms. For HTM, the TM library mainly acts as a thin proxy layer, translating the compiler-identified memory accesses and transaction start / end into the right use of the HTM primitives. In addition, the TM library may provide fallback paths in case a hardware transaction repeatedly fails. The simplest of such is grabbing a global lock in case of repeated transaction abort.

If transactions get mapped onto HTM, the overheads of the transactional read / write barriers have to be small. In the small language integration layer introduced in Chapter 3 (Section 3.3), this is achieved by using only single instruction inline assembly sequences for the barriers and short, hand-crafted assembly sequences for transaction start and end.

For compilers that support TM language primitives through a TM library, the work per read / write barrier might be more significant, including a function call / return and parameter passing. In case of STMs that have to perform more work per transactional access, these overheads may not be too significant. For ASF, where the barrier essentially is a single `LOCK MOV`, these overheads have to be removed. Fortunately, the DTMC compiler uses LLVM which can perform aggressive inlining of functions at link time[2], which significantly reduces overheads.

For most of the work here, I have used and extended TinySTM with various ASF backends. Our joint publication at EuroSys 2010 describes the stack and presents results for various HTM implementations and benchmarks [213].

5.2.2 Lock Elision

In addition to using transactional memory directly, transactions may be used to convert lock-based critical sections that pessimistically force serialisation through *lock elision* into optimistic transactions which serialise only if concurrent executions conflict. There are various ways to perform such a transformation.

Manual Software Lock Elision The programmer manually converts the usage of locks / critical sections into the right transactional primitives (atomic blocks, direct usage of the TM functions / instructions). The lock usually remains as a fall-back path, in case the critical sections do not make progress with TM.

[1]See `https://gcc.gnu.org/wiki/TransactionalMemory`.

[2]GCC eventually also received link-time optimisation, but as of today (version 4.9) is unable to inline the transactional memory barriers.

Failing transactions will therefore eventually acquire the lock and execute the original content of the critical section, guaranteeing progress properties of the original critical section construct.

To properly serialise between lock-taking critical sections and concurrent lock-eliding transactions, the transactions will have to check that the lock variable is free at their linearisation point. This can be easily achieved by transactionally reading the lock and aborting if it is not free. In that case, if the lock is free it will remain free until transaction commit / abort. An interesting trade-off is when the lock should be checked for reading: pushing the check to the end of the transaction reduces the contention window with lock-acquiring critical sections, but may also cause the transaction to run off inconsistent state that is still modified by the critical section. While the transaction will eventually abort in that case, either because it conflicted with one of the "consistent-making" stores of the critical section, or it finds the lock taken, inspecting inconsistent state may cause anomalies, in particular, if the transaction can exhibit side-effects as is the case in ASF. In the literature, this is called "lazy subscription" and considered unsafe [244, 311].

Integrated and Semi-Transparent Software Lock Elision Performing lock elision "under the hood" without the programmer changing code is compelling because it unlocks potential performance gains for existing programs without code rewrite. Potential targets are eliding locks in language level constructs, such as Java's `synchronized` blocks, and C++ `std::mutex` or derived locks; or locks present in interpreted languages, such as the global interpreter lock in Python. Similarly, at a lower level the `pthread_mutex_*` library interface can be adapted to perform transactions instead of acquiring the associated lock.

The general mechanism remains the same as in manual lock elision: instead of acquiring the lock, the code is changed to start a transaction instead. If the HTM then treats normal memory loads / stores as transactional (ASF inverse mode), the modifications can be localised in the locking primitives without requiring generation of two code paths with instrumentation of all shared accesses in one of them. Localising the changes to the locking code can then be accomplished without changing the application code; if shared libraries are used for the implementation of the locking primitives, the application binary may remain the same. There are patches that change the popular GNU C libary [318] to perform elision in the `pthread_mutex_*` functions using HTM. In our work on ASF, we have investigated approaches that use `LD_PRELOAD` to load an interposer between the application and the exiting implementations of these functions. In addition, we have added lock elision for locks used inside Java HotSpot [290], and Python (driven mostly by Martin Pohlack, unpublished).

In some of these, complications arise from varying interfaces to the locking primitives. If locks are elided, the underlying lock is not physically acquired, because that would require a write which in turn would conflict with other eliding transactions. Instead, the lock is acquired logically: through a combination of transactional tracking of the data and enforcing the lock free state for the duration of the transaction. Locking interfaces which permit querying the lock variable while holding it (e.g. C++ `std::unique_lock::owns_lock`, or performing `trylock` operations on held locks) may need adaptation to precisely distinguish between the following cases: no lock held, not eliding; holding the queried lock; eliding queried lock; eliding a separate lock; running a separate transaction. Additional bookkeeping may be required, but should be performed thread-locally and in a transaction-safe manner in order to avoid transaction aborts.

Transparently eliding locks also keeps the programmer unaware of the changed performance trade-offs. In order to get good performance out of an elided critical section, the programmer must: (1) avoid writes to shared data, in particular data that is often shared between threads, but possibly not part of the core synchronisation set of the algorithm; (2) be aware of conflict detection granularities, and avoid false sharing; (3) avoid I/O and system calls as these tend to abort the transactions, too.

While all of these points constitute good advice for making (non-elided) critical sections run fast (and in fact most parallel code, including other lock-free techniques), due to reduced cache line bouncing and reduced length of the critical section, they will only slow down execution with locks, whereas they can permanently thwart progress with transactions and reduce performance to lower than the fall-back path due to previous retries. Conversely, it may be surprising for programmers to have expected performance gains for elided, non-conflicting critical sections suddenly evaporate due to changes / mechanisms invisible or unintuitive to the programmer.

Due to the unknown nature of the critical section and the cost of unsuccessful transactional execution that eventually has to grab the lock, semi-transparent approaches benefit greatly from *prediction*: a predictor is consulted with the identity of the critical section / lock and returns whether eliding the lock acquisition would be beneficial or not. If the predictor indicates that elision is not beneficial, the lock will be taken straight away. In comparison to branch predictors (which are pretty well understood, but continue to evolve), two significant differences remain for lock elision predictors: (1) what exactly is the "identity" of a critical section, and (2) how to update the predictor.

The identity used to index into the predictor should capture a high-level abstraction of the nature of a transaction. In semi-transparent lock elision, the predictor can be a software component consulted in the implementation of the elidable locking primitive. As such we found that indexing the predictor based on the address of the lock provides good results. Indexing based on the instruction address might also be useful, because it differentiates between different usages of the same lock variable, but relies on inlined locking primitives. Alternatives include: using parts of the lock acquisition function's return stack (GCC exposes a `__builtin_return_address` function to such effect), or supplying an application-specified ID value.

Transparent Hardware Lock Elision In contrast to the integrated and semi-transparent approaches, fully transparent hardware lock elision [58, 62] integrates all required elision functionality into the CPU. A full-blown, aggressive implementation could then elide all critical sections, irrespective of whether they come from a (inlined) library, the OS kernel, or are built manually. In their simplest form, locks are built out of an instruction sequence to acquire and release the lock; with the acquire path usually containing one atomic read-modify-write instruction (or load-linked / store-conditional sequence to that effect) that tries to transition the lock from a FREE to a TAKEN state, and a check of successful acquisition. The release operation then will perform the inverse by storing a FREE value back in the lock variable.

Transparent elision will then perform the following: (1) detect an atomic RMW instruction transitioning a lock; (2) instead of acquiring the lock, start a transaction; (3) check that the lock is free, and (4) acquire the lock *locally*. Finally, when a store operation is detected that transitions the lock from the local taken state back to the original free value, the local write operations (FREE to TAKEN, TAKEN to FREE) are discarded and the transaction committed.

Due to the local execution of the lock acquire / release operations, multiple concurrent threads will see the lock as being free, transition it locally to taken and perform the elided critical section concurrently. If there are no true data conflicts between the critical sections, they can execute concurrently.

Similar to the integrated approaches describe earlier, predicting whether a detected critical section can be elided successfully is crucial for this approach; in transparent HLE, however, another (logical) layer is required, too: is a particular atomic RMW sequence actually a lock acquisition. From my experience, reliably detecting and eliding critical sections is challenging because:

- Different instruction sequences being used to acquire the lock
- Similar instruction sequences being used for other purposes, such as lock-free data structures, reference counting, and as a fence replacement

- Various instruction sequences used to free the lock variable
- Lock algorithms transitioning the lock from a FREE1, to TAKEN1 and then FREE2 state, making it hard to detect the transaction end, and impossible to remove the costly lock variable modification through elision; for example ticket locks
- Lock algorithms that have other complex behaviours such as MCS locks, which spin locally if the lock was acquired
- Modifications of the lock variable (or neighbouring memory locations) during the lock elision; for example transitioning the lock from a TAKEN1 to a TAKEN2 value during the critical section; for example in the 32bit x86 Java HotSpot version
- Wrongly detected lock acquisitions cause nesting of elision, and missed / non-existing release operations will then continue the elision until the transaction mechanism runs out of capacity, or aborts otherwise

For these reasons, I believe that fully transparent lock elision is not feasible, or can only have very narrow coverage. Unpublished studies that I performed at AMD analysing instruction traces of a huge set of relevant workloads for lock / unlock patterns confirmed that.

Intel's TSX proposal offers lock elision [303, 367], but side-steps (and arguably implicitly acknowledges) the issue of detecting proper lock acquisitions / releases over other spurious memory modifications: TSX requires annotations of the memory instructions that constitute the actual lock acquisition and release with `xacquire` and `xrelease` instruction prefixes. While these prefixes require software modification, the resulting code is still backwards compatible with older, elision-unaware CPUs, because the prefixes are selected such that they are ignored / meaningless in non-TSX ISAs (they map to the `repe` / `repne` prefixes that are only meaningful with string instructions).

5.2.3 Summary: Challenges of Lock Elision

In summary, lock elision in either form may be a viable way to apply transactional memory to accelerate execution of threaded applications; but several challenges remain. Lock elision reuses code inside critical sections and places it inside transactions. Supporting transactional execution as a default for all memory accesses (instead of explicit annotation) eases this transplantation. In addition, the transactional memory mechanism should abort upon encounter of any construct that it cannot support cleanly inside a transaction.

These two conditions and the linearisability of transactions ensure *correctness*: the elided code does not observe behavioural differences between elision and actually acquiring the lock variable[3]. The resulting construct does not, however, give any performance guarantees. With imprecise, incorrect detection of critical sections, and frequent aborts due to spurious contention, or unsupported instructions; the performance of the elided critical sections may be worse than serialising on the original lock.

Proper prediction in hardware or software is therefore important to ensure that unsuitable critical sections are detected and their elision is not attempted regularly.

Finally, lock elision is unable to cope with data conflicts that would be masked by the serialisation and may not be obvious to the programmer. False sharing is particularly bad, examples often seen are: the data structure root node being allocated in the same cache line as other data that is frequently written to; `malloc` meta-data in the same cache line as a previous data element and being modified by a new allocation; and reference counting of objects that are being traversed / read.

[3]Except specific behaviours that may rely on ordering induced by the RMW operations on the lock variable[82].

5.3 Evaluation Experiments

Together with project collaborators, I have evaluated the performance characteristics of the proposed ASF BeHTM ISA extensions and their usability in various scenarios. The analysis focusses on accelerating concurrent data structures (RB-trees, linked lists, skip lists), larger transactional memory applications (STAMP), and semi-transparent lock elision (memcached). Further work using ASF has been performed on eliding the global interpreter lock in Python and eliding locks (from monitors and `synchronized` methods) in the HotSpot Java VM. The results of the two latter works could not be published due to unfortunate timing of AMD liquidating the AMD OSRC Dresden office at the end of 2012. The source code modifications of the HotSpot JVM are, however, available [290] and anecdotal evidence of design decisions and complexities has been incorporated into this thesis.

The evaluation appears in the following publications which are available in proceedings, or attached to this thesis:

- ASF: AMD64 Extension for Lock-free Data Structures and Transactional Memory (MICRO 2010 [214])
- Evaluation of AMD's Advanced Synchronization Facility Within a Complete Transactional Memory Stack (EuroSys 2010 [213])
- From Lightweight Hardware Transactional Memory to Lightweight Lock Elision (TRANSACT 2011 [254], Appendix B.4)
- Between All and Nothing–Versatile Aborts in Hardware Transactional Memory (TRANSACT 2015 [337], Appendix B.7)

Other publications from collaborators continue to evaluate ASF's performance and unique feature set:

- Optimizing Hybrid Transactional Memory: The Importance of Nonspeculative Operations (SPAA 2011 [255]),
- A Scalable Lock-Free Universal Construction with Best Effort Transactional Hardware (DISC 2010 [222]),
- Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory (ASPLOS 2011 [244]),
- TM-dietlibc: A TM-aware Real-world System Library (IPDPS 2013 [299])

5.3.1 Integer Set Data Structures

Integer sets are widely used as a small micro-benchmark for evaluating transactional memory performance and scalability. Due to their small size, and control over algorithms, update rates, data sizes, and concurrency, they are ideal to understand BeHTM implementations in simulation.

Figure 5.2 shows the performance of a typical set of integer set implementations: liked lists, red-black trees, skip lists, and hash sets. Four different ASF implementations are tested: small vs large dedicated, fully-associative buffers for tracking the working set ("LLB 8" vs "LLB 256"), and the optional use of the set associative L1 data cache for tracking reads ("w/L1").

Comparing different data structures, it is clear that the linked list has limited scalability due to the linear access pattern causing aborts between updating and traversing transactions. The other data structures show much better scalability because update and traversal regions conflict much less frequently. The linear scan in linked lists does not only cause poor scalability, but also high resource requirements, as O(N) entries need to be tracked in the working set.

The tiny ASF implementation that uses only a dedicated buffer with eight entries ("LLB-8") does not provide enough capacity for any of the data structures requiring traversal. The hash set, however, does not require any traversal (unless buckets overflow) and thus the small implementation does offer similar performance to the bigger designs.

The skip list exposes conflicts in the set associative data cache when comparing performance of "LLB-256" and "LLB-256 w/ L1". In the latter, the reads are tracked in the L1 data cache; due to memory layout effects, the tracking capacity can be significantly smaller than the 256 entries offered by the LLB (the associativity in the worst case); and patterns with constant large offsets between accesses are known to sensitise such behaviour.

Overall, ASF provides good scalability, algorithm and tracking capacity permitting.

Size Sensitivity Figure 5.3 explores the sizes of data structures and hardware mechanisms in more detail for the linked list and the red-black tree. As expected, the linked list performance falls of a cliff when it exceeds the capacity of the tracking structure (going from six to 14 elements for "LLB-8"), and generally deteriorates when the list gets longer, as longer lists have longer traversal phases with higher chances for conflict.

For the red-black tree, the situation is very different: larger trees actually increase the chances of concurrent threads traversing different paths and thus performance increases until the tree starts to hit the limitations of the tracking structures.

Early Release Clearly, the linked list is not a great transactional data structure; with additional hardware support, however, it can outperform other, more complex data structures: Figure 5.4 shows the impact of the `RELEASE` instruction. Thanks to the simple algorithm, the linked list can use `RELEASE` to release the already traversed part of the list; emulating hand-over-hand locking with transactions. As a result, performance is very high, and independent of size; only when the list becomes too big to fit into the L1 cache, performance suffers due to the requirement of traversing every element on the path.

Overheads of HTM and STM Because of the small size of the workloads and the use of simulation, it is possible to annotate separate phases of the transactions (start / commit, application code, transactional accesses) and compare the performance of STM and HTM. Figure 5.5 visualises the cycle break-down for single-threaded runs. Clearly, TinySTM is significantly slower than ASF; most of the different comes from the cost of the transactional memory accesses. For linked list, STM needs 30.1x the number of cycles for transactional accesses, for skip-list 15.4x, for the red-black tree 32.7x, and for the hash set 9.4x. Generally, the overhead is higher for accesses which would have executed in a tight scan loop with a predictable access pattern. That is the reason why the overhead for hash set is the lowest: hash sets are accessed not in a streaming fashion, and require only few accesses in general, and those can overlap with the STM orec hash table access.

Maximum supported data structure size While hash tables generally perform well under transactional execution, they can also deteriorate; effectively turning every hash bin into a linked list. In Figure 5.6, we explored the maximum supported size of an oversubscribed hash table with different ASF implementations. With more associativity ("ASF-C8" vs "ASF-C4"), the buckets can be filled significantly more; allowing more elements to be crammed into a hash table before having to perf costly resize operations. Another implementation ("ASF-LC") uses a small, fully associative *overflow* structure in hardware (the load-store queue) to catch those few accesses that cause cache conflicts and would limit the size of transactions. Using the fully associative structure as a fall-back further increases the supported load factors and overall size of the hash table. Compared to the earlier "LLB8 w/ L1" design which used the fully associative LLB only for writes, the fall-back aspect (for both loads and stores) makes a crucial difference.

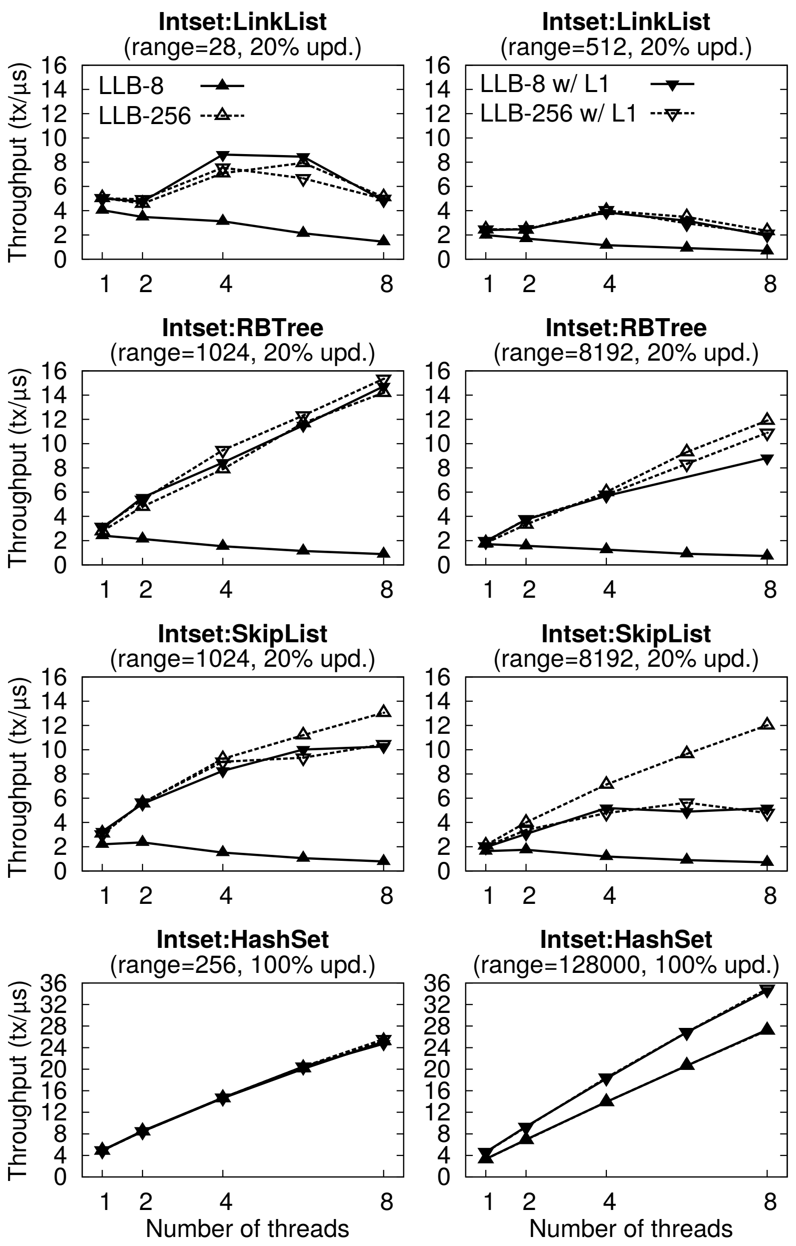


Figure 5.2: Scalability of IntegerSet with linked list, skip list, red-black tree, and hash set, with four ASF implementations and varying thread count and key range (throughput; higher is better). From [213].

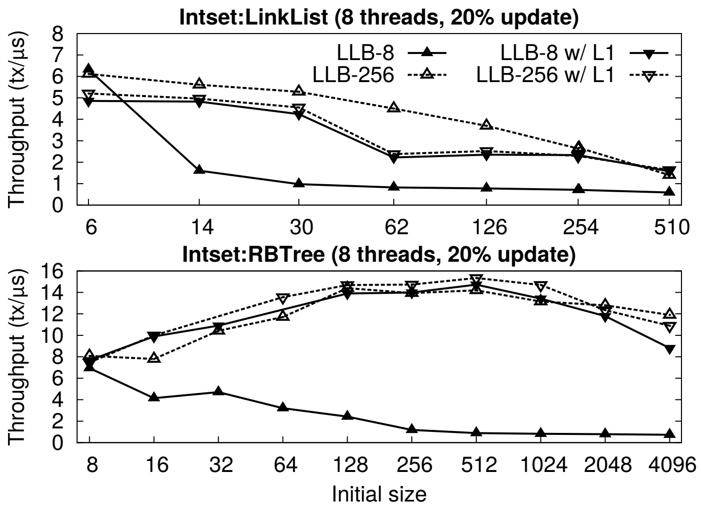


Figure 5.3: Influence of ASF capacity on throughput for different ASF variants (red-black tree and linked list with 20% update rate with eight threads). From [213].

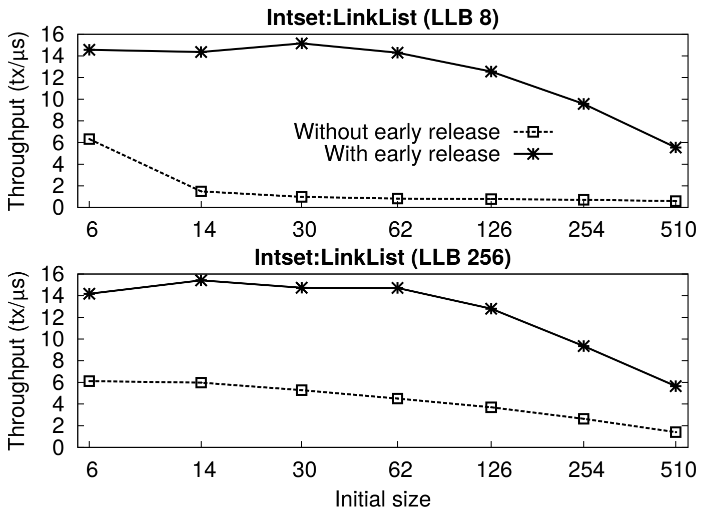


Figure 5.4: Early release impact: throughput amelioration with linked list (20% update rate, eight threads). From [213].

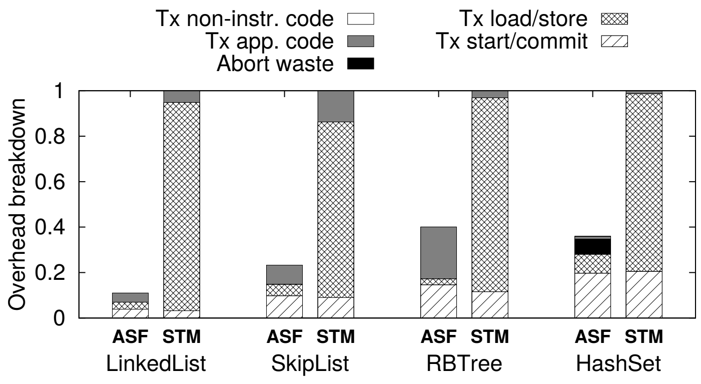


Figure 5.5: Single-thread overhead details for ASF-TM (with LLB-256) and TinySTM. All values normalized to the STM results of the respective benchmark. From [213].

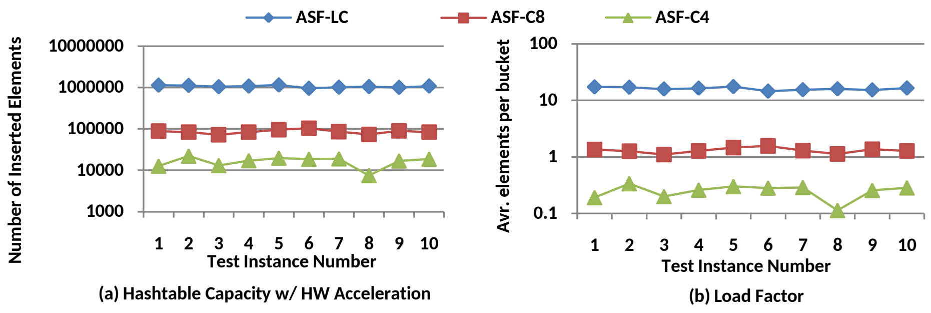


Figure 5.6: Comparison of ASF-LC, ASF-C8, and ASF-C4 for performance robustness of lock-free data structures. From [214].

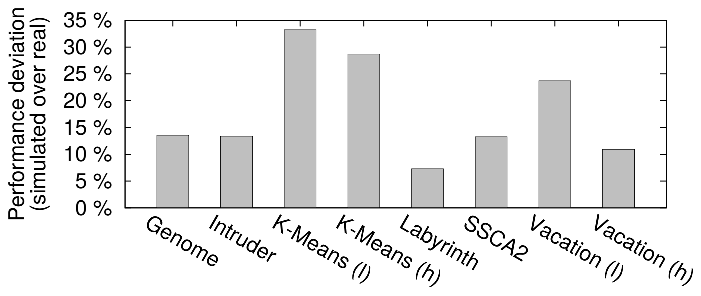


Figure 5.7: PTLsim accuracy for the runtime of the STAMP benchmarks (no TM, no ASF, one thread) for simulated with respect to native execution. From [213].

5.3.2 STAMP: complex workloads

In addition to micro-benchmarks, the STAMP benchmark suite offers more complex workloads for transactional memory. Before investigating the performance of HTM, I evaluated that the simulator produces reasonable performance estimates for the workloads by comparing simulated to native performance. Figure 5.7 shows the that the single-thread performance error is around 30% or less, with most workloads having errors of 15% or less.

Vacation and K-Means seem to exercise mechanisms in the microarchitecture that perform differently in PTLsim-ASF and in our selected native machine. Clearly, PTLsim cannot model all of the performance relevant microarchitectural subtleties present in native cores, because many of them are not public, highly specific to the revision of the microprocessor, and difficult to reproduce and identify.

One source of the inaccuracies we observed might be a PTLsim quirk: although PTLsim carefully models a TLB and the logic for page-table walks, it only consults them for loads. Stores do not query the TLB and therefore are not delayed by TLB misses, do not update TLB entries, and are not stalled by bandwidth limitations in the page-table walker. The effect on accuracy likely is minor since translations for many stores already reside in the TLB because of a prior load. Nonetheless, we will add a better simulation of stores in a future release of PTLsim-ASF. Despite these differences, we think that PTLsim models a realistic microarchitecture and captures several novel interactions in current microprocessors. For our main evaluation we conduct all experiments–including the baseline STM runs–inside the simulator to make sure that our results are not affected by the discrepancies.

DTMC and PTLsim evaluation In a first series of evaluations, we used the DTMC compiler, TinySTM-ASF, and the detailed PTLsim-ASF simulator to run the HTM[4] versions of STAMP workloads.

Figure 5.8 presents scalability results for selected applications from the STAMP benchmark suite. We also compare the performance of ASF-based TM to the performance of finely tuned STM (TinySTM) and to

[4]All accesses inside the transactions are annotated.

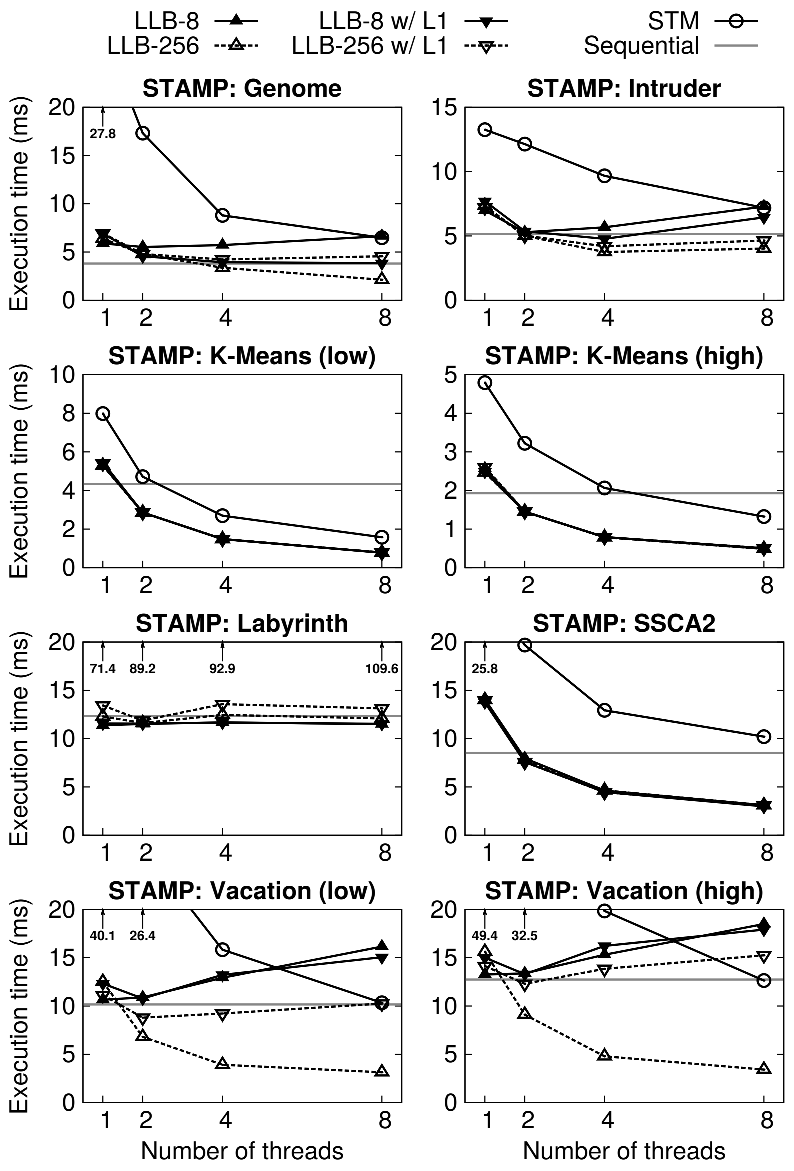


Figure 5.8: Scalability of applications, with four ASF implementations and varying thread count (execution time; lower is better). The arrows indicate STM values that did not fit into the diagram. The horizontal bars show the execution time for execution of sequential code (without a TM). From [213].

serial execution of sequential code (without a TM). We observe that ASF-based TMs show very good scalability and much better performance than STM for some applications, notably genome, intruder, ssca2, and vacation. Other applications such as labyrinth do not scale well with LLB-8 and LLB-256 because the TM uses irrevocable mode, which serialises execution, extensively, yet performance is still significantly better than STM. Interestingly, the applications that do not scale well are those with transactions that have large read and write sets (according to Table III in [166]). For applications with little contention and short transactions, all four ASF variants perform well. For other applications, LLB-256 usually outperforms the other implementation variants because LLB-8 suffers from the transaction lengths and L1/LLB is susceptible to cache-associativity limitations. Yet, it is interesting to note, even the LLB-8-based implementation provides benefits for many applications.

Figure 5.9 provides a breakdown of the abort reasons in the STAMP applications with different ASF implementations. Unsurprisingly, the implementation with the small dedicated buffer (eight-entry LLB) suffers from many capacity aborts for most benchmarks, while the larger dedicated buffer (256-entry

LLB) usually has the least capacity aborts. Adding the L1 cache for tracking transactional reads ("+L1") does not always reduce capacity aborts, but actually increases them for several benchmarks. Three reasons contribute to the increase. First, although the L1 cache has a large total capacity, it has limited associativity (two-way set associative) and therefore usable capacity is dependent on address layout. Second, our current read-set tracking implementation does not modify the cache-line displacement logic. Non-tx accesses may displace cache lines used for tracking the read set. Finally, cache lines may be brought into the cache out of order and purely due to speculation of the core. These additional cache lines may further displace lines that track the transaction's read set. Since displacement of cache lines with transactional data causes capacity aborts, the large number of those is not only caused by actual capacity overflows, but may be caused by disadvantageous transient core behavior. For our current study, we fall back to serial mode to handle capacity aborts, therefore reducing contention aborts for benchmarks with high capacity failures. To leverage the partially transient nature of capacity aborts, one could also retry aborting transactions in ASF and hope for favorable behavior. Furthermore, we will tackle the issue from the hardware side by containing the random effects and ensuring that we meet the architectural minimum capacity. Both aspects are subject of current research.

The power of selective annotation In a follow-on study, we evaluated more advanced design points for the HTM (previously mentioned usage of the load-store-queue for providing additional worst-case capacity for loads and stores and using the cache for both loads and stores) in a simpler AMD-internal simulator. In addition to a different hardware design, we also investigated the difference between full annotation (STAMP's HTM mode), and much reduced hand annotation (STM mode) with ASF to evaluate the usage of ASF's elective annotation. In Figure 5.10 we show performance (as throughput) for the STAMP workloads. Based on their scalability, there are three groups of behaviour.

For the first group (bayes, labyrinth, and yada), the HTM version of STAMP ("ASF-AS") scales poorly in comparison to ASF-LC/C8/C4. This is because ASF-AS does not support selective annotation which reduces the speculative memory footprints of speculative regions. On closer inspection we find that the average ratio of speculative accesses to total memory accesses is only 8.25% while the rates for bayes, labyrinth, and yada are even lower (0.03%, 0.01%, and 1.12%). Fewer non-tx accesses cause less resource overflows and less contention: while about 89% of all transactions fit into the provided resources, the remaining transactions serialise and are often not just big, but also long-lived and prevent ASF-AS from scaling.

Bayes and yada perform better with ASF-LC and ASF-C8 than ASF-C4 since many capacity overflows due to set-associative conflict misses are eliminated by the higher cache set-associativity of ASF-C8 and the fully associative load/store queues of ASF-LC.

Applications of the second group (indtruder, kmeans, ssca2) perform very similarly on all designs; their transactions are only short-lived and trigger few capacity overflows (0.12%, i0.01%, and 0.03% respectively). Therefore, they do not require selective annotation, a high cache set-associativity, or the load/store queues. Comparing that to our earlier analysis, intruder is the only difference, here; it stopped scaling with four cores on PTLsim with a high number of transactional conflicts as abort reasons.

Finally, genome and vacation in the third group show some performance differences among the four design options. Vacation scales better with ASF-LC and ASF-C8 due to the reduction of set-associative conflict misses, which is inline with it's erratic behaviour for any set-associative implementation in our earlier analysis (with the fully-associative LLB-256 being the only scaling alternative).

Investigating applications of the first and third group in more detail, we find that the execution cycles spent on overflowing and serialised ("virtualization") transactions are removed with selective annotation and higher associativity caches / tracking structures; in Figure 5.11

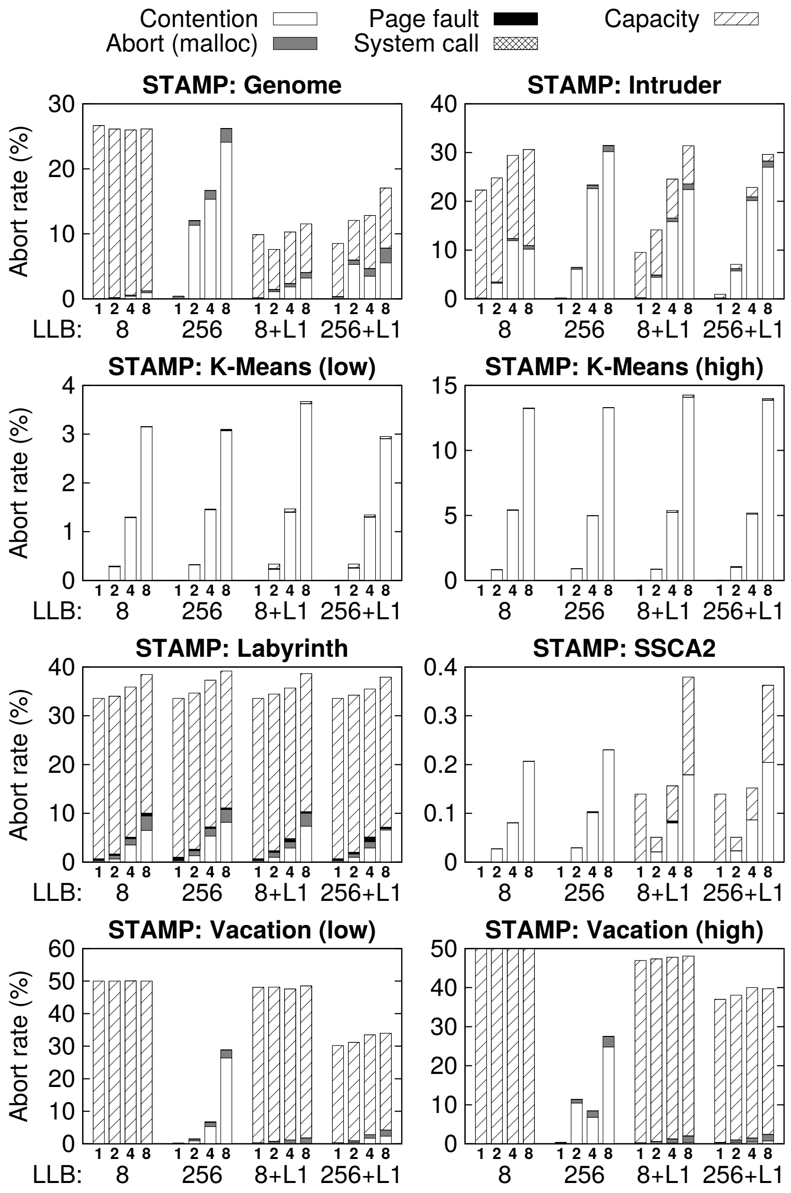


Figure 5.9: Abort rates of applications, with four ASF implementations and varying thread count. The different patterns identify the cause of aborts. From [213].

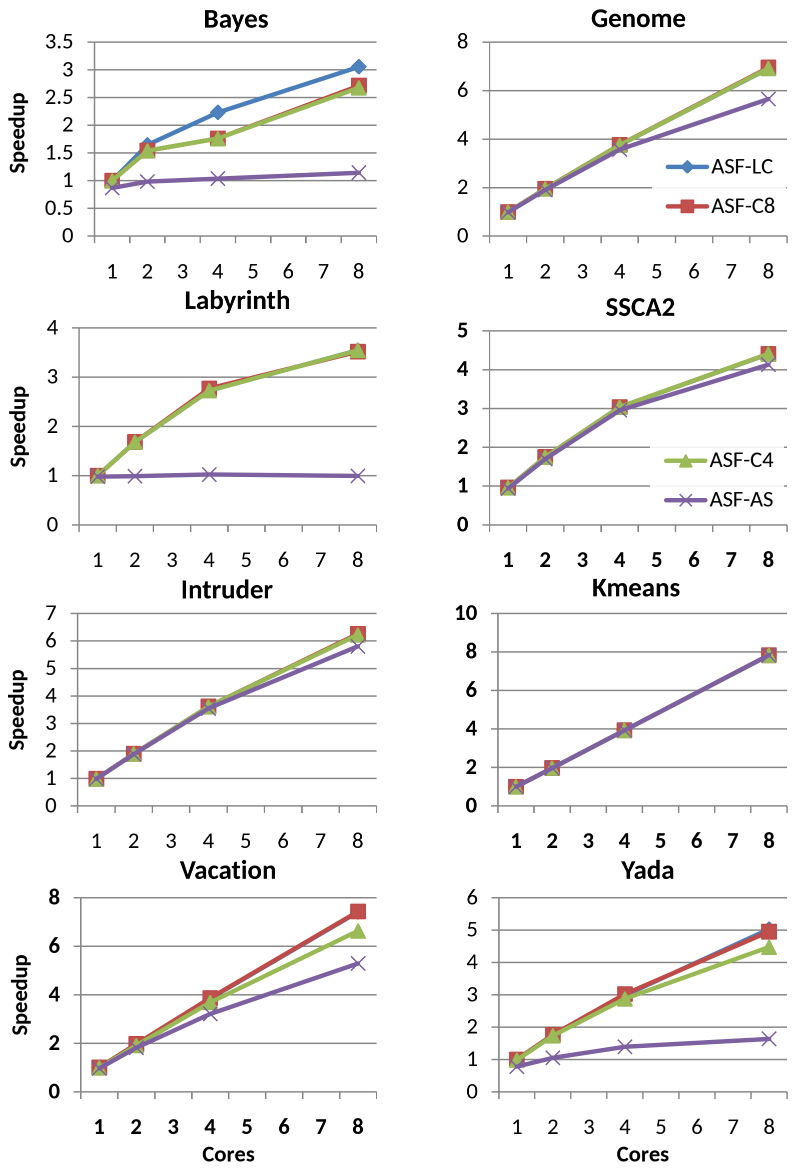


Figure 5.10: Scalability of ASF design options. The execution time of each application is normalized to the sequential execution time of the application without transactions. From [214].

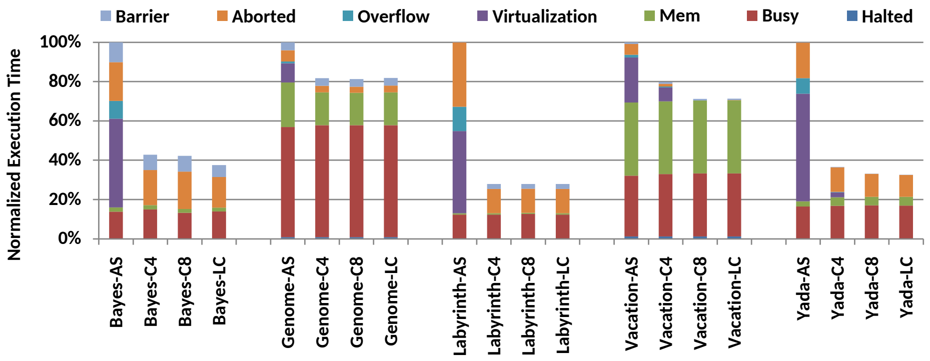


Figure 5.11: Execution time breakdown of the STAMP applications running with eight cores. Results are normalized to the execution time of ASF-AS. From [214].

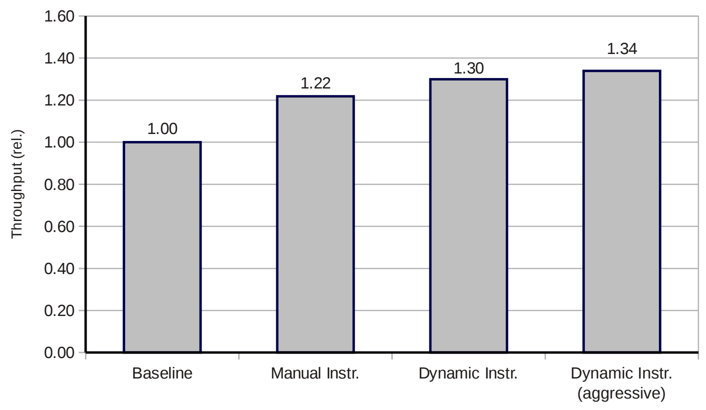


Figure 5.12: Performance of various lock elision mechanisms in memcached compared to the native performance with four application threads and cores. From [254].

5.3.3 Memcached: elision and transaction characteristics

The final analysis shows our semi-transparent approach to lock elision with a new application: the in-memory database memcached. By replacing `pthread` locking code dynamically (through `LD_PRELOAD`) with ASF transactions (using `SPECULATE_INV`), we can evaluate performance and other characteristics of elided critical sections in the application without any changes to source or binary. We simulate the application in PTLsim-ASF, and find that with four cores, eliding the critical section protecting the hash map at the heart of memcached, we get performance improvements of up to 34% (Figure 5.12).

Our instrumentation approach outperforms manual instrumentation thanks to larger coverage (more code paths instrumented), and careful dynamic selection of elision targets (more detail in the paper) preventing ineffective elision after a short learning period. Improving coverage is important, as a single taken lock has a high cost, as it will serialise all execution by aborting concurrent eliding transactions.

For memcached, we investigated the transaction characteristics in more detail. Figure 5.13 shows histograms and CDFs for the number of cache lines accessed and cycles spent in transactions. In memcached, transactions are small and they read about three times as much (median 14 cache lines, maximum 25 cache lines) as they write (median four, maximum six cache lines). Furthermore, transactions execute quickly (up to 200 cycles) and their execution times depend largely on the cache hit / miss behaviour; so overall durations are varied and clustered.

Similarly, the instruction count and breakdown in Figure 5.14 show similar trends. The dynamic number of instructions executed per transaction is small (less than 80 instructions), and the clustering reflects few distinct execution paths through the code; which results in similar clusters for the numbers of load / store instructions. Comparing memory instructions and cache lines used, stores cluster into fewer cache lines than loads; likely due to bundled writes and lookup accesses for loads.

5.3.4 Evaluation Summary

In summary, we have shown that ASF accelerates applications with transactions and elided transactions efficiently. Thanks to its low single-threaded overheads (especially when accessing transactional memory locations), it outperforms state-of-the-art STMs, such as TinySTM. ASF handles many benchmarks well; yet some transactions exceed the provided tracking capacity or exhibit unfavourable access patterns. Overall, despite its best-effort nature, ASF outperforms STMs by an order of magnitude and provides significant improvements over single-threaded execution already with two threads for the applications tested.

Comparing the different implementation variants, the associativity of L1-based implementations may

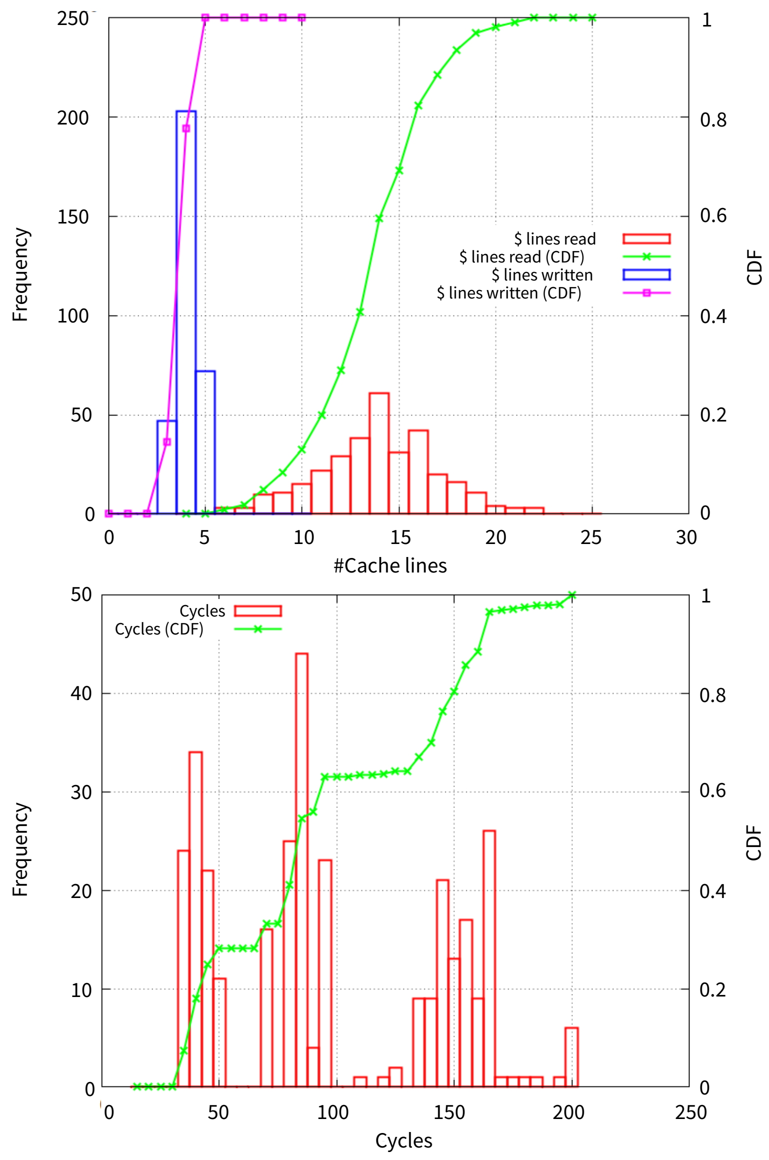
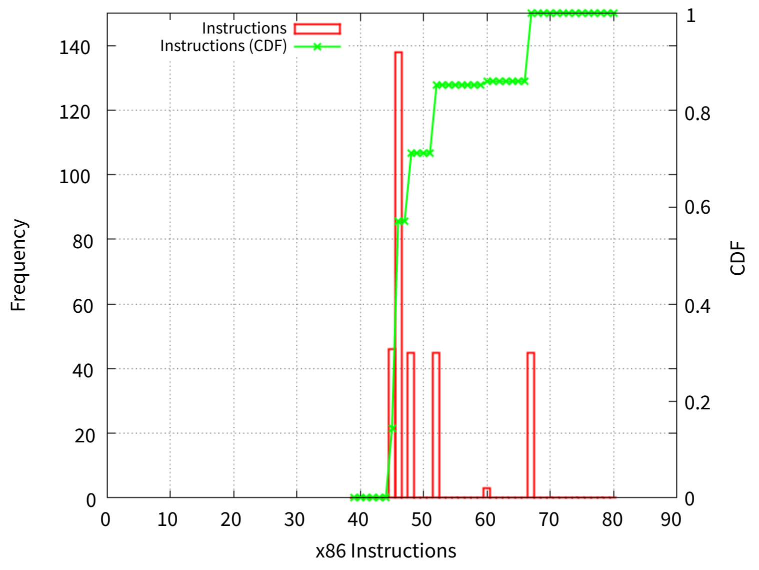


Figure 5.13: Transaction characteristics (histograms and CDFs) of elided critical sections in memcached. Top: number of read / written cache lines; bottom: cycles spent in the transaction. From [254].



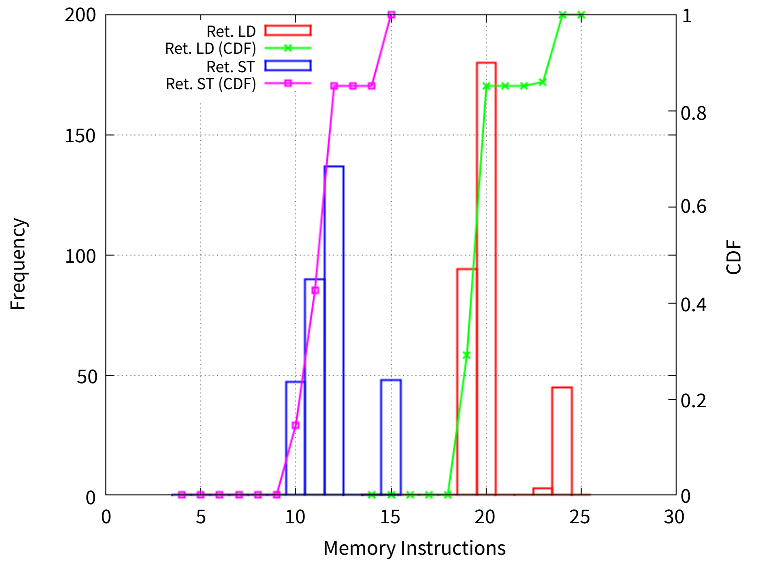


Figure 5.14: Instruction breakdown (histograms and CDFs) of elided critical sections in memcached. Top: number of dynamic instructions; bottom: number of load / store instructions. From [254].

be too limiting some address layouts, while the capacity of fully-associative buffers can be too limiting. One of our designs combines a large associative structure (the L1 cache) with an additional fully-associative overflow mechanism (the load/store queue here); this combination provides fewer performance "cliffs" at reasonable additional cost. During our experiments, we tuned memory allocation (and some data structures) to avoid false conflicts where independent, conflicting data happens to be collocated in the same cache line.

Finally, the simulation platform has been extremely helpful in understanding the performance characteristics and executing applications realistically has allowed us to make great progress in developing the transactional memory software stack. Conversely, the rich software stack enabled more complex applications that provided richer feedback for the hardware design and more than once exposed design and implementation bugs in the HTM primitive. In the next section, I will highlight the simulator implementations that I created to evaluate ASF.

5.4 Simulator Implementation Details

For our analysis of both architectural and micro-architectural behaviour of ASF, I extended two popular, cycle-level (cycle-approximate) simulators: PTLsim [135] and the derived Marss86 [253]. I chose these simulators for their support of running multi-core simulations of full-system software stacks with a modern ISA (x86 / AMD64). In addition, the simulators support fast emulation to the region of interest through running the system-under-simulation in extended versions of the Xen hypervisor [72] (PTLsim), and QEMU [81] (Marss86).

In addition to these two cycle-level simulators, I have implemented architectural emulation of ASF inside AMD's SimNow [260] fast emulation platform, and also by Martin Pohlack in Intel's PIN tool [92]. Both tools were created while I was working at AMD and have not been cleared for public release.

The advantage of these ISA-level tools is the simplicity of the transactional memory implementations: once the new instructions for ASF are decoded, their implementation is usually straightforward, due to the simple access to architectural state, strict serial execution of the instruction stream (no OOO-speculation, or ILP), and much higher level of abstraction (no explicit caches, no coherency protocol, no interconnect, etc.). In comparison to STM / HyTM libraries, the implementation of the transactional functions can be simple, because the emulation systems themselves cause significant slowdown. Overall throughput is still good enough for experimentation and several orders of magnitudes faster than cycle-level execution.

The simple architectural implementations (consisting only of a few hundreds lines of code each) therefore will only (1) store a copy of the instruction pointer and stack at transaction start, (2) perform conflict detection and data versioning in simple (thread-safe) hash-tables / hash-maps, and (3) perform abort according to conflicts, interrupts and exceptions in accordance with the ASF ISA specification.

The difference between the SimNow and PIN implementations are scope: SimNow models a full-system stack, including OS code, interrupts, page-faults, etc., so the emulation can capture these interactions accordingly and, for example, can abort a transaction when a contained memory access exhibits a page fault. Furthermore, tracking uses physical addresses due to the availability of page-tables. The main downside to SimNow is the forced serialisation between multiple emulated CPU cores. These execute in a round-robin fashion, which greatly simplifies both the internal data structures of the tool, and also the ASF implementation, because the instrumentation does not have to be thread-safe.

In PIN, on the other hand, the full-system perspective is lost, due to the user-level only nature of the instrumentation. This in turn allows quick experimentation (no need to boot another operating system), but also hides interactions from the ASF implementation (invisible page-faults and interrupts). PIN is, however, a concurrent DBT framework and therefore the instrumentation of especially the transactional memory instructions need to be thread-safe. For our simple prototype, we grab a global lock in the instrumentation before the actual instruction, add the conflict detection and data versioning logic, execute the actual memory instruction, and then free the global lock.

In both SimNow and PIN, the introduced serialisation greatly simplifies the logic and can use simple C++ hash-maps for tracking both conflicts and perform versioning.

The remainder of this section will focus on cycle-level simulators, because they are more complex and require more intricate work for faithful implementation of ASF. The full source code modifications to both the PTLsim and Marss86 simulators are available publicly [262, 304], so the discussion here will focus on high-level interactions and complications, rather than a full source-code level walk through.

5.4.1 Characteristics of Cycle-Level Simulator Environments

Cycle -level simulators for modern multi-core CPUs can be distinguished along the following general execution characteristics:

- trace-driven vs execution-driven
- clocked simulators vs discrete event simulators
- accelerated simulation availability

In addition to differences in the general simulation methodology, the simulated models may differ. The main components we are looking at are the CPU and the interconnect / memory hierarchy. CPU core models range from non-timing models, fixed-IPC models over pipelined in-order core models to very detailed out-of-order core models. Memory systems generally vary along the number of supported agents (single-core vs multi-core), coherency protocol (fixed vs variable), and topology (fixed vs flexible). Simulation fidelity in memory hierarchies differs with respect to modelling both latencies (fixed vs actual queueing) and bandwidths (fixed vs limited bandwidth and back-pressure). Finally, one significant importance for memory hierarchies is whether they actually send and store data in memory requests and caches, or whether those structures are used exclusively for modelling placement, timing and bandwidth of requests.

In this work, I have extended the PTLsim and Marss86 simulators [135, 253], with the latter derived from the former. They both feature a very detailed out-of-order core model with pipeline stages, reservation stations, replay, speculation, etc., faithfully modelling a modern out-of-order core microprocessor.

They also support seamless switching between fast emulation (Xen / QEMU) and detailed simulation. In addition to full-system simulation, both offer user-level only simulation, but due to limitations (no OS involvement, paging, interrupts, and deficits in multi-threading), I have not made extensive use of this feature, but instead focus on full-system execution.

Simulation is driven by time, that means every component of the simulator is clocked on every cycle and consults its data structures to see if waiting entries are ready to progress to the next stage of execution, need sending on the bus etc.

Both simulators decode both user-level and privileged instructions of the AMD64 instruction set and break them up into micro-ops that will flow through the pipeline. The various pipeline stages can process multiple instructions per cycle (super-scalar) and issue / execution of μops can start / complete out-of-order. A reorder buffer (ROB) keeps track of in-flight instructions; and registers are handled through a standard renaming table.

The core model consists of a variety of functional units with easily configurable execution capacity. Mops can be specified to require specific functional units for a specific number of cycles, and communication latencies for the bypass networks between functional units can be customised, too.

Memory instructions flow through a load-store-queue and can also be executed out-of-order aggressively. The memory hierarchy will then dictate how many cycles a load / store requires.

Aside from general infrastructure differences between PTLsim and Marss86 (Xen vs QEMU), the memory hierarchy is the main difference between the two simulators. PTLsim consists of a single cache hierarchy separate per core, simulating L1i, L1d, L2 and L3 caches (physically indexed and tagged, inclusive strictly per-core. These are connected to a fixed latency DRAM. By default, caches were not coherent, i.e., entries could be present in multiple caches in conflicting states. Marss86 heavily extended this memory model through earlier work on MPTLsim which appeared in [208]. This improved cache model adds the following new features to the already very detailed core model from PTLsim: proper bandwidth modelling of caches and interconnects, coherency messages which ensure coherence and model delays due to interactions in the coherency protocol, multiple coherency protocols (I used the MOESI configuration), directories to keep track of cache state in specific core-local caches, and large shared caches.

Both memory subsystems rely on a single flat physical backing store to simplify the correctness of the coherency protocol. This means that in addition to the timing interactions (querying caches, sending out requests, etc.), loads and stores will access the single shared flat memory to read / write the data in question. This will ensure that for every address there is always only a single value that corresponds to the most recently written version of this address, irrespective of the coherence protocol actually implemented in the timing layer.

This simplification guarantees basic coherence properties basically for free; greatly reduces burden on the coherence protocol implementation and the verification. It does, however, also lead to unfortunate side-effects in specific timing conditions that cause very unrealistic simulation of tight intra-thread communication. I will explore his is issue further in Section 5.4.4.

In earlier work, I configured and tweaked the core and (single-core) memory hierarchy to produce timing / performance results similar to an AMD K8 CPU [158, 159]. For this thesis I updated the adaptations so that the core matches an AMD Family 10h processor core (AMD Phenom / Phenom II).

Detailed Load Path Walkthrough The load path of *PTLsim* is relatively straight-forward, and its key code pieces are depicted in Figure 5.15. When a load μop (represented as a `ROBEntry`–reorder-buffer entry) gets to the issue stage, like all instructions, it checks that all its input operands (used for address calculation) are available. It then computes the effective virtual address, and translates it to physical addresses. Immediately after, the simulator loads the data from the single global physical address view (with `loadphys`).

Only after that, the various timing conditions are checked (is there an L1 data cache bank conflict, are there any unresolved or overlapping earlier stores, are there any held bus locks). Eventually, the load queries the TLB for a present translation entry and subsequently probes the L1 data cache directly.

Depending on the hit / miss information, the load is put on the `issued` / `miss` list and is assigned the appropriate latency value. The global PTLsim `clock` function clocks all components and waiting misses decrement their remaining cycles. If they hit zero, a wake-up callback marks the destination register ready so that depending instructions can consume the value. While there is a separation between the core and the memory hierarchy, for simulator efficiency, L1 cache hits are treated through fast-path logic in the simulator. The slow-path of the memory hierarchy is shown in Figure 5.16 and details the rather explicit state machine and fixed layout of the cache hierarchy.

In the original PTLsim, multiple cores had completely independent cache hierarchies. Thanks to the data-less cache hierarchy, this was not a problem, as loads and stores all hit the same single global physical data store directly (and in sequence). For simulating the first-order effects of coherence, I added invalidations and cache-to-cache transfers by looking up in caches of the neighbouring hierarchies (in `probe_other_caches`) and invalidating their entries (in zero time).

```
void ROBEntry::issue() {
  fuinfo[opcode] & cluster.fu_mask & fu_avail
  /* load data from source registers */
  if (ld) issueload (..);
  if (asf) asf_pipeline_intercept.issue (..);
}

void ROBEntry::issueload(..) {
  physaddr = addrgen(..);
  data = loadphys(physaddr);
  asf_pipeline_intercept.issue_probe_and_merge(physaddr, data);
  /* Check store queue for earlier overlapping stores and fences */
  /* Perform forwarding if possible, replay otherwise */
  asf_pipeline_intercept.issue_load (..);
  /* Check L1 bank conflicts */
  /* Allocate entry on load queue */
  /* Check bus lock of others, acquire if necessary */
  /* Handle misaligned / small loads: merge two parts, shift data */
  caches.dtlb.probe(addr);
  probecache(addr);
}

void ROBEntry::probecache(..) {
  bool hit = caches.probe_cache_and_sfr(addr);
  if (hit) {
    cycles_left = LOADLAT;
    changestate(rob_issued_list);
    if (invalidating) caches.probe_other_caches(addr);
    return;
  }
  changestate(rob_cache_miss_list);
  caches.issueload_slowpath(addr);
  if (invalidating) caches.probe_other_caches(addr);
}

void OOOCore::dcache_wakeup(resp) {
  rob = resp.rob;
  rob->physreg->complete();
  rob->lsq->datavalid = 1;
}
```

Figure 5.15: Key core-side code used in the PTLsim load path.

Finally, the code excerpt shows the various integration hooks of the normal pipeline into the ASF pipeline extensions that I will describe in the next section.

In the PTLsim-derived simulator *Marss86*, the core-side of handling loads looks very similar. The key changes are: data is *not* loaded at issue time, but instead when the load completes; there are less alignment checks (to accelerate simulation); and the memory hierarchy is actuated through a data-driven request interface, including for L1 accesses. Figure 5.17 highlights the key source code constructs.

The memory hierarchy of Marss86 is a complete rewrite: it supports directories, a flexible cache structure, realistic timing for all coherence interactions, and pluggable coherence modules that work independently of the cache structures. All messages between components are represented as messages / requests, and the wake-up actions are chained together by timed call-back functions. Further, the simulated components are split into cache controllers, interconnects, and signals (serving as the timed call-backs). Surprisingly, there is no direct distinction between requests and responses, instead, the direction of travel of a request (from an upper or lower interconnect port) defines whether it is a request

```
void CacheHierarchy::issueload_slowpath(addr) {
  L1hit = L1.probe(addr);
  L2hit = L2.probe(addr);
  missbuf.initiate_miss(addr, L2hit);
}

MissBuffer::initiate_miss(addr, L2hit) {
  int idx = find(addr);
  if (idx >= 0) {
    /* Merge request with existing miss */
    return;
  }
  /* Handle full condition */
  /* Create new entry */
  Entry &mb = missbuf[..];
  bool L3hit = hierarchy.L3.probe(addr);
  if (L2hit || L3hit) {
    mb.cycles = ..;    //L2 / L3 LATENCY
    return;
  }
  if (probe_other_caches(addr))
    mb.cycles = CROSS_CACHE_LATENCY;
  else
    mb.cycles = MAIN_MEM_LATENCY;
  return;
}

MissBuffer::clock() {
  foreach (mb) {
    mb.cycles--;
    if (mb.cycles == 0) {
      /* Install in L1 / L2 / L3 */
      /* Switch to next state */
      if (mb.state == L1) {
        /* Set HTM bits */
        lfrq.wakeup(mb.entry);
      }
  }
}

LFRQ::wakeup(entry) {
  ready[entry] = true;
}

LFRQ::clock() {
  foreach(entry) {
    if (ready[entry])
      callback->dcache_wakeup(entry.req);
      ready[entry] = false;
      free[entry] = true;
  }
}
```

Figure 5.16: Simplified view of functions and flow used in the memory system of the PTLsim load path.

```
ROB::issueload(..) {
  /* as before ... */
  /* do NOT load data here */
  req = memHier->get_free_req();
  req->init(.., MEMORY_OP_READ);
  req->set_coreSignal(dcache_wakeup);
  bool L1hit = memHier->access_cache(req);
  if (L1hit)
    dcache_wakeup(req);
  else
    physreg->changestate(WAITING);
  return;
}

OOOCore::dcache_wakeup(req) {
  if ((request->get_type == ASF_ABORT) ||
      (request->get_type == ASF_COMMIT)) {
    asf_pipeline_intercept.cache_done();
  }
  ..
  rob  = req.rob;
  data = loadvirt(rob.lsq->virtaddr);
  asf_pipeline_intercept.load_binds_data(rob, &data);
  /* Check for store-to-load forwarding */
  rob.lsq->data = extract_bytes(data, ..);
  rob->physreg->complete();
  rob->lsq->datavalid = 1;
}

MemHier::access_cache(req) {
  ret = cpuController->access_fast_path(req);

  /* Success: L1 hit or write */
  if ((ret == 0) || (req->type == WRITE))
    return true;
  return false;
}
```

Figure 5.17: Core logic and memory system glue logic used in Marss86. Notice how the code is similar to that of PTLsim.

or a response. In addition, despite the elaborate modelling of timing of the cache tags, the caches still do not contain any data. While the pluggable coherence protocol module determines timing, the actual value coherence is provided again by a single global physical memory view. Figure 5.18 shows the high-level connection between the various interconnect and cache call-backs and how they are chained together depending on a hit / miss in specific caches. Note how the overall logic is distributed between the different caches; that way, caches can make decisions locally, and be connected in different topologies easily.

Detailed Store Path Walkthrough As opposed to loads, PTLsim and Marss86 handle stores at the commit stage of the pipeline; and their code is generally quite simple. Figure 5.19 shows the abbreviated code of *PTLsim*; and it is interesting to see that stores do actually not interact with the cache hierarchy, but instead they directly write to the global physical memory view. Similar to loads, I added simple first-order coherence (or rather, its timing effects) by snooping the caches of other hierarchies and invalidating their copy for local stores in zero time. Furthermore, the code excerpt shows the hooks for the ASF pipeline integration layer.

```
CacheController::handle_interconnect_cb(msg) {
  if (msg->sender == upperInter) {
    /* This is a request */
    /* Check space */
    /* Allocate pending request */
    deps = find_deps(msg->req);
    if (deps)
      /* Deal with dependencies */
    else
      cache_access_cp(..);
  } else {
    /* This is a response or snoop req / snoop resp */
    entry = find_match(msg);
    if (entry)
      if (msg.hasData)
        return complete_request(msg);
      else
        return handle_response(msg);
    /* Handle the remainder in the future */
    add_event(cache_access_cb, ..);
  }
}

CacheController::cache_access_cb(req) {
  if (req->isASF()) {
    /* Handle commit / abort messages */
    add_event(asfAbort, commitLatency, req); // or asfCommit
  }
  hit = probe(req);
  if (hit)
    signal = cache_hit_cb;
  else
    signal = cache_miss_cb;
  add_event(signal, cacheAccessLatency_, req);
}

Cachecontroller::cache_hit_cb(req) {
  if (req->isSnoop)
    coherence_logic_->handle_interconn_hit(req);
  else
     coherence_logic_->handle_local_hit(req);
}

Cachecontroller::cache_miss_cb(req) {
  if (req->isSnoop)
    coherence_logic_->handle_interconn_miss(req);
  else
     coherence_logic_->handle_local_miss(req);
}
```

Figure 5.18: Key parts of the redesigned memory system simulation framework in Marss86.

```
ROBEntry::commit() {
  /* Scan the ROB for all sub−uops of the current x86 instruction */
  /* Check for and handle exceptions */
  if (uop.opcode == OP_st) {
    lock = interlocks.probe(addr);
    if (lock)
    { /* Deal with bus locks */ }
     asf_pipeline_intercept .issue_probe_and_merge(addr);
  }
   if (uop.is_asf)
      asf_pipeline_intercept .commit(..);
   /* Update registers */
   if (uop.opcode = OP_st)
     caches.commitstore(lsq, addr);
   /* Cleanup */
}

CacheHierarchy::commitstore(lsq, addr) {
  storemask(addr, lsq.data, lsq.mask); // writes to backing store
  probe_other_caches(addr);
}
```

Figure 5.19: Path for stores becoming visible in memory in PTLsim.

In *Marss86*, the pipeline integration is again very similar; instead of writing to the backing store directly, however, the store is sent as a proper request to the memory hierarchy; in Figure 5.20. Subsequently, the core still updates the global physical memory view.

5.4.2 ASF Simulator Implementation

Conceptually, the core of the ASF implementation consists of three layers (see Figure 5.21 for structural linkage, and Figure 5.22 for an example flow): an *architectural layer* provides a per-core undo-log for transactional writes that also performs conflict detection when the architectural memory layer is read / updated; a *micro-architectural layer* that augments the existing caches, memory requests, etc., and provides a faithful implementation of the various implementation variants described in Chapter 4. The micro-architectural layer adds transactional read / write bits for cache lines and signals aborts to the core on conflicts and evictions of transactional lines (capacity aborts). Finally, an *ISA layer* that decodes ASF instructions n the AMD64 decoder and represents the abstract transaction state in the appropriate registers.

Architectural Tracking Layer For simplicity, the architectural layer also tracks transactional accesses on cache line granularity, which is assumed to be 64 bytes in all caches in both simulators, and is also fixed in the ASF specification (see Chapter 3, and the ASF specification in Appendix A). The architectural layer can abort transactions running on the local core when there is a remote access that is conflicting with a tracked entry; for that, it hooks into the `issueload` / `commitstore` functions just before they are actually performing the load / store. Figure 5.23 shows the C++ interface, and Figure 5.24 visualises the different flows for loads and stores intersecting with the architectural layer tracking logic.

Conflicts are detected as the first step for loads and stores by checking all other cores' undo-logs for entries with the same address. If such an entry is present, it will be restored and the transaction on that core flagged for abort. Transactional invariants guarantee that a transactionally written line is only ever in use by at most a single core.

Non-transactional memory accesses will then continue with their original flow and read / update

```
ROBEntry::commit() {
  ..
  if (uop.opcode == OP_st) {
    req = memHier->get_free_request();
    req->init(.., MEMORY_OP_WRITE);
    req->setCoreSignal(dcache_wakeup);
    memHier->access_cache(req);
    storemask_virt(lsq->virtaddr, lsq->data, ..); // writes to backing store
  }
  ..
}

OOOCore::dcache_wakeup(req) {
  ..
  if (request->get_type() == MEMORY_OP_WRITE)
  { /* Do nothing but logging */ }
  ..
}
```

Figure 5.20: In Marss86, stores behave similarly to PTLsim, but they are presented to the cache hierarchy, instead of bypassing it.

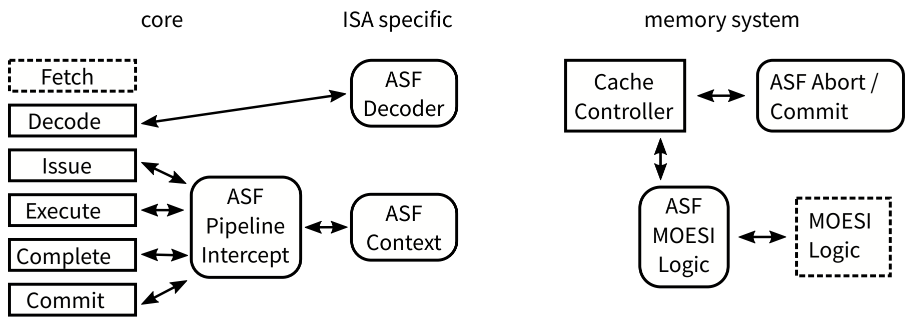


Figure 5.21: High-level graphical overview of the implementation of ASF in the Marss86 simulator.

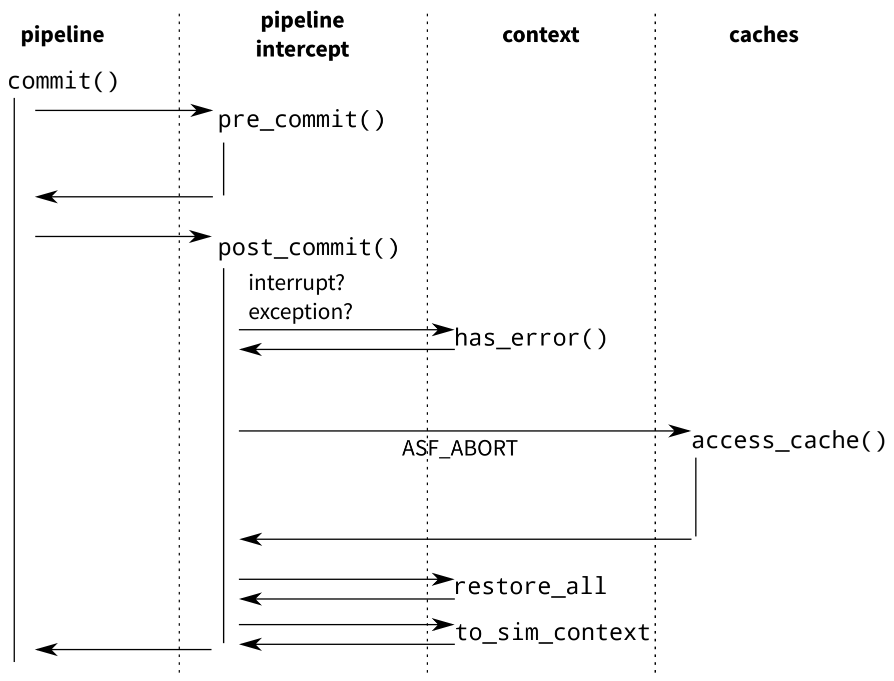


Figure 5.22: High-level interaction of the standard pipeline and ASF additions in case of an aborting transaction.

```
class ASFContext {
  public:
    enum ASFStatusCode {..};
    enum ASFType {ASF_NORMAL, ASF_INVERTED,
      ASF_ASYNC};

    bool to_sim_context(Context& c,...);
    void undo_to_sim_context(Context& c);

    /* Start / end transactions */
    void enter_spec_region(const Context& c,...);
    void enter_spec_inv_region(const Context& c);
    void enter_spec_async_region(const Context& c,
      Waddr store_virt, Waddr storage_phys_start,
      Waddr storage_phys_end);
    void leave_spec_region();
    void abort_spec_region();

    bool in_spec_region();
    bool has_error();
    ASFStatusCode get_status()

    /* Methods to create particular errors */
    void contention(W64 rip, Waddr addr);
    void capacity_error(W64 rip, Waddr addr);
    void interrupt(W64 rip);
     ...

    /* Handle architectural memory state and aborts */
    bool snapshot(Waddr physaddr);
    bool restore(Waddr physaddr);
    void restore_all ();
    bool restore_other(Waddr physaddr,...);
    bool read_remote_line(Waddr physaddr, W64 *data);
    void handle_arch_contention(Waddr physadr,
      ASFContext *requestor, bool is_store);

  protected:
    struct BackupLine {..};
    struct BackupStorage : std::map< Waddr, BackupLine> {..}
    BackupStorage backup_storage;
    BackupLine* find(Waddr physaddr);

    /* Roll-back information */
    RIPVirtPhys abort_rip;   W64 saved_rsp;
    /* Clobber / link storage information for advanced aborts */
    Waddr storage_virt, storage_phys_first, storage_phys_last;
};
```

Figure 5.23: Interface of the architectural layer used for data versioning and conflict detection in the Marss86 implementation of ASF.

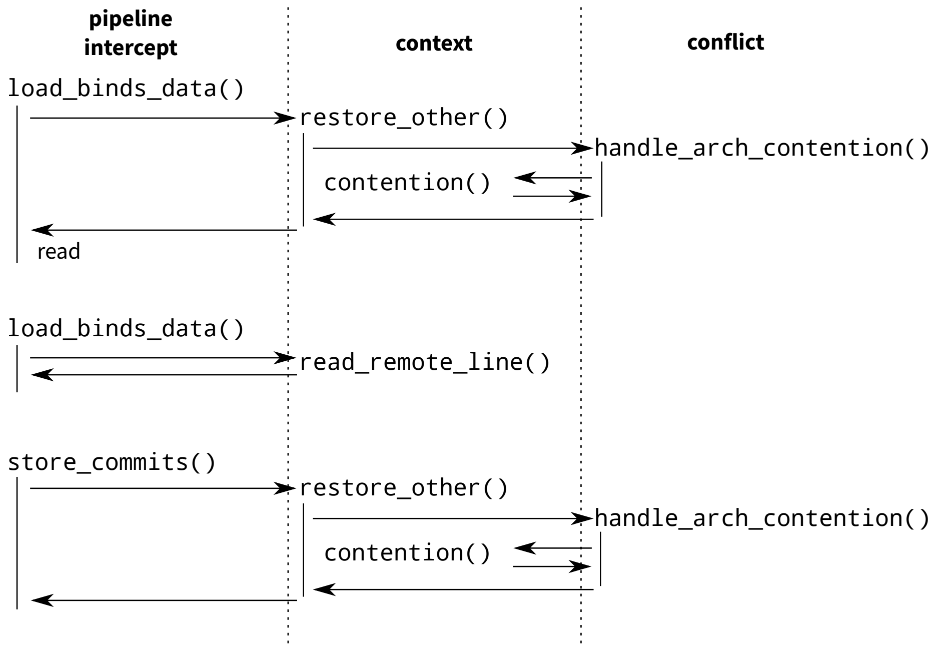


Figure 5.24: Interaction of the ASF pipeline intercept logic and the architectural layer for data versioning. Showing two options for handling RaW conflicts: Top: full contention and abort case; middle: the load reads from the pre-transactional value. Note that the caches may still flag this as contention, depending on their conflict detection policy. Bottom: WaW conflicts always causing an abort in the architectural layer.

non-transactional[5] data and continue their execution.

Transactional memory accesses require additional steps, first: stores will check the local undo-log, and if no entry is present, will create a new entry with the architectural data present before the store updates it in the `snapshot` function. If an entry is already present, no new copy is created, since the original copy already contains the pre-transactional non-speculative state of the cache line in question.

In Marss86, the architectural memory layer uses a per-core `std::map` to implement the $phys\_address \rightarrow data$ mapping. The map will provide fast lookup and modification (faster than $O(N)$, often $O(log(N))$ or even $O(1)$), and also store only the required elements. As such, the capacity is unlimited and thus the architectural layer does not induce any architectural capacity aborts. The microarchitectural layer will eventually limit the size of transactions and abort if they overflow the microarchitectural tracking mechanisms so memory consumption is not a major issue. In practice, the total storage required will be bound by the size of the tracking structure; for the analysed proposals in this thesis, this will be 32kB of additional storage per simulated core plus a small amount of storage needed for the map's internal data structures.

In PTLsim, the architectural memory layer was not yet factored out, and instead was combined with the Locked-Line Buffer used for tracking both transactional reads and writes (conflict detection) and providing versioning for transactional writes. Therefore, it was easy to extend that implementation to also provide *conflict detection* in the L1d cache, but tracking writes in the L1d was complicated due to the intrinsic data-less design of the caches.

In comparison, the separation of the architectural and micro-architectural / timing concerns greatly simplified the design of the cache-based ASF implementation used in Marss86, and allows for much greater flexibility.

Eagerly writing transactional stores to the flat memory view and performing undo-logging simplifies

[5]Non-remote-transactional; the own speculative values will be read through the normal code since no conflict detection with the own undo-log is performed!

the overall integration, because no additional code is required to handle overlapping transactional stores and later loads that are misaligned and / or have different access size. Instead, the existing code for mangling the required bytes as necessary is sufficient and can just read from the flat memory abstraction.

The cost of the required eager conflict detection for every load / store is not problematic, as the simulator itself is single-threaded, and has significant other overheads (as opposed to an efficient STM implementation, where these trade-offs might be made differently).

The architectural layer keeps itself consistent by enforcing the single-transactional-writer property as described through eagerly rolling back concurrent modifications for conflicting lines and then creating a new entry in the requester's undo log before performing the new transactional store directly into memory.

The abort condition needs to be communicated to the micro-architectural and ISA levels of the simulation, and this achieved by the core querying the abort state of the architectural memory layer on every cycle.

Communication also happens in the other direction. The ISA and micro-architectural layer need to be able to control the architectural memory layer (aborts, transaction start and end). This is achieved through processing in the centralised per-core ASF state logic which can invoke the architectural layers `abort` and `commit` methods. These will perform an undo of the entire log (for `abort`) and then clear the undo log (both functions). Upon transaction start (non-nested `SPECULATE`), the architectural layer is already empty.

Overall, the architectural layer does *not* provide full TM semantics, because it only performs *aW conflict detection. The consistency of the read-set must be tracked by the microarchitectural layer.

Micro-architectural Layer / Pipeline Control The microarchitectural level of the ASF implementation consists of two major blocks: ASF-enabled caches / buffers for modelling the conflict detection and versioning mechanisms (with some extension to other tracking structures and request formats), and a central component that hooks into the respective pipeline stages and coordinates all ASF-specific features.

In Marss86, the caches are relatively decoupled from the core logic through a request / wakeup interface. ASF changes the caches by adding ASF.R and ASF.W bits for every cache-line (as separate bits in the tag) and also adds transactional memory handling to the state machine driving the coherence logic. The coherence logic is factored out in `CoherenceLogic`, and I have extended the existing MOESI logic (in `MOESILogic`), added the tracking of the transactional state and also the detection of conflicts for a new `ASF_MOESILogic`, with the interface shown in Figure 5.25.

In Marss86 (and similarly in real coherency protocols), there is not just a distinction between reads and writes, but the coherency protocol consists of more request types, some with additional subclasses. The simple RaW, WaR, WaW conflict matrix extends as follows:

- ASF.R lines conflict with: `MEMORY_OP_EVICT`, `MEMORY_OP_WRITE`, and `MEMORY_OP_UPDATE(new_-state)` where `new_state` $\neq$ `SHARED`
- Lines marked with ASF.W conflict with any incoming hitting snoop message from a remote core / agent

Figure 5.26 shows an excerpt of how that logic is implemented in the simulator.

The core-side part of the ASF logic is concentrated in the `ASFPipelineIntercept` component (Figure 5.27) that hooks into the correct stages of the standard processor pipeline and intercepts processing of ASF instructions and those that may affect ASF state. In addition, the ASF pipeline intercept may also interfere with the flow of the actual pipeline, namely when a transaction abort needs to stop the core from committing further instructions.

The functions `load_binds_data` and `store_commits` interface with the architectural tracking layer for every in-flight memory instruction and ensure functional correctness of the transactional memory

```
class ASF_MOESILogic : public CoherenceLogic {
  public:
    /* CoherenceLogic interface implementation */
    void handle_local_hit(CacheQueueEntry *queueEntry);
    void handle_local_miss(CacheQueueEntry *queueEntry);
    void handle_interconn_hit(CacheQueueEntry *queueEntry);
    void handle_interconn_miss(CacheQueueEntry *queueEntry);
    void handle_cache_insert(CacheQueueEntry *queueEntry, W64 oldTag);
    void handle_cache_evict(CacheQueueEntry *entry);
    void complete_request(CacheQueueEntry *queueEntry, Message &message);
    void handle_response(CacheQueueEntry *entry, Message &message);
    bool is_line_valid (CacheLine *line);
    void invalidate_line (CacheLine *line);

    /* ASF-specifics: notification signals and call-back handlers */
    static void ASF_commit(W64 addr, CacheLine *line);
    void (*get_ASF_commit()) (W64, CacheLine*) {return ASF_commit;}
    static void ASF_abort(W64 addr, CacheLine *line);
    void (*get_ASF_abort()) (W64, CacheLine*) {return ASF_abort;}
  private:
    MOESILogic moesi_logic;    // Wrapped default MOESI logic
    /* Convenience functions for querying ASF state of cache lines */
    static ASF_RWCacheLineState get_ASF_state(const CacheLine *line);
    static void add_ASF_state(CacheLine *line, ASF_RWCacheLineState s);
    static void clear_ASF_state(CacheLine *line);
    static bool is_ASF_spec(const CacheLine *line);
};
```

Figure 5.25: Interface of the `ASF_MOESILogic` class that adds ASF state tracking and conflict detection to the Marss86 simulator.

implementation by providing a safety net of conflict detection and data versioning, as described earlier. The `commit_store` function creates an undo-log entry for a transactional store.

Finally, the ASF pipeline intercept component also tracks the timing of aborts caused by contention (`check_conflicts`) and through other abort reasons, such as interrupts, illegal instructions, etc., through `handle_far_control_transfer`.

For the ASF `SPECULATE` and `COMMIT` instructions, new uops are added: `OP_spec` and `OP_com` that trigger the actions of starting / committing of the transactions through their flow through the pipeline in the ASF pipeline intercept component. ASF transactional memory instructions are decoded into the normal code sequence (potentially adding address arithmetic etc) and have their load / store uop marked with an ASF bit which triggers special handling in the ASF pipeline intercept functions.

ISA Layer The ASF ISA layer is split into the decoder front-end that has added the encodings for the ASF instructions to an empty area in the AMD64 instruction set encoding space, and converts the instruction bytes into instruction sequences containing the `OP_spec` and `OP_spec` uops respectively. The decoder also had to be changed to detect the LOCK prefix in front of normal AMD64 loads / stores (`LOCK MOV`s), and add the ASF flag to the resulting load / store uop.

The ISA behaviour is largely contained in the `ASFContext` class which knows about the ISA-specific details of the ASF implementation, e.g., which registers hold which parameter in case of an abort, transaction start etc. The ASF context is also responsible for tracking transactional nesting depth. It is mainly involved through the ASF pipeline intercepts at the issue and commit stage to provide the right values in the right registers.

Basic Layer Interaction Generic transactional flow is induced from the ASF instructions, as follows:

```
ASF_MOESILogic::handle_local_hit(..) {
  moesi_logic.handle_local_hit (..);
  if (req−>get_isASF()) {
    if (req−>get_type() == MEMORY_OP_WRITE)
      add_ASF_state(line, ASF_W);
    else if (req−>get_type() == MEMORY_OP_READ)
      add_ASF_state(line, ASF_R);
  }
}
ASF_MOESILogic::handle_interconn_hit(..) {
  state  = get_ASF_state(line);
  if (state  & ASF_W) {
    /* _aW Conflict */
    clear_ASF_state(line);
    invalidate_line (line );
    abort_core       = true;
    convert_to_miss = true;
  }
  else if ((state  == ASF_R) && invProbe) {
    /* WaR Conflict */
    clear_ASF_state(line);
    abort_core = true;
  }

  if (abort_core) {
    controller−>send_evict_to_upper(.., isASF = true);
  }
  if (convert_to_miss)
    moesi_logic.handle_interconn_miss(..);
  else
    moesi_logic.handle_interconn_hit(..);
}
```

Figure 5.26: Marss86 ASF coherence logic for conflict detection.

```
class ASFPipelineIntercept {
  protected:
    ASFContext      *asf_context;
    ThreadContext   *thread;
    bool            wait_for_cache;

  public:
    /* Issue Stage: transactional loads / stores */
    int  issue(ROBEntry& rob, IssueState& state, W64 radata, W64 rbdata, W64 rcdata);
    int  issue_load(ROBEntry& rob, LSQEntry& state, LSQEntry* sfra);
    bool issue_probe_and_merge(W64 physaddr, bool inv, W64& out_data, ROBEntry *rob);
    int  issue_store(ROBEntry& rob, LSQEntry& state);
    void load_binds_data(ROBEntry& rob, W64 *data);

    /* Commit Stage: transactional stores, and normal commit intercept */
    void store_commits(ROBEntry& rob);
    bool commit(const Context &ctx, ROBEntry& rob);
    int  pre_commit(Context& ctx, int i);
    int  post_commit(Context& ctx, int i);

    /* Intercepts for handling OOO misspeculation */
    void annul_replay_redispatch(ROBEntry& rob);
    void reprobe_load(ROBEntry& rob);

    /* Handling cache & fence interactions */
    void notify_cache_done();
    bool fence_can_commit();

  private:
    int  issue_mem(ROBEntry& rob, LSQEntry& state, LSQEntry* sfra);
    int  issue_release(ROBEntry& rob, LSQEntry& state,
          Waddr& origaddr, W64 ra, W64 rb, W64 rc, PTEUpdate& pteupdate);
    bool commit_load(ROBEntry& rob, Waddr physaddr, Waddr virtaddr);
    bool commit_store(ROBEntry& rob, Waddr physaddr, Waddr virtaddr);
    int  vcpuid() const;
    int  check_conflicts(Context &ctx, int commitrc);
    int  handle_far_control_transfer(Context &ctx, int commitrc);
    void rollback();
};
```

Figure 5.27: Interface of the `ASF_MOESILogic` class that adds ASF state tracking and conflict detection to the Marss86 simulator.

Starting a transaction involves the OP_spec uop and will simply record the start of the transaction and the current instruction pointer as part of the ASF context for a potential later abort.

Commits clear the architectural memory tracking layer and will send a ASF_COMMIT request to the cache which will then eventually clear the ASF markings of the lines there and end the transaction through the ASF_commit callback of the ASF-MOESI logic.

Aborts eventually will show up as error conditions in the ASF context and will be checked in the ASF pipeline intercept post_commit method. If such an error is detected, the core sends a ASF_ABORT request to the cache that will then discard all transactional updates and then clears all ASF marking utilising the ASF-MOESI logic ASF_abort callback. In addition, the architectural tracking layer will actually undo all the writes and discard the entries of its undo-log. The core in this case must wait for the abort to complete before it can start a new ASF transaction.

A variety of causes can cause aborts of transactions. Enumerating all of those reasons was a major challenge in the simulator implementation of ASF. The following abort reasons are captured:

Conflicts in architectural layer will set the ASF context to contention state directly. The ASF context state is polled for errors on every post_commit call of ASF pipeline intercept and before the commit of an ASF_com uop (in commit).

Conflicts in microarchitectural layer are detected in the cache hierarchy, or more precisely in the ASF-MOESI coherence logic in handle_interconnect_hit. The cache sends a special ASF_CONTENTION message (as a snoop message) to the victim's (holder) CPU core's backprobe method. If the core is currently inside a transaction, the ASF context will be set to the contention abort state, which then will trigger the abort in the next post_commit.

Out of capacity situations are also detected by the ASF-MOESI coherence logic, namely in handle_-cache_insert, i.e., when a new cache line is being inserted. If the new cache line displaces an old line that has ASF mark bits set, the eviction will also send an ASF_CONTENTION message to the core. The core disambiguates between contention and capacity overflows through the sender's core-ID field. If a core "contends" with itself, then the issue is reflected as an ASF capacity abort in the ASF context, again checked in a subsequent post_commit.

Interrupts and exceptions originate from the pipeline and are detected through the ASF pipeline intercept's handle_far_control_transfer called from post_commit. These events are already detected and handled specially in the simulator's pipeline, due to their non-local control flow changes that require a pipeline flush. The ASF pipeline intercept (1) converts the detected condition into the right ASF abort case and ASF context representation and (2) ensures that the non-local control flow *appears to be sourced from the aborted transaction*.

Disallowed instructions are also detected like interrupts and exceptions above. The simulator employs special *assists* which are special uops that provide complex functionality that does not have to be modelled as separate uops, but instead can occur at a controlled point in the pipeline (when at the head of the ROB), and also query / set simulator data structures directly. The assists are used for complex instructions, such as calling into the OS, I/O, setting flags etc. The ASF implementation maps those instructions that are disallowed in an ASF transaction to the respective assist types and flags a disallowed-instruction error if such an assist is detected. Allowed instructions call the assist function with the transaction staying alive.

5.4.3 Simplifications

Due to the simulator implementation, several aspects of a real microarchitectural design can be simplified significantly. The decentralised nature of the implementation would usually require many small changes in the design of a microprocessor. In the simulator implementation function calls are virtually free, it is therefore easy to contain most of the implementation logic inside a single component and only add intercept hooks from the relevant pipeline stages into a centralised component. In a real design such an implementation would likely cause timing violations due to the long wires required to connect all pipeline stages to the centralised ASF component.

In general, in a simulator it is easier to cheat physics than in real CPU designs. Accessing registers, copying data etc. can all happen in zero simulated time. In my design I have used this to simplify the logic for transaction start and the abort path. I am confident, however, that these changes do not have a critical impact on the fidelity of the implementation because they are (1) not on the performance critical path and (2) if they were, there are well known techniques to improve the performance at the cost of additional implementation effort in the real design.

It is typical for a microarchitectural design flow to start out with a high-level design similar to this simulator implementation. During the refinement process, it may become necessary to split / replicate state and change interfaces between components so that timing violations can be dealt with. This usually means that the actual implementation will be somewhat more complex, but still be true to the original high-level design and have similar (if not identical) performance characteristics. Compared to other evaluation approaches (such as widely used one-IPC in-order cores), a design in a cycle-approximate simulator is the closest step before going to the actual processor RTL which had been out of scope for this work.

Another simplification of a simulator implementation (in particular in a timing-driven simulator) is the sequential execution and global visibility of events. Since the simulator goes through every component's `tick()` function sequentially, there are no concurrent accesses to shared data structures, such as the bus, memory or register state. This simplifies the design from a software engineering perspective (multi-threading the simulator in some way would make this harder!), and also from a modelling perspective, because some safety mechanisms that control sequencing in a real microarchitectural design (where everything happens in *parallel* by default!) are simply of no concern. For high simulator fidelity however, important simulated events are modeled to take simulated time and then there is concurrency between simulated events; I have described some issues I have found with our implementations earlier. Again, the correct sequencing of the logic of a high-level design is a typical activity when creating the low-level RTL implementation, and had been out of scope.

The global visibility is another manifestation of the cheat-physics option in simulators. Since there are no actual wires involved when connecting components and no bandwidth / pin-count limitations, it is easy to directly connect two components that may be at other ends of the final chip, where they would probably have to go through a designated shared interconnect. None of the timing ASF components in the Marss86 ASF implementation use such a short-cut, but the PTLsim-based implementation was forced to employ such a mechanism to implement ASF conflict detection because there was no fully modelled coherency protocol available.

The mechanism where both simulator implementations use zero-cycle global communication is the architectural memory tracking layer which acts as a safety net and ensures correct write tracking even in cases where the more detailed timing simulation may exhibit a buggy corner case.

5.4.4 Simulator Extensions and Complications

Clearly, cycle-approximate simulators operate at a higher level of abstraction than microprocessor RTL. As such one expects them to be easier to extend and prototype features in while still giving accurate performance predictions and hint at interaction patterns between components.

During my work on both PTLsim and Marss86, I did, however face several challenges specific to the nature of the simulator implementations. First and foremost, the main concern is the general level of immaturity in these simulators. In comparison to a commercial microarchitecture, the simulators often contain more bugs, because they are not as rigorously validated / tested. I have fixed a significant number of bugs in both PTLsim and Marss86 simulators, in different areas of the design; often in tricky areas such as pipeline mispeculation recovery, coherence protocol, etc.

The second challenge in simulators is the lack of microarchitectural features. If a feature is not relevant to model, it is often not included in the simulator implementation due to lack of awareness / development time, or a concious decision, such reduced simulation complexity and thus faster execution time or simpler simulator code.

In PTLsim, the lack of a cache coherence protocol required me to add a first-order approximation model (simple MSI coherence) with constant cache-to-cache propagation delays and a simplified coherence protocol. The PTLsim model has zero-cycle cache-line invalidations and no bandwidth limitations of both normal memory requests / responses and snoop messages.

In both simulators, the great simplification of a single global flat memory space makes it hard to deliberately have multiple locally-visible versions of data in the system. Adding such a mechanism then needs laborious and error-prone detection of all places that silently rely on the simplification and their change to the new mechanism. The single flat memory view also permits zero cycle communication between cores in a window where the flat view already has the updated new value of the producing store, while the microarchitectural view has not yet propagated the invalidation to the consuming load that therefore still has a cache hit which should read the old data, but can already inspect the freshly produced data through the flat memory view.

The availability of such short-cuts and safety nets is great for rapid prototyping, but when a feature needs prototyping that requires the proper functionality, retro-fitting a detailed model can be hard. In particular for interconnect networks and coherence protocols, where both liveness and correctness are of paramount concern for commercial designs and are areas for notorious bugs and research [70, 122], the reliance of a safety-net leads to a potentially large number of issues lurking in the corner cases of the actual mechanism.

For the simulator implementation of ASF this meant that a significant amount of development, debugging and bug-fixing effort was spent on existing features / functionality. In particular in Marss86's new cache coherence protocol I spent several man-months on debugging deadlocks and correctness issues of the coherence protocol. These had not be found because the coherence protocol was never responsible for the actual data delivered between loads and stores, and therefore the protocol contained functional bugs. Another missing feature that I added to both simulators was the proper support for the AMD64 memory model, in particular in-order loads. Both simulators aggressively execute loads from different addresses out of order, which can be visible to carefully crafted litmus tests. The AMD64 memory model (and the Intel equivalent) [209] does enforce that loads execute in order. I therefore had to add the required logic to detect and fix when the effects of out-of-order loads were visible to the application [27].

Another unfortunate simplification in the Marss86 simulator is the direct support of misaligned / cache-line straddling loads / stores. These induce very subtle corner-case in real microarchitectures for ISAs that support these, as they require split / merging of multiple cache lines and need delicate tracking logic. This logic is essential in real microarchitectures and had been present in PTLsim which split these misaligned loads / stores in the cores and exposed them as multiple aligned loads / stores (with valid bit masks) to the other components in the system. For the Marss86 simulator, this logic had been removed to speed up simulation due to reduced complexity in the simulator[6]. Since caches, LSQs, etc., never transport any actual data, this is only a small issue, since the actual access to the global flat memory view can happen in a misaligned fashion. For the ASF implementation, however, this complicated the logic for handling marking and tracking of transactional cache lines.

Finally, the C++ structure with heavy use of templates and encapsulation made it sometimes a little challenging to connect the right components with one another. Especially reaching into the coherence logic from inside the CPU core on a transaction abort / commit needed poking holes through several layered interfaces. This observation puts the earlier observation into perspective that in C++ everything is just a method call away. Similar problems would be expected from a real implementation due to the distance between the decision making and the mechanism that is invoked; there they manifest in timing violations / routing problems if a plain wire is used to connect the remote components.

ASF-specific Simulator Challenges For the ASF implementation, a few things were crucial and tricky to get right in the simulator environment. I have already mentioned the general unreliability of the coherence protocol implementation, and the effects of the single flat global memory view.

For ASF, these complicated the implementation, because I effectively had to track conflicts in two layers: in the architectural memory view where an undo-log is kept per core for both versioning and conflict-detection with stores, and in the microarchitectural realm for both timing and correctness.

Initially, I had hoped that the architectural view could provide a safety net so that the transactions would be sound even in the face of a buggy coherency protocol. The issue is, however, that the outcome of most conflicts is strongly dependent on timing which is not faithfully modelled in the architectural view. The result was that both layers often disagreed on when a conflict would happen and also which way around concurrent conflicting accesses would be ordered.

I therefore restricted the architectural view to only perform data versioning and conflict detection on the transactional stores with other loads and stores. These still, however, had a large number of conflicts that disagreed in timing and abort decision between the microarchitectural and the architectural layer, so I eventually reduced the amount of conflict detection induced by the architectural layer to a minimum by not detecting architectural read-after-write conflicts where a load tries to read from a location that is being transactionally written. Instead, I used the data in the undo-log and forwarded that to the load, effectively performing selective lazy versioning of these stores. This required, however, careful tie in of the forwarding of the old data into the path that handles misaligned loads.

The architectural layer is then only responsible for detecting conflicts between transactional (and non-transactional) stores and will influence abort decisions in these cases.

In all other cases, the correctness of ASF depends on the proper function of the coherency protocol which previously was only relevant for effects on simulated performance rather than correctness. For that reason I had to rework and repair the coherency protocol in the Marss86 implementation.

In addition to the conceptual who-aborts-whom-when issue when having two mechanisms for conflict detection, the general coordination between the two layers was challenging; very similar to the issues mentioned in Chapter 4 about overlapping conflict detection intervals when moving data and the tracking responsibility. Some errors were double undo operations overwriting stored memory, abort decisions being made but being only able to abort transactions at the boundary of x86 instructions and cycles.

[6]Personal communication with Marss86 maintainers.

5.5 Summary

In the course of my work on ASF, I have added support for the instruction set extension to two state-of-the-art simulators for complex CPU cores executing the x86/AMD64 instruction set–PTLsim and Marss86. Both of these provide a significantly more detailed core model than typical one-IPC in-order pipeline models used in most research for HTM. As a result, I have obtained a much deeper understanding of microarchitectural interactions; not least thanks to the significant amount of debug work performed. The resulting HTM implementation (ASF) is thus more believable and realistic than other feature-loaded proposals.

A second result of my deepened understanding and improved functionality, I have been the maintainer of PTLsim. All changes to the simulator, and the extensions to the Hotspot JVM are available publicly as open-source [262, 290, 304]

ASF has been widely evaluated by myself, collaborators in the VELOX project and related joint research, and has been used as the HTM of choice in other TM publications.

Chapter 6

Extensions and New Use-Cases

6.1 Introduction

The published ASF instruction set extension was mainly designed to provide resource efficient transactional execution of transactions with explicitly marked accesses. The inverted mode described already in Section 3.4.1 has been an acknowledgement to supporting unmodified binary code inside a transaction and executing it transactionally.

The non-transactional accesses were mainly envisioned to bypass / ease capacity restrictions for accesses that did not require transactional conflict detection and versioning. In this chapter, I will introduce various extensions to the ISA and the usage of non-transactional execution for new programming patterns.

The next section (Section 6.2), will introduce the general concept of using non-transactional accesses for communication between transactions and the challenges such a model introduces. In Section 6.3, this ad-hoc communication is put on structured foundations and embedded into a more generic notion of parallel nesting of transactions. Finally, in Section 6.4, I will present how non-transactional memory accesses and light-weight extensions to the ASF ISA can be used to resurrect a transaction if it is aborted. Section 6.5 will conclude this chapter.

The work in this chapter has been presented at SPAA 2012: communicating transactions (joint work with Yujie Liu, and Michael Spear) [274], SPAA 2013 and TRANSACT 2015: resurrecting transactions (joint work with Martin Nowack, Michael Spear, and Christof Fetzer) [289, 337].

6.2 Ad-hoc Communicating Hardware Transactions

ASF–the BeHTM presented in the previous chapters of this thesis has *weakened* atomicity properties, due to (1) reduced register snapshots allowing register state to leak out of a transaction, (2) immediate non-transactional stores that will escape the transaction when the store instruction commits, and (3) leaking exceptions out of an aborted transaction.

As already explored, the original usage of these non-transactional accesses was rather implicit: in baseline ASF, transactional accesses had to be explicitly marked and so the non-transactional memory accesses remained. The idea was to carefully manage scarce transactional resources by forcing annotation. During the design of ASF, the non-tx access provided some challenges (mixing tx / non-tx accesses to the same memory location, unintended leakage / overwriting of state through an aborted transaction, and interactions such as aborts in the middle of instructions that invoke multiple non-transactional modifications, such as the stack modifying instructions: `push, call`, which can modify two state containers that

```
Shared: a = b = 0;
Thread 1:                    Thread 2:
tx_start();                  tx_start();
...                          ...
  nontx_store(*a, 1);          while(!nontx_load(*a));
  while(!nontx_load(*b))       nontx_store(*b, 1);
tx_commit();                 tx_commit();
```

Figure 6.1: Two transactions using non-transactional loads and stores to communicate and coordinate their commits.

are not versioned. These challenges and complexities were reflected in feedback that we received for our initial ASF specification draft from both our project partners in the VELOX project and other academic and industry institutions (collected and presented in [215]).

In addition to said challenges, however, several curious, peculiar and later interesting use cases of non-transactional accesses started to emerge.

One pattern that started out as a challenge is that of using non-tx loads and stores to *coordinate* and *communicate* between ongoing transactions. Figure 6.1 shows a simple case for such a behaviour.

The example at the time was merely used to illustrate complex memory visibility rules that may affect transactional behaviour. In the example, if non-tx stores become visible only after transaction commit, the transactions will never commit (but instead quite likely abort due to a timer interrupt eventually).

Similar ad-hoc communication patterns were used by Riegel, et.al., and Dalessandro, et.al., to coordinate commits between software and best-effort hardware transactions [244, 255]. In their use-cases, the very specific nature of the communication made it possible to work around the identified challenges and for example tolerate non-transactional updates from aborted hardware transactions. Nevertheless, for the creation of [255], Riegel and I had to clarify and refine the semantics of non-transactional memory operations over the state of the published specification at the time.

It is clear that such usage of non-transactional stores (loads are easier to deal with) is hard to generalise and provide as a general-purpose high-level mechanism to the programmer. Together with Spear, et.al., I therefore investigated and developed a more high-level generic communication mechanism and architecture, which is described in the next section.

6.3 Structured Communication: Delegation and Nesting in Best-Effort HTM

In our (joint work with Yujie Liu and Michael Spear) publication at SPAA 2012, we refined the ad-hoc communication model and introduce the concept of a `TXChannel` which is a safe wrapper around the core of non-transactional stores from within transactions. TxChannels enable *delegation*, which can offload work from inside a transaction to be performed by another service thread. We use delegation to perform actions that are not supported inside a hardware transaction (such as memory allocation) on one of the service threads without aborting the hardware transaction.

We generalise the notion of delegation to full, parallel *nesting* in the full paper [274]. There, we also explore data sharing between parent and child transactions of different qualities; an issue we have faced also in ASF, when sharing data between transactional and non-transactional parts of a transaction. Furthermore, recent work by IBM on their Power HTM ISA and implementation [340] and their support for suspend / resume shows similar problems when sharing data between suspended transactions and the currently executing, non-transactional code sequence [374]. In this particular example, the non-transactional code overwrote the return stack of the suspended transaction; not unlike the problem

```
procedure SENDMSGASF(src, dst, type, d[])
    $TxChannel[1 \ldots 7] \leftarrow d[0 \ldots 6]$                ▷ store the payload
    $TxChannel[0] \leftarrow (src, type)$
end procedure

procedure RCVMSGASF(src, dst, type, d[])
    repeat
        $(s, type) \leftarrow TxChannel[0]$                     ▷ spin
    until $s = src$
    $d[0 \ldots 6] \leftarrow TxChannel[1 \ldots 7]$
end procedure
```

Figure 6.2: Sending and receiving small messages with ASF, using non-transactional accesses and relying on well-formed ping-pong communication.

outlined in normal-mode ASF and the handling of the non-transactional stack accesses.

The remainder of this section will focus more on the implementation challenges of the channels themselves and adds some more detail for a particular use-case: that of delegated memory allocation from transactions.

6.3.1 Transactional Channels

We define a TxChannel as an unbuffered, bidirectional communication medium with exactly two endpoints. Communication is asynchronous and polling-based. The TxChannel contains a fixed-size message which can be read and written from either end through `LdTxChannel` and `StTxChannel` functions.

Threads will bind to a single channel and cannot change the binding during the execution of a transaction. Channels can be changed in between transactions however.

In its simplest case, a TxChannel consists of a *single word* in memory that is sufficiently padded that it will not overlap any other transactional / non-transactional memory location; for ASF, placing the channel communication word in its own 64 byte cache line is sufficient. Performance considerations may add additional padding to prefetch and cache index conflicts. The strict binding mechanisms ensure that threads and transactions do not have issues with overlapping working sets, and the access functions ensure that a proper protocol can be kept.

Due to the well-formedness of the usage of the channel, multi-word channels can be synthesised on top of single-word non-tx ASF primitives: one approach extends the channel to support fixed-size messages by introducing a status word that indicates whether earlier multi-word communication through single non-tx stores is complete, and spinning in the receiver on the status word. The example source code in Figure 6.2 shows the simple behaviour.

Further handshaking can be used to transfer long messages through a TxChannel by chunking the message into fixed size parts and acknowledging each successful transfer between endpoints. The extended code example in Figure 6.3 uses the fixed-length functions and performs the handshakes.

6.3.2 Communication Patterns

To simplify the communication patterns, we propose the use of a single *distinguished software thread*(DT). DT is the switchboard of the communication graph between the transactions and controls the higher-level communication / delegation / nesting protocol. Several BeHTM and STM threads share the same DT, as shown in Figure 6.4.

Transactions communicating with DT observe a *channel protocol* that will ensure that communication is well-formed on a high level with respect to transaction commits and aborts and access to the channel

```
procedure SENDLONGMSGASF(src, dst, type, d[])
    $rem \leftarrow len(d)$
    while $rem > 0$ do                                          ▷ store operands in temporary
        $n \leftarrow max(rem, 7)$
        $tmp[0 \ldots (n-1)] \leftarrow d[i \ldots (i+n-1)]$
        $(rem, i) \leftarrow (rem - n, i + n)$
        $sendMsgASF(src, (type, n, rem), tmp))$
        $rcvMsgASF(dst, t, ign)$                                 ▷ wait for ack
        $assert(t = (type, OK))$
    end while
end procedure

procedure RCVLONGMSGASF(src, dst, type, results[])
    $i \leftarrow 0$
    repeat                                                      ▷ receive message in chunks
        $rcvMsgASF(src, t, d)$
        $(type, n, rem) \leftarrow t$
        $results[i \ldots (i+n-1)] \leftarrow d[0 \ldots (n-1)]$
        $sendMsgASF(dst, (type, OK), \varnothing))$             ▷ ack
        $i \leftarrow i + n$
    until $rem = 0$
end procedure
```

Figure 6.3: Sending long messages by chunking and acknowledging each chunk.

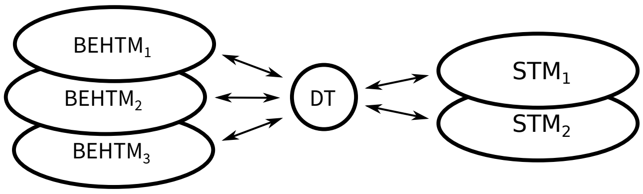


Figure 6.4: Multiple BEHTM and STM transactions using a single service thread DT.

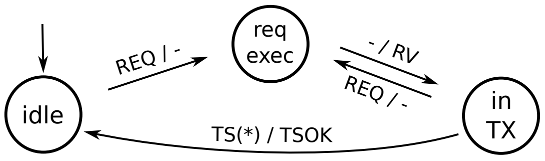


Figure 6.5: TxChannel state transitions, with messages sent to / by DT. If a transaction sends REQ to DT, it must send a TS on commit / abort. TS cannot be sent while DT is processing a request (reqexec).

is properly coordinated. The main challenge is to notify DT of aborts and commits of the main BeHTM transaction that initiated the communication. Figure 6.5 depicts the main interaction: upon receipt of the channel message, DT will register on-abort / on-commit handlers and perform the action that was encoded in the message ("req exec" state). Once the request handling is complete, DT returns the result of the request (return value (RV)), and waits for further requests (state "in TX"), or a transaction status (TS(*)) message from the initiating thread.

The TS message needs to be sent by the originating thread when and if the transaction commits / aborts, after any pending request has been completed by DT (or its delegates) and the thread receives the corresponding return value message.

6.3.3 Aborts During Channel Operation

Enforcing the strict state machine of the TxChannel makes clear who is waiting and who is writing in TxChannel. The low-level primitives tolerate aborts during the send phase. The state protocol ensures that once a request has been started by DT, it can be undone / committed due to DT registering the respective handlers / undo / redo actions. DT it self does not abort synchronously when the invoking BeHTM aborts. If the invoking BeHTM aborts, it will inspect the channel state and realise that it sent data to DT / DT has already responded. The BeHTM will then notify DT of the abort (after possibly waiting for the request to finish executing by polling for the RV message).

Committing BeHTM similarly will wait (with non-tx loads) for the successful receipt of the RV message and then commit the hardware transaction and *then* signal DT of the successful commit.

6.3.4 Use Case: Memory Allocation

One possible use case for the described delegation mechanism is memory allocation from within BeHTM transactions. These are challenging, because they can not always be hoisted out of the transactions (automatically), for example due to scope limitations in the compiler. Executing memory allocation code fully transactionally would work, but will likely encounter contention with other concurrent threads allocating memory, and often also due to meta-data placement with threads accessing nearby, but independent data elements.

Furthermore, memory allocators will eventually invoke the OS to ask for additional heap space. Such OS invocations usually are not gracefully handled by BeHTM and will abort the invoking transaction repeatedly. Executing memory allocators inside ASF with non-inverted semantics would prevent parts of the contention issue, but is not safe: an abort in the hardware transaction may yank execution out of the middle of the allocator code and leave behind inconsistent state, such as taken locks, intermediate logging data etc.

Therefore, memory allocation / deallocation can be a prime candidate for a delegation mechanism, also because the interface to allocation / deallocation code is small (one word in for size / pointer, pointer / no return value).

Embedding `malloc` and `free` into the TxChannel mechanism is simple: these functions will invoke stubs that will send the single parameter and poll for the return of the DT request handling. The allocated memory can then be used inside the invoking BeHTM without any additional operations. For memory allocation, DT registers an on-abort handler that will free the allocated memory; for memory frees, DT will postpone them to after transaction commit in a on-commit handler.

One challenge with using TxChannels for delegating transactional operations out of the transaction is the cost of communication between the hardware transaction and DT. This cost has to be contrasted with the cost of an aborted and re-executed transaction if an abort-allocate-cache-retry mechanism is used to handle in-tx memory allocation.

To get an understanding of the cost associated with communicating between two hardware threads in a ping-pong fashion, we investigated on a real system (AMD Phenom(TM) II X4 945 processor at 3.0 GHz).[1] Naive polling and using a single cache line for the channel, the round-trip time was approximately 600 clock cycles (approx. 200ns), which we were able to reduce to a mere 300 clock cycles (approx. 100ns) with careful data placement (separate cache lines for each communication direction), prefetching and polite polling.

Depending on the size of the transaction and the cost of memory allocators, these costs will be relatively small in comparison to both transaction abort and memory allocator invocation. Reducing the scalability of the memory allocator due to a centralised DT may cause scalability bottlenecks, but may also simplify the allocator design. Other work [275] has shown benefits from centralising specific concurrent application parts.

6.4 Between All and Nothing–Versatile Aborts in Hardware Transactional Memory

One problem with non-transactional, immediate stores inside a transaction is the issue of the aborts that happen synchronously with the abort reason and asynchronously with the flow of non-tx code. As demonstrated in previous sections, this can complicate usage of non-tx primitives that require multiple instructions to modify a data structure. In the example of TxChannels in Section 6.3, it required careful layout and ordering of the channel's send operation because it requires multiple non-tx stores to perform a send.

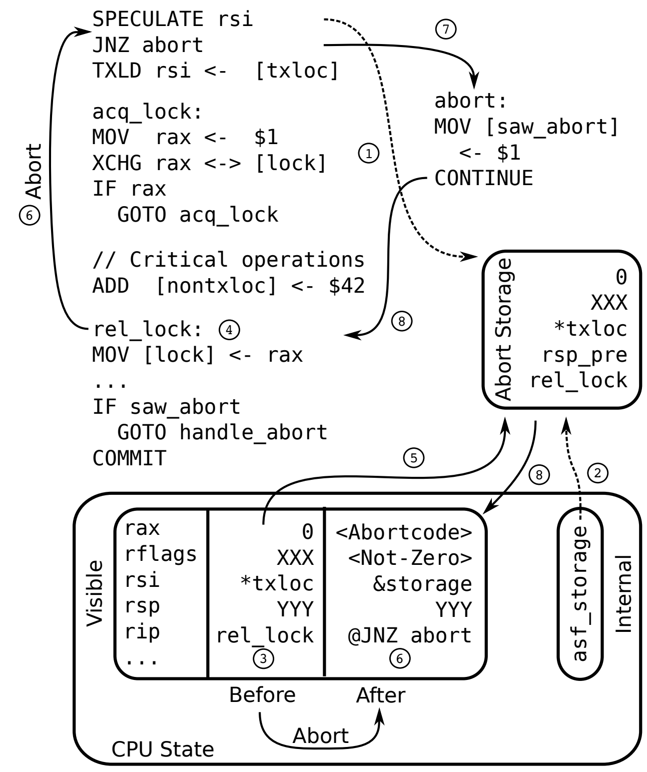
IBM's recent HTM proposal in Power 8 [340] solves this problem by performing transaction *suspend* and *resume*. Using suspend will put the transaction in the background; conflicts will still roll-back the transactional modifications, but will not change the flow of execution. Instead, the abort and change of control flow to an abort handler will be performed on resume of the transaction. Transaction suspend / resume is an elegant solution for the application programmer, but it has two drawbacks: (1) it requires operating system support, because the aborted state of a suspended transaction needs to be recorded in the thread context when switching; and (2) anecdotal evidence suggests that suspending / resuming is slow and complex to verify. Interestingly enough, several interactions when sharing data between a suspended transaction and the code that runs while the transaction is suspended are very similar to the problems that occurred with ASF's non-transactional accesses, for example to the call stack [342]. Naturally, some of the solutions are similar, as well: enforcing the `tx.begin` primitive to be at the outermost layer of the call stack.

In our (joint work with Martin Nowack, Michael Spear, and Christof Fetzer) publication at SPAA 2013 (brief announcement) and TRANSACT 2015 (full paper), we explored the area between ASF's immediate aborts and IBM's suspension of transactions. We propose a mechanism that allows the *resurrection* of an aborted transaction. In our proposal, transactions will abort immediately if an abort condition is detected (such as a transactional conflict); this ensures that the operating system does not have to be modified, because it always sees a clean, non-transactional architectural state, because all ongoing transactions will be architecturally aborted before OS invocation.

In the abort handler, however, all information to resurrect the aborted transaction is available, and execution can continue precisely where the abort had interrupted the execution inside the transaction.

For successful resurrection, the abort handler needs to know the full architectural state of the transaction (including program counter, flags and general purpose registers) at the time of the abort. When

[1]CPU performance simulators are notoriously bad at capturing realistic core-to-core and cache-to-cache communication delays.



```
SPECULATE rsi
JNZ abort
TXLD rsi <-  [txloc]

acq_lock:
MOV  rax <-  $1
XCHG rax <-> [lock]
IF rax
  GOTO acq_lock

// Critical operations
ADD  [nontxloc] <- $42

rel_lock:
MOV [lock] <- rax
...
IF saw_abort
  GOTO handle_abort
COMMIT
```

```
abort:
MOV [saw_abort]
  <- $1
CONTINUE
```

Figure 6.6: Basic functionality of abort with continuation.

that information is restored, the application will continue execution where it left off due to the abort. This feature was largely enabled by extending ASF's limited register snapshot feature (see Section 3.2.7) which again started out as a feature to save precious resources, but then turned into a semantic feature of its own right.

ASF will not save / restore the register state at transaction begin / abort. Instead, the abort handler can inspect the registers and see their values as they were in the transaction. Only a selected number of registers is checkpointed: the program counter, in order to change the control flow to the abort handler and the stack pointer, to simplify call stack handling after abort. The `rAX` and flags registers are overwritten to convey reasons for the abort to the abort handler.

In our resurrection proposal, we show how we can store the in-transaction copies of the overwritten registers without extending the architectural thread state (no shadow registers): the application provides a continuation buffer in memory to the `SPECULATE` instruction, and the CPU uses that buffer to store the in-tx values of the registers. We also add an instruction that will restore the registers from the continuation buffer in memory. Logically, the transaction abort can then be viewed as a user-level interrupt to the transaction and we provide a very slim (five words) interrupt / resume state.

Figure 6.6 highlights the life cycle and main interactions: `SPECULATE` is extended so that it accepts a memory buffer location parameter (1), looks up the virtual address and translates it to a physical address, and also checks write permissions to the location. Any page faults that could occur when accessing the buffer are thus already resolved before the transaction starts. In the event of nested transactions, the `SPECULATE` instruction ignores this parameter: Since ASF only supports flat/subsumption nesting, there is no meaning or benefit to saving multiple register checkpoints.

The CPU keeps the resulting physical address in an internal register (2) and starts the transaction. The transaction executes, mutating the CPU's register state (3). In case of an abort (4), the processor first copies `rax`, `rip`, `rsi`, and `rflags` into the application-provided buffer (5), and then updates these registers and control flow to reflect the abort condition (6), with `rsi` additionally holding the buffer address. Furthermore, `rsp` will no longer be restored:[2] this prevents stack smashing due to signals or interrupt handlers running within the abort handler. The application code checks for aborts and branches to an abort handler (7). The abort handler can simply restore `rsi` and `rsp` from the the buffer pointed to by `rsi` and reproduce the original ASF abort functionality. However, it can also resume the code in the transaction by restoring all overwritten registers from the buffer (8). Since existing assembly primitives cannot restore all registers without overwriting an additional temporary register, we provide a new `CONTINUE` instruction that performs a simple micro-code sequence to restore the registers.

6.4.1 Challenges

Despite its simple design, several challenges in the resurrection approach became apparent during the actual implementation. Continuing after an abort only to finish a non-transactional section of code (and then abort) is straight-forward; no additional transaction needs to be started. For full transactional resurrection, however, the continuation code needs to start a new transaction with a new continuation buffer and then resurrect the old transaction through the state in the original continuation buffer. Unfortunately, during the starting of the new (shell) transaction and completing the resurrection of the previous aborted transaction, the shell transaction might abort.

This could create a spiral of new shell transactions that try to resume another shell transactions resume operation, with unlimited depth and unlimited requirement for continuation buffers. To cut the potential deadlock, we detect the "fractal" case and jump directly to the resurrection of the original outermost transaction. This limits the amounts of continuation buffers required to two, and also reduces the susceptibility to pathological abort timing conditions. Figure 6.7 shows the pseudo-code for the resumption operation with these details highlighted.

Aside from complications in the usage and ISA design of the resurrection primitives, the resurrection mechanism uncovered a latent issue with the existing ASF implementation: resurrection relies on the consistency of the register state of the thread and the memory operations performed. One issue was that if an ASF abort occurred during an instruction that was broken up into multiple micro-ops, the register state was not consistent: the instruction pointer pointed already to the next instruction, but parts of the previous instruction had not executed and not performed their effect (register / memory write). In normal transactional memory implementations, this is rarely a problem, as long as all state modifications are rolled back / discarded and not used to infer / reconstruct properties of the transaction at the time of abort.

Repairing this defect in the simulator required more careful timing of aborts with the execution of the instruction stream and only triggering aborts once execution is between two ISA-level instructions (see Section 4.4.6).

6.4.2 Use-cases of Transactional Resurrection

Executing multiple non-tx stores As presented, transactional resurrection solves the problem of aborts disrupting sequences of multiple non-tx stores, because the execution of the non-tx code sequence can be continued after the abort. That way, a simple mechanism could register (in a thread-local variable with a non-tx store) a do-not-abort (DNA) intent before executing non-transactional code. If the enclosing transaction aborts, the abort handler will evaluate the DNA flag, if set will store an abort-registered (AR) message (in another thread local location, or overwriting the DNA flag), and then resurrect the aborted transaction. The aborted transaction will then continue with its non-transactional code sequence and, at

[2]The old value of `rsp` from `SPECULATE` is stored in the buffer, instead.

```
 1: procedure SAHTM_RESURRECT($s$)                                     ▷ $s$ holds the resume state
 2:     if $s.resume\_ip \in$ lines(5 – 10, 17 – 27) then                ▷ Aborted while resuming
 3:         $s \leftarrow s.other$                                     ▷ Squash abort recursion
 4:     end if
 5:     SPECULATE $s.other$                                  ▷ Start HTM container transaction
 6:     $error \leftarrow$ rax
 7:     if $error = 0$ then                                  ▷ Successful start of HTM container
 8:         SAHTM_REPLAY()                         ▷ Replay transactional working set from SW log
 9:         pop regs
10:         CONTINUE $s$                    ▷ Restore full state and return to resurrected transaction.
11:     else                                                 ▷ Abort in the resurrected transaction
12:         $s \leftarrow$ rsi                                ▷ Update resumed state from new abort site
13:         $s.abort\_rsp \leftarrow s.other.abort\_rsp$
14:         return $(error, s)$                    ▷ Outer logic handles abort condition and retries
15:     end if
16: end procedure
17: procedure SAHTM_REPLAY                            ▷ Replay and validate transactional accesses
18:     for $(addr, val, rw) \in log$ do                                      ▷ from the SW log
19:         if $rw = READ$ then
20:             txload $tmp \leftarrow [addr]$                                   ▷ Add to read set
21:             if $val \neq tmp$ then                                     ▷ and validate value
22:                 ABORT $CONTENTION$                  ▷ Use HTM abort to unravel validation failure
23:             end if
24:         else
25:             txstore $[addr] \leftarrow val$                                     ▷ Redo stores
26:         end if
27:     end for
28: end procedure
29: procedure SAHTM_TXLOAD($addr$)                                 ▷ Software-assisted read barrier
30:     $log \leftarrow (log, (addr, \emptyset, READ))$       ▷ Append a sentinel protecting against an abort
31:                                                   ▷ between line 32 and 33 missing replay of $addr$
32:     txload $val \leftarrow [addr]$                                     ▷ Add to HTM read set
33:     $(log, (addr, \emptyset, READ)) \leftarrow (log, (addr, val, READ))$    ▷ Update with proper read value
34:     return $val$
35: end procedure
36: procedure SAHTM_TXSTORE($addr, val$)                          ▷ Software-assisted write barrier
37:     $log \leftarrow (log, (addr, val, WRITE))$                               ▷ Append to log
38:     txstore $[addr] \leftarrow val$                                    ▷ Add to HTM write set
39: end procedure
```

Figure 6.7: Resurrection and replay of aborted transactions, logging read / write barriers. For simplicity, we omit handling different sizes in SAHTM_TXSTORE and SAHTM_TXLOAD.

```
 1: procedure ASTART($handler$)                          ▷ Start AOU and specify alert handler
 2:     $buf_1, buf_2 \leftarrow$ malloc()
 3:     $(buf_1.other, buf_2.other) \leftarrow (buf_2, buf_1)$          ▷ Cross-reference resume buffers
 4:     $locs \leftarrow \emptyset$                                     ▷ No AOU-watched locations, yet
 5:     SPECULATE $buf_1$                                ▷ Start the enclosing HTM transaction
 6:     $error \leftarrow$ rax                           ▷ HW returns here from application on update
 7:     if $error \neq 0$ then
 8:         ACONTINUE(rsi, $error, handler$)             ▷ Handle abort condition and resume
 9:     end if
10: end procedure
11: procedure ACONTINUE($s, error, handler$)
12:     push regs
13:     repeat
14:         if $s.ip \in$ lines 21 – 34 then
15:             $s \leftarrow s.other$                   ▷ Squash abort recursion.
16:         end if
17:         if $error.type = CONTENTION$ then            ▷ Someone updated our AOU locations
18:             $handler()$                              ▷ Invoke the alert handler
19:         end if
20:         ...                                          ▷ Handle other abort reasons
21:         SPECULATE $s.other$                          ▷ Start the continuing HTM transaction
22:         $error \leftarrow$ rax                       ▷ HW returns here from application on update
23:         if $error \neq 0$ then
24:             $s \leftarrow$ rsi                       ▷ Update buffers on error
25:         end if
26:     until $error = 0$                                ▷ Handle all abort conditions
27:     for $(addr, exp\_val) \in locs$ do               ▷ Re-add AOU locations
28:         txload $val \leftarrow [addr]$
29:         if $val \neq exp\_val$ then                  ▷ Value-based validation for updates while handling
30:             ...                                      ▷ Signal alert and retry
31:         end if
32:     end for
33:     pop regs
34:     CONTINUE $s$                                     ▷ Continue to the application code, does not return
35: end procedure
```

Figure 6.8: Implementing alert-on-update with ASF.

the end of it, will check for any AR events. If such an event is detected, the transaction will return to the abort handler and finally perform the full, delayed abort action.

Use as an alert-on-update mechanism Alert-on-update (AOU) [153] is a mechanism that allows polling-free communication through memory between user threads sharing virtual memory. In AOU, the user of the mechanism declares to watch one or multiple locations. Whenever another thread writes, all threads that were watching that location will get notified synchronously by having their flow of execution being diverted to an alert handler. Originally presented as a separate ISA extension, the non-transactional loads and stores in ASF together with the resurrection mechanism make it possible to synthesise AOU primitives without additional changes to the ISA. Generally, the mechanism works as follows (further detail can be found in Figures 6.8 and 6.9 and in the full paper): it is easy to conceptually map watched locations to a transactional read set, alert notifications to aborts, and the alert handler to the transactional abort handler. Wrapping large sections of application code into transactions is problematic for forward progress. Non-transactional memory accesses for all application code and transactional resurrection will, however, maintain the full state of the aborted transaction. The mechanism therefore works as follows: application code is wrapped into ASF transactions when the AOU functionality is to be used, and all

```
36: function ALOAD($address$)                         ▷ Adds an alert on update of location $address$
37:     repeat
38:         $val_1 \leftarrow [address]$
39:         $locs \leftarrow locs \setminus (address, *) \cup (address, val_1)$   ▷ Add value early to prevent data race
40:         txload $val_2 \leftarrow [address]$
41:     until $val_1 = val_2$                          ▷ Ensure that this was a race-free ALOAD
42:     return $val_1$
43: end function
44: procedure AEND                                     ▷ Disable AOU handling
45:     COMMIT
46:     free($buf_1, buf_2$)
47:     $locs \leftarrow \emptyset$
48: end procedure
```

Figure 6.9: Implementing alert-on-update with ASF, part 2.

memory accesses from the application remain non-transactional. To mark a memory location to perform AOU, the implementation simply adds them to the transactional read set. Due to strong isolation properties, any concurrent store to such a location will abort the transaction and thus allow notification of the concurrent update. Using the resurrection mechanism, the aborted / notified thread can re-establish the set of watched locations and continue executing the main flow of execution, not losing any progress from the abort.

A few of the challenges for implementing the mechanism are: handling non-AOU induced aborts, for example due to page faults or system calls. Generally, all these cases are handled as follows: hardware aborts the transaction, the abort handler checks the abort reason and if necessary (system call) re-executes the illegal operation (perform the system call) and then resurrects the transaction. The set of current AOU locations is maintained as a list updated only with non-transactional accesses. Therefore, the resurrected transaction can again add the watched locations and perform value-based validation on them. Unfortunately, this leaves this AOU implementation briefly susceptible to the ABA problem. Another option would be to exhibit false positives–application notifications even though none of the AOU-watched locations has changed. Using lazy cache cleaning on abort would help and tighten the area of ABA susceptibility / the amount of false positives, but not reduce either to zero.

In the full paper, we implement the presented AOU mechanism and extend an STM implementation with a fast revalidation signal in order to avoid issues of privatisation-safety without adding heavy synchronisation operations and waits in the transaction commit handler.

6.4.3 Working Set Handling

Saving and restoring the full register state of the aborted transaction enables the two use cases just presented: postponing an abort after a sequence of non-transactional memory accesses, and implementing alert-on-update. To enable full transactional suspend / resume-like behaviour, the transactional working set itself needs to be preserved across the abort / resurrection, too.

In our publication, we identify two ways of keeping the working set across aborts. The first approach is hardware-centric and changes the ISA and implementation of the BeHTM mechanism, such that aborts for non-conflict reasons will perform lazy rollback (only when a later conflict is detected / when a fresh transactional state is requested). We have added a "dirty" transaction start instruction that continues using the remaining transactional state in the tracking structures for the transactional resurrection; if the transactional state had to be discarded in the meantime, the newly started transaction will abort with a new abort code "cannot resurrect".

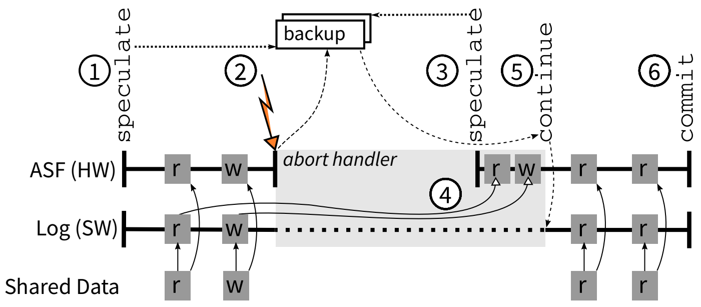


Figure 6.10: Suspend/Resume mechanism: (1) A hardware transaction is started (SPECULATE with "backup" as argument), but the transaction body is instrumented so that accesses will also be logged; (2) In case of an abort, ASF records the instruction pointer `rax` and executes the abort handler; (3) If the transaction can be recovered, the handler starts a new hardware transaction; (4) The working set is replayed; (5) The transaction resumes the normal execution (using CONTINUE); (6) The transaction commits its hardware transaction and resets its logs.

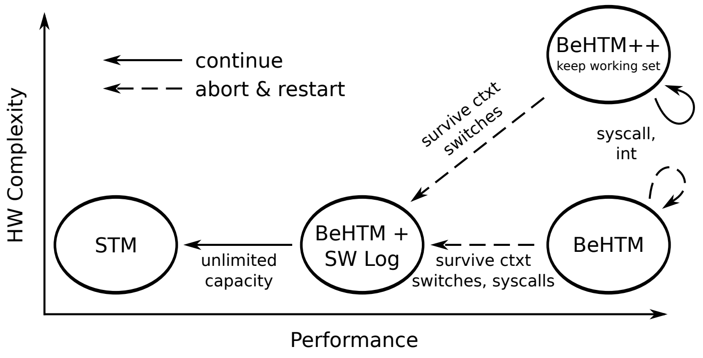


Figure 6.11: Switching between HTM algorithms and STM.

In essence this is the Power suspend / resume feature on a microarchitectural level, while the architectural layer performs a full abort / resurrection operation to avoid modifications to the architectural state of applications.

Software-assisted HTM Instead of changing the underlying transactional working set tracking in hardware, it is possible to use the mechanism as a way to fully implement suspend / resume hardware transactions *with software assistance* in a mode called Software-assisted Hardware Transactional Memory (SAHTM). We employ simple thread-local logging of the read / write sets and they are only used to reinstate the hardware versioning and conflict detection structures on resurrection. Non-tx memory operations are used to add data to the logs, so that they survive the abort of the hardware transactions. Upon resume, simple value-based validation is used to ensure that the working set is still intact. Figure 6.10 visualises the flow of transactional execution and resurrection; and Figure 6.11 shows how these mechanisms can be deployed together and how a hybrid TM may switch between them.

6.4.4 Evaluation

To evaluate the transactional resurrection scheme, I implemented the new instructions inside the existing implementation of ASF in Marss86 (see also Chapter 4). Together with my co-authors, we extended two STM runtimes: TinySTM and RSTM [101, 160].

Overhead of Software Logging The extensions to TinySTM use the transaction resurrection feature and mainly evaluate the added overhead of performing read- / write-set logging in software in addition to HTM. We used DTMC [213], the Dresden TM Compiler, to instrument and *optimise* memory accesses

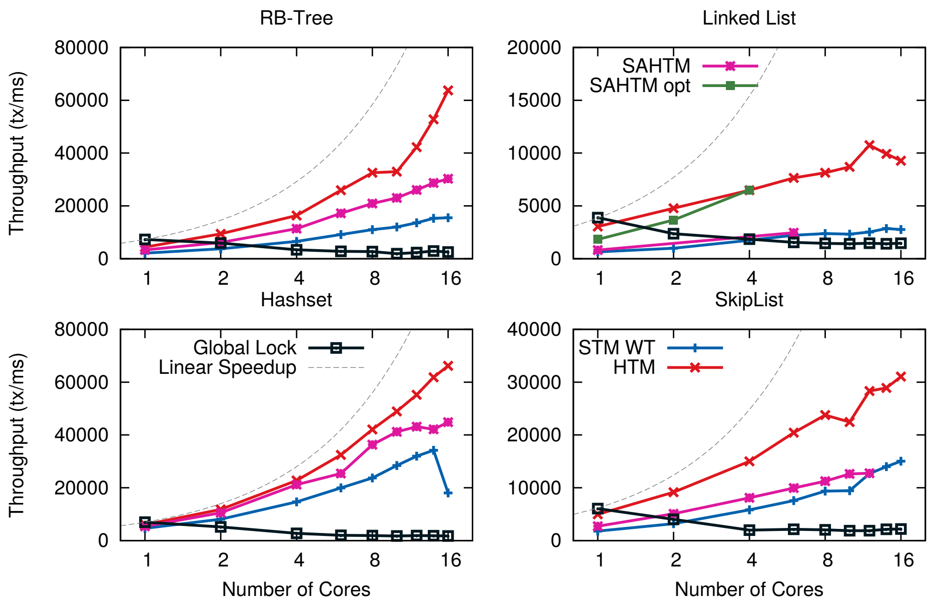


Figure 6.12: Resurrection and the overhead of performing read / write set logging in software and conflict detection in hardware.

inside transactions. DTMC uses LLVM and supports link-time optimisation (LTO), which allows inlining and further optimisation of the SAHTM read / write barriers. Although the additions for logging will increase the overhead for the SAHTM barriers over plain HTM loads / stores, we assume that their performance impact is smaller than full STM read / write barriers. The main reason is that the logging is a thread-local fire-and-forget write-only operation which has good chances to “disappear” in the available ILP of modern CPUs.

The results for selected micro-benchmarks (a worst-case for measuring STM barrier overheads, because there is very little additional non-barrier code) in Figure 6.12 shows even though the logging only barriers are faster than full STM barriers, they are significantly slower than simple HTM barriers (that get optimised to single HTM accesses).

Upon close inspection of the generated assembly code, we found that the LLVM-based DTMC misses an optimisation opportunity of hoisting a load of a constant out of the inner data-structure search loop. As the disassembly in Figure 6.13 shows, the resulting code keeps loading the address for the thread-local storage and log for every barrier, even though they are marked as thread local and are never written to. Despite various attempts at coercing the compiler into hoisting out the load from the loop, we were unsuccessful in doing so. We then resorted to manual assembly modifications, and the results, while preliminary, clearly show the benefit and reduced overhead (graph “SAHTM opt” in Figure 6.12).

Evaluating Alert-on-Update Functionality The alert-on-update functionality provides a non-polling, memory-based, user-level notification mechanism. We have extended RSTM with such a notification mechanism to solve the privatisation problem: a committing software transaction will force immediate revalidation of all concurrent transactions to ensure that their read set is still consistent. Without a mechanism to synchronously notify, privatisation-safe STMs need to rely on polling to wait for other transactions to perform their revalidation before privatised memory can be accessed outside of transactions.

As such, we do not expect a significant performance boost (in particular not for the simple workloads we investigated), but look to see whether our AOU implementation on top of resurrection ASF has any performance problems. As outlined in the previous section, the AOU implementation is conceptually plain ASF

```
lock mov 0x8(%rcx),%rcx
lock mov (%rcx),%rax
cmp      %rbx,%rax
jl       <loop>
```

logging with
compiler optimisation

```
mov     0x8(%rbx),%r14
add     $0x8,%rbx
mov     0x27210(%r13),%rax
cmp     %r12,%rax
jae     <abort>
mov     %rbx,(%rax)
movb    $0x11,0x7(%rax)
mov     %r14,0x8(%rax)
addq    $0x10,0x27210(%r13)
lock mov (%rbx),%rax
cmp     %rax,%r14
jne     <abort>

mov     (%r14),%r15
mov     0x27210(%r13),%rax
cmp     %r12,%rax
jae     <abort>
mov     %r14,(%rax)
movb    $0x11,0x7(%rax)
mov     %r15,0x8(%rax)
addq    $0x10,0x27210(%r13)
lock mov (%r14),%rax
cmp     %rax,%r15
jne     <abort>

cmp     -0x30(%rbp),%r15
mov     %r14,%rbx
jl      <loop>
```

logging with
manual optimisation

```
mov     0x8(%r13),%r14
lea     0x8(%r13),%rcx
mov     %rcx,(%rax)
movb    $0x11,0x7(%rax)
mov     %r14,0x8(%rax)
add     $0x10,%rax
mov     %rax,0x27210(%r12)
lock mov 0x8(%r13),%rcx
cmp     %rcx,%r14
jne     <abort>

mov     (%r14),%rbx
mov     %r14,(%rax)
movb    $0x11,0x7(%rax)
mov     %rbx,0x8(%rax)
add     $0x10,%rax
mov     %rax,0x27210(%r12)
lock mov (%r14),%rcx
cmp     %rcx,%rbx
jne     <abort>

cmp     %r15,%rbx
mov     %r14,%r13
jl      <loop>
```

Figure 6.13: Disassembly code snippets of a linked list traversal showing missed hoisting opportunity by the compiler (left) and manually optimised binary (right). Earlier compiler versions also had to perform a costly read of the `tx` descriptor in TLS to access the log for every log barrier. The code blocks correspond to the expanded single instructions in the "plain ASF" example and show the added complexity due to logging.

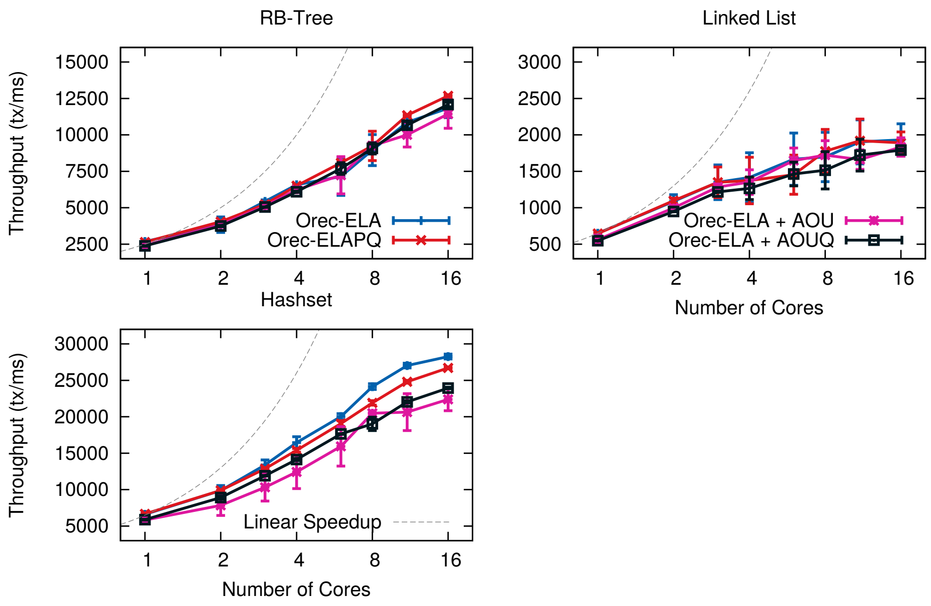


Figure 6.14: Throughput for RSTM OrecELA with AOU-enhanced privatization safety.

simpler, because the hardware transactions only monitor a read-set (the AOU watched locations), while all other memory modifications (including those by the underlying STM) are performed as non-tx operations inside the enclosing AOU hardware transaction. In our implementation, we left the STM algorithm largely unchanged and performed the validation in the AOU alert handler.

Performance data in Figure 6.14 shows that the overhead of AOU is indeed small compared to the overhead of STM, while offering stronger TM semantics.

6.5 Summary

The design process of ASF was largely steered by finding the right balance between programmability and ease of hardware implementation. At several stages during the initial ISA drafting, decisions were made to offload complexity to the software side to keep the hardware implementation lean. When we revisited the resulting constructs, we found in several cases that the resulting "imperfect" implementations actually offered new mechanisms that could be leveraged systematically: communication from within transactions using non-transactional loads and stores, and continuing transactions by capturing their full architectural state at abort.

This bottom-up approach of exposing an imperfect abstraction to save resources, followed by sketching a possible exploitation through novel constructs, and a subsequent prototyping phase to explore performance trade-offs and expose architectural sharp edges, forms a significant piece of the thesis of this work. Understanding costs and details of hardware implementations then allowed us to close the loop and tweak the hardware primitives *without* prohibitive cost to hardware / system software, and still smooth out those sharp edges exposed due to unawareness of the possible use case in the first place. One such example are the new `continue` instruction and the mechanism for lightweight capture of the full register state at transaction abort in the resurrection proposal.

In this chapter, I have shown these synergies and how to strike a balance between compelling feature set at good performance levels and still simple hardware implementation costs. In summary: economical design and interesting new feature sets do not oppose each other, but instead can amplify each other.

Chapter 7

Outlook: Semantic Challenges and Further Architectural Improvement Opportunities

7.1 Introduction

Work on a complex ISA extension such as ASF (and BeHTM in general) is never really complete. Shifting workloads, further work on decomposing / regularising the ISA interface and new use cases provide ample opportunity to extend, morph and reevaluate previous decisions on ISA design.

This chapter pays tribute to such forces of constant change and introduces work that has started, but not been completed during the course of my PhD. The points identified here have been presented at various transactional memory workshops discussing early transactional memory ideas: the decomposition of HTM and lock elision primitives in Section 7.2 has been presented at WTM 2013 [1] ; issues and solutions for handling access to high-quality time sources in Section 7.3 has originally been presented at WTTM 2010 [2].

Finally, several ideas were only published in the form of patent applications, Section 7.4 will sketch the issues solved and solutions that we envisioned. Finally, there is work that I have undertaken at my employers AMD and ARM that is not published at a conference or as a patent / patent application (yet). Sadly, I can therefore not share any information about that work, except the fact that there is more to come.

7.2 Decomposing HTM and Lock Elision Primitives

Speculative lock elision (SLE) essentially converts critical sections protected by a lock into transactions executing in parallel. In the Intel TSX proposal [303, 367], instruction prefixes (`ACQUIRE`, `RELEASE`) mark normal memory accesses as lock acquisition / release operations. For example, a simple test-and-set lock protecting a critical section looks as follows:

With the `ACQUIRE` / `RELEASE` primitives, the processor performs prediction to see whether it makes sense to elide and can convert the critical section into a transaction. Furthermore, accesses to the lock variable change: instead of acquiring the memory for writing, only a local modification is made; and the processor tracks the value history of the lock. If the lock is eventually transitioned back to the original

[1]Euro-TM Workshop on Transactional Memory (WTM 2013); Prague, Czech Republic, April 14, 2013; `http://www.eurotm.org/action-meetings/wtm2013/program`

[2]2nd Workshop on the Theory of Transactional Memory (WTTM); Cambridge (MA), USA, September 16, 2010; `http://lpd.epfl.ch/gramoli/wttm2/html/index.html`

```
reg <- taken
ACQUIRE exchange(reg <-> *lock)
if (reg != free) goto spin_and_retry
...
RELEASE store(free -> *lock)
```

free value, the processor effectively compresses this value history cycle into a *single read* of the free value.

The net effect of these changes is that multiple concurrent critical sections can locally "acquire" the lock concurrently and do not abort each other due to write-after-write conflicts. On top, the processor performs transactional execution of the critical section content, performing conflict detection and data versioning, to enforce isolation.

Unfortunately, these mechanisms (prediction, local stores, compression of store-chain cycles, and transactional execution) are only exposed as an integrated package. My goal is to make each of these components available as explicit instructions, following the RISC principle. The separate availability allows programmers to write their own elision mechanisms more effectively, and can be applied in other transactional algorithms.

Opt-In Lazy Versioned Stores We can treat local stores as lazily versioned stores: they keep the transactions linearisable, but do not eagerly send out write requests which could abort other transactions. Instead, they will send out their write requests (invalidating snoop messages) only at the end of the transaction when it commits. Lazy versioning may require additional hardware (for late sending out of stores) and may therefore be constrained in capacity. Also, depending on transaction characteristics, eager versioning may have beneficial effects on system performance due to exposing conflicts as early as possible. We therefore propose to add a `LAZY` prefix to allow the programmer to mark their stores as lazy versioning, making this an opt-in feature.

Squashing Cyclic Value Histories Secondly, tracking the value history can be used to essentially squash a sequence of writes (e.g. $store(A \to mem1)$, $store(B \to mem1)$, $store(C \to mem1)$ inside a single transaction to a single global $store(C \to mem1)$).

If that remaining store restores to the value initially *read* in the transaction, the entire store cycle can be dropped and compressed to that single load. We propose to mark the initial load with a `CYCLE` prefix to enable the HW tracking of the value history of subsequent stores.

Software-visible Hardware Predictor Finally, exposing the prediction mechanism to software can be achieved with the following ISA design: a `BRANCH_ON_PRED <target>, <id>` instruction branches to address `<target>` when the processor predicts true. The prediction is steered by software through positive / negative feedback instructions: `PRED_GOOD <id>` and `PRED_BAD <id>`. The processor uses the usual correlation mechanisms (branch history, address of branch instruction, correlation with other branches) to improve the prediction for `BRANCH_ON_PRED`, maximising the amount of `PRED_GOOD` feedback.

Putting it All Together Together with simple transactional memory primitives, the existing lock elision can be expressed as follows:

```
BRANCH_ON_PRED slowpath, 17
SPECULATE
if (CYCLE lock != free)
    PRED_BAD 17
    goto slowpath
LAZY (CYCLE) lock := TAKEN
```

for the entry, and the following sequence for ending the elided critical section:

```
(CYCLE) lock := free
COMMIT
PRED_GOOD 17
```

The result is a speculative lock elision primitive that is very flexible, because applications can control their abort handlers, and still get the benefits of the special handling of the lock variable. Selective lazy annotation for stores can enable higher throughput transactions, and similar manual code changes (moving stores in eager transactions as late as possible) have been suggested elsewhere [249, 282].

7.3 Safely Accessing Timestamps in Transactions

Many applications rely on time measurements for a variety of tasks. Depending on the characteristics of the measurement device (the *clock*), an observer may be able to reason about the order of events. There is a very intuitive notion of such time-based reasoning[3] which programmers rely on, although in the processor manuals, we have found little about the permissible deductions for the available hardware clocks. In particular, there is a lack of documentation regarding the interplay of concurrent clock accesses and order induced by other means such as the memory model. Practitioners needing to reason in this realm therefore rely on a blend of observed behaviour, online testing and documentation, for example in the Linux kernel [107, 239].

The lack of rigid specifications for standard processing is matched by the way transactional memory systems treat the subject. Although reasoning about atomicity, isolation and observed serialization / linearization order has been discussed prominently, we find that reasoning about clocks in transactional context is largely absent from the current research work. In this void, implementations are free to exhibit different behaviours of time sources in relation to transactional processing. We believe that with the recent commercial availability of transactional memory and lock elision it is necessary to integrate reasoning about time sources and transactions (and general shared memory communication) more tightly. As a base-line, and without loss of generality, we look at the properties of an ubiquitous clock on x86 microprocessors, the *Time Stamp Counter* (TSC), and its read-outs, which we will refer to as *time stamps*. We assume and explicitly formulate properties which we believe are intuitive and hold for TSCs, and then derive high-level assumptions about simple concurrency control constructs, such as mutual exclusion.

Subsequently, we find that lock elision [58], if it attempts to mimic mutual exclusion soundly, should restrict access to the TSC. Due to the close relationship between lock elision and transactional memory [34], we propose to adopt the semantics of time in mutual exclusion also for transactions. Being a new programming construct, transactions may define any behaviour they wish (much alike to the many forms of memory semantics in transactional memory systems [165]). We will define a set of semantics and show example code sequences to show the effects of weak and strong semantics on observable results.

Finally, practical considerations may have a huge impact on requirements of such a semantics, and on what is feasible to provide in real systems. To that end, we propose a solution that we believe is light enough to be implemented in hardware, but permits safe TSC accesses inside transactions.

Our contributions are the following: we formalise intuition about the interplay between memory and time-stamp order; we show how these orderng rules extend for plain synchronisation and are subsequently violated in naive lock elision mechanisms, allowing code to malfunction despite full memory isolation. Finally, we propose several mechanisms that provide strong *temporal* isolation, providing parallel execution for code that requires properly ordered time stamps.

[3]We ignore effects of the relativity theory, in particular, relativity of simultaneity in both this paper and our assumption about intuitive reasoning.

The remainder of this section is structured as follows: Section 7.3.1 provides a background of concurrency control mechanisms and time stamps, and Section 7.3.2 reviews their semantics. We then show critical interactions between time stamps and transactions in Section 7.3.3, and draft two implementations that properly handle these in Section 7.3.4. Sections 7.3.5 and 7.3.6 provide an outlook and summarise the work, respectively.

7.3.1 Background

Concurrency Control Critical sections traditionally operate on a strict mutual exclusion property: Instructions of two critical sections must not interleave if both critical sections are protected by the same lock variable. Recent literature calls this mode of operation Single Lock Atomicity (SLA) [224].

Database transactions provide *serialisability* [7], the strongest form of isolation. Briefly, transactions may overlap if there exists an equivalent execution in which no transactions overlap. *Strict* serialisability (and the similar *linearisability* [21]) is a stronger form of serialisability that restricts the serial order such that observed real-time order between temporally non-overlapping transactions is maintained.

Transactional memory [34] brings the notion of transactions to general-purpose systems. The large variety of semantics [165] differs mainly in how they deal with interleaving of memory accesses that are not part of transactions and accesses that are part of a transaction. Generally, weak and strong isolation are considered: *strong isolation* always orders memory accesses outside of transactions with accesses inside of transactions (treating them as implicit, one-instruction mini-transactions), whereas *weak isolation* does not enforce such order and usually specifies other, more complex semantics.

Transactional Lock Elision Transactional execution offers benefits over strict mutual exclusion imposed by critical sections, because transactions can be optimistically executed in parallel, as long as the specified isolation level is maintained. The idea to convert lock-protected critical sections into transactions and extract additional performance has been proposed in previous work on lock elision [58].

To *transparently* elide locks in existing applications, the semantics provided by lock elision must not be weaker than that of mutual exclusion. Otherwise, applications may crash or misbehave, for example causing data loss.

Access to Time Stamp Counters Applications must measure time and its progression, to determine durations of, order, and coordinate events. Computers provide time sources of varying quality. Without loss of generality, we will focus our analysis mainly on the CPU's time stamp counter (TSC), which has seen significant quality improvements during the last decade. We assume the TSC is a suitable real-time source,[4] and that applications (through libraries such as libc) rely on such behaviour. For brevity, and without loss of generality, we ignore the potential offset between different TSCs because it can be bounded by a small constant ϵ with initial calibration. Our results hold also in those cases, with some additional margins added to comparisons accounting for the difference. Furthermore, we do not consider TSC wrap-around.

On the AMD64 architecture, the TSC can be read with the `RDTSC` and `RDTSCP` instructions. We consider only `RDTSCP` because it is properly serialised with the instruction stream. We do not consider other clocks, because the TSC is the most frequently used fast, stable time source.

7.3.2 Semantics of Time Stamps

In this section, we will explore and formalise assumptions made about time stamps in existing code to guide our understanding of how time stamps should be treated in transactions. As outlined earlier, we

[4]This has been true for most x86 microprocessors since 2007.

```
        Initially : c  = 0
t1  = RDTSCP
c  := 1
                     lc  = c
                     t2  = RDTSCP
```

Figure 7.1: Dependent reads from the time stamp counter produce properly ordered time stamps: If $lc = 1$, then $t2 > t1$.[5]

rely on intuition and will generalise from small examples to guide for illustration purposes.

Traditional Code Memory causality and synchronised time stamps should allow us to reason about time stamp relations across multiple cores and application threads:

Definition 1 (Causal time stamp counters). A system that produces time stamps whose values reflect the order imposed by memory accesses and other ordering constraints in non-transactional code is called causal.

In Figure 7.1, two `RDTSCP` instructions are ordered by a memory dependence. We assume for this and similar examples[5] $t_2 > t_1$ always holds; generally, existing order such as memory dependence and program order also orders time stamps.

Critical sections implemented through locks serialise execution through memory dependencies. Therefore, for time stamps t_1 and t_2 read inside critical section CS_1 with $t_1 < t_2$, and t_3, t_4 read in CS_2 also ordered $t_3 < t_4$, and CS_1 and CS_2 being protected by the same lock variable, we know from *Causal TSCs* that either $t_2 < t_3$, or $t_4 < t_1$. In other words:

Definition 2 (Temporal mutual exclusion). For two critical sections CS_1, CS_2 protected by the same lock, the intervals spanned by the set of obtained causal time stamps $T(CS)$ do not overlap:
$[min(T(CS_1)), max(T(CS_1))] \cap$
$[min(T(CS_2)), max(T(CS_2))] = \emptyset$.

TSC-oblivious Transactions Because transactions are a new programming construct, they do not need to maintain strict compatibility with legacy code and its assumptions. Therefore, access to the TSC is treated differently in the various implementations of transactional memory: software TMs (STMs) usually do not track `RDTSC(P)` instructions and so may allow an application to infer temporal placement and overlap of the transactions. AMD's proposed hardware transactional memory, Advanced Synchronization Facility (ASF), [384] does not allow transactions to execute `RDTSC(P)` and aborts them if they try. Although this makes it impossible to detect temporal overlap for transactions, it limits the applicability of transactional programming and of lock elision.

Intel's recent Transactional Synchronization Extensions (TSX) [303, 367] do not restrict access to the TSC in both transactions and elided critical sections. This creates a semantic gap between lock elision and code using locking, but because the elision does not work fully transparently,[6] software can be exposed to the changed semantics [285].

The Need for Stronger Semantics We pointed out earlier that it is desirable to elide critical sections to enable parallel execution of non-conflicting critical sections and thus reduce the issue of sequential bottlenecks. Hardware will track memory accesses of elided critical sections and ensure they correlate to a sequential execution by tracking conflicting accesses. However, as we will show in the next section, this is

[5]We denote threads as columns, and time progressing downward. Interleaving of events corresponds to the interleaving of rows. Reads into local variables are marked with `=` and stores to memory with `:=`.

[6]Applications need to use annotated instructions to acquire and release the lock variable and enable elision, and can query whether they run in a transaction / elided critical section with the `XTEST` instruction.

```
                   TX2.begin
TX1.begin
t1 = RDTSCP
                   t = RDTSCP
t2 = RDTSCP
TX1.end             ...
                   TX2.end
```

Figure 7.2: Overlapping time stamp values violate SLA.

```
          Initially : c = 0

TX1.begin
                     TX2.begin
                     t2 = RDTSCP
c := 1
t1 = RDTSCP
TX1.end
                     lc = c
                     TX2.end
```

Figure 7.3: Two transactions can exhibit contradicting ordering if $lc = 1$ and $t_2 < t_1$.

insufficient if applications make use of the TSC inside critical sections. Applications may infer concurrent execution of critical sections that were supposed to execute strictly sequentially. Such mismatch between expected behaviour and implementation breaks transparency of the elision and can lead to crashes, or other misbehaviour.

In addition to supporting stronger semantics for the lock elision case, we also consider a stronger semantics incorporating TSC accesses for (hardware) transactions. Providing the same semantics in both modes makes sense: (1) using hardware transactions as a drop in replacement for locking, and (2) providing a well-known semantics for TSC usage in transactions that is easy to understand and reason about.

7.3.3 Semantic Issues of Time Stamps in Transactions

In this section we illustrate problematic orderings of transactions and the use of time stamp accesses from within them. We aim for transactions and elided critical sections to have semantics equivalent to proper mutual exclusion; and will therefore use the term transactions also for elided critical sections. We will illustrate corner cases using transactional annotations (instructions `TX.begin` and `TX.end` delimit a transaction) and see how they behave if SLA was used. With that we mimic the intuitive mutual exclusion semantics and derive the required semantics for TSC accesses in lock elision and transactions.

Simple Overlap Case In Figure 7.2, the case $t_1 < t < t_2$ directly violates *temporal mutual exclusion* in SLA – the result for time stamps in traditional critical sections obtained in Section 7.3.2. We would like transactions to provide the equivalent:

Definition 3 (Temporal Isolation). If two transactions T_1 and T_2 read from the TSC, with T_1 reading multiple time stamps in the interval $[t_1, t_2]$ and for each time stamp t read in T_2, it must not be the case that $t \in [t_1, t_2]$.

Order Mismatch: Memory vs. Time Stamps In addition to detecting temporal overlap through time stamps, two transactions may observe mismatches in the respective memory or time stamp orders. Figure 7.3 shows an execution that orders transaction `TX1` before `TX2` through memory; however, because $t_2 <$

```
          Initially : c = 0

                    TX2.begin
                    t2 = RDTSCP
t1 = RDTSCP
c := 1
                    lc = c
                    TX2.end
```

Figure 7.4: Failure of strong temporal isolation if $lc = 1$ and $t_2 < t_1$. Instructions on the left do not execute from a transactional context.

t_1, the time stamps indicate the reverse order. Proper SLA semantics would not allow such contradicting orders under *Causal TSCs*. We therefore extend temporal isolation to agree on the order of transactions with the order imposed by memory accesses:

Definition 4 (Consistent Temporal Isolation). If two transactions T_1 and T_2 are ordered such that T_2 observes T_1's memory accesses, for all time stamps t_1 read in T_1 and time stamps t_2 read in T_2, $t_1 < t_2$ has to hold.

Weak and Strong Temporal Isolation If applications accesses the same data inside and outside critical sections, they do not *explicitly* establish additional ordering between the accesses outside of critical sections and those inside. In those cases, order can be created only through reasoning with the underlying memory semantics or transitive ordering with earlier synchronisation. In particular, if accesses to shared data happening outside critical sections are not ordered through other dependency chains with accesses to said data from within critical sections (usually called a data race), the involved accesses are subject to complex, sometimes even undefined behaviour.

With transactional memory, however, some systems provide *strong* isolation which properly orders transactions with respect to all overlapping memory accesses outside of transactions, usually by assuming that accesses outside of transactions are one-instruction mini-transactions. Systems that do not provide such isolation, but only order transactions among each other, provide *weak* isolation.

We observe a similar interaction with TSC accesses, extending the weak / strong (memory) isolation property to weak / strong *temporal* isolation (WTI / STI): Figure 7.4 highlights the interaction. Because there is only a single transction, the example clearly does not violate either SLA, nor does it produce overlapping time stamp intervals. It also does not violate purely memory-based strong isolation semantics, because TX2 is ordered after the non-transactional store from a memory perspective. We would like to enable the intuitive combination of strong memory isolation and *Causal TSCs* through strong temporal isolation.

Definition 5 (Strong Temporal Isolation). All time stamps t_{tx} read within a transaction T must reflect all observed memory order T has been subject to. In particular, if through any order / dependency chain T is ordered after a TSC access (not necessarily inside a transaction) reading time stamp t_{nontx}, with strong temporal isolation $t_{nontx} < t_{tx}$ holds for all such accesses and time stamps.

Under weak temporal isolation, such additional conditions on time stamps read within transactions are not enforced.

The Need for Strong Temporal Isolation Strong temporal isolation restricts execution more than a system that provides only weak temporal isolation because it forbids executions such as the one in Figure 7.4. However, SLA does not inherently order critical sections and code outside of critical sections, so it may seem sufficient to provide only weak temporal isolation for transparent lock elision (and transactions). However, if we modify the example in Figure 7.4 slightly to obtain that of Figure 7.5, we find

```
        Initially : c = 0

                    TX2.begin
                    t2 = RDTSCP
t1 = RDTSCP
TX1.begin
TX1.end
c := 1
                    lc = c
                    TX2.end
```

Figure 7.5: Transparent lock elision requires strong temporal isolation: with SLA, if $t_2 < t_1$ then $lc = 0$; however, with weak temporal isolation $lc = 1$ is possible.

otherwise. Using locks (SLA semantics) instead of transactions, we can deduce: if $t_2 < t_1$, `TX2` must also read the old value of c because it must have executed entirely before the empty transaction `TX1` which in turn executed before the update to c. Strong temporal isolation will enforce the same implication to hold, but the depicted schedule is possible with weak temporal isolation. Therefore we conclude:

Observation 6 (Weak Temporal Isolation insufficient for SLE). Transparent transactional lock elision with support for accesses to TSCs requires a stronger semantics than weak temporal isolation.[7]

7.3.4 Supporting Time Stamps in Transactions

In the previous section, we extended the semantics outlined in Section 7.3.2 to transactions. We furthermore showed that existing transaction systems with unrestricted access to time stamps can violate these rules and intuition. We will now construct systems that are able to detect and avoid violations of the desired temporal semantics. Our focus is on solutions that work on-line, and require only small additional effort. We strive for solutions that can be implemented in microprocessors providing transactional memory / lock elision, but are applicable to software transactional memory solutions and other clocks, as well.

An easy way to enforce any of the presented temporal semantics is forbidding all access to time stamp counters from within transactions and elided critical sections. Hardware transactional memory proposals such as AMD's ASF work like that and abort all executed transactions that try to use the `RDTSC(P)` instruction. However, because time stamping is employed in many applications, it is desirable to find less rigid solutions that provide temporal isolation.

In summary, the desirable semantics from the previous section can be enforced through the following *temporal isolation rules*: (1) the time stamp order needs to agree with the order established by the memory accesses in transactions and non-transactional code; and, (2) time stamp intervals of transactions / elided critical sections must not overlap.

Single TSC Access Restricting transactions to access the TSC at most once removes the problem of overlapping time stamp intervals (enforcing Rule (2) above). A simple implementation could therefore track locally that transactions do not read from the TSC twice, and abort the ongoing transaction otherwise.

The simplest solution to enforce consistent time stamp and memory order (Rule (1) above), *Single-AbortAll*, is to allow only a single transaction access to the TSC and abort all other concurrently overlapping transactions (e.g., by adding a dedicated memory location TSC_A to each transaction's read set and treating `RDTSC(P)` as a write to the proxy location). Going back to the example execution of Figure 7.3, `TX1` would abort immediately at the acquisition of t_2, regardless of `TX1`'s content.

[7]Strong temporal isolation, for example. In future work we will pursue whether there are semantics weaker than STI that would also enable transparent SLE.

A more selective approach is *Single-AbortTSC*, which aborts only those concurrent, live transactions that have or will eventually access the TSC. That can be achieved by recording remote TSC usage, and handling the conflict only if the local transaction has already accessed the TSC, and if not, letting future `RDTSC(P)` instructions in this transaction check whether their enclosing transactions have seen a remote TSC access. Compared to Single-AbortAll, the advantage is that transactions that do not access the TSC need not be aborted. The execution in Figure 7.3 would only need conflict handling when `TX1` obtains t_1. If `TX1` had not used `RDTSC(P)`, it could continue execution.

Multiple TSC Accesses Allowing a transaction to read from the TSC multiple times allows transactions to observe the passage of time.[8] Similar to Single-AbortTSC, in *Multi-AbortTSC* transactions will not abort all other concurrently running transactions at the first read of the TSC, but rather ensure that no overlapping time stamp intervals can form.

In this case, each transaction checks at the second or later read from the TSC that it has not received any notifications from remote TSC accesses; otherwise, a conflict exists. This case needs to be handled by a conflict resolution policy, for example by self-aborting on the second local `RDTSC(P)`, or by aborting all other time stamp-using transactions.

The advantage of this technique is that transactions can execute in parallel, and each can access the TSC multiple times, as long as the intervals formed by each transaction's first and last accesses to the TSC are disjunct. In the Figure 7.2 example, the conflict would be detected at the read of t_2 because of the concurrent TSC read to t . If the reads for t_2 and t would have been ordered the opposite way (and then also $t_2 < t$), no conflict would exist.

Enforcing order: Again, in addition to enforcing disjunct time stamp intervals, we need to make sure that these intervals are ordered consistently with all other orders – in particular those imposed through memory accesses. Revisiting the simple example in Figure 7.3: so far, the algorithm Multi-AbortTSC does permit the contradicting order. Consistent ordering between time stamps and memory accesses can be achieved by ensuring that `TX2` commits before `TX1` because it read the earlier time stamp. Waiting at `TX1`'s commit point for a *can_commit* message from `TX2` will ensure consistent temporal isolation because the memory conflict on c caused by the contradicting time stamp order will be visible to `TX1`, see Figure 7.6.

These commit ordering messages do not need to broadcast, because `TX2` knows that `TX1` has acquired a later time stamp through the tracking of accesses to the time stamp counter. `TX2` can therefore signal `TX1` (and all later) transactions directly that they must wait for a commit signal, and when they can commit.

Waiting alternatives: Instead of `TX1` stalling the CPU at commit, it may be possible for it to continue execution of instructions behind the transaction transactionally, essentially increasing the length of the transaction by adding the following, non-transactional instructions to its transactional tail. That way, useful work can be done, reducing the performance penalty incurred by waiting. Of course, this may lead to additional conflicts (due to additional memory accesses adding to the working set), and additional transactional TSC accesses that need special handling.

Another option is to put the core in a low-power state and wait for the special *can_commit* signal from `TX2` (similar to the low-power mode that can be entered with the `MONITOR` / `MWAIT` instruction combination [257]).

[8]Some researchers have argued that observing the passage of time would violate the atomicity of transactions, implying that atomicity is "instantaneousness". However, we find that neither serialisability, linearisability nor transactional isolation prohibit the observation of time, but instead find practical use for applications wanting to measure execution time of code executing inside transactions / elided critical sessions, for example.

```
                 Initially : c = 0

TX1.begin
                        TX2.begin
                        t2 = RDTSCP
c := 1
t1 = RDTSCP
                        <Before TX1>
<After TX2>
 ...
TX1.end?
<wait>
   |                    lc = c
<Conflict c>
   |                    TX2.end
(TX1.end)
```

Figure 7.6: Ordering commits between transactions to match time stamp order will ensure that disagreeing observations (memory orders) cannot be established due to abort.

Handling Non-transactional TSC Accesses Strong temporal isolation requires that TSC accesses outside transactions need to participate in the conflict detection mechanisms. However, we may want to treat conflicts with non-transactional TSC accesses differently to ensure progress and minimal obstruction for non-transactional code. We therefore suggest biasing conflict resolution in favour of non-transactional TSC accesses, by always aborting the concurrent transactions with which the non-transactional `RDTSC(P)` conflicted (instead of retrying / delaying the non-transactional access).

7.3.5 Outlook

Although we believe our examples and implementations cover all critical interactions, and thus provide identical semantics for transactions and lock elision to fully sequential mutual exclusion, we have not yet proven our solutions to be correct or our restrictions to be minimal.

We are particularly interested in the applicability of our proposed implementations. Clearly, our system is more permissive than AMD ASF's strict ban of TSC reads inside transactions (and more transparent than Intel's TSX lock elision), but we still need to abort and serialise a significant number of transactions. We believe there is a limit to which one can make parallel execution behave similarly to sequential execution, but some applications may just not care. For example, applications may use time stamps only to measure durations, instead of using them to establish order of events. Automatic inference of such properties seems very hard; therefore, we have restricted our analysis to provide as much performance as possible with strong temporal isolation. A viable alternative is to leave the decision to software – for example, by offering an explicit choice between strong and weak temporal semantics with multiple types of transactions and / or time stamp accesses. This new freedom opens up a new space for looking into the interactions between different types of accesses and exactly how much each needs to be weakened to provide compelling performance, while remaining useful for applications.

7.3.6 Conclusion

In addition to communication through memory, time stamps provide another way for parallel code to coordinate. Therefore, time stamps need to be considered by systems that control parallel execution such as transactional memory and lock elision. We expect that the wide availability of systems in the near future will only amplify this need and make this a promising direction for future research. Existing critical sections provide very intuitive semantics of temporal isolation, which we have not found to be provided by most STM and HTM solutions. In this section, we have identified the issue of time stamp order and have shown with multiple examples how applications can observe time stamps inconsistent with the isolation level or with the order observed from memory accesses. We formalised the intuitive reasoning steps and extended them to transactions and lock elision mechanisms, advocating that they both should be equally strong. We drafted various implementations that will provide weak and strong temporal isolation without the need to fully serialise execution.

7.4 Further HTM Ideas

Industrial research has to strike a careful balance between publishing and keeping findings proprietary. One step to publishing research work is to patent ideas and mechanisms, and then publish them in research literature.

During my time at AMD, me and my collaborators developed several ideas that have not been published in academic conferences, but some of which have been published as patents and patent applications. Others still remain in the process of being evaluated for patenting / being drafted as patents.

This section briefly discusses the scientific problem and core of the solution in plain terms, and links to the respective patent documents for reference.

7.4.1 Roll-forward Mode

During the design of the initial ASF specification [186] in 2010, we already identified the synchronous nature of HTM aborts as an unfortunate side-effect, especially with sequences of non-tx operations which should be completed before handling the abort. The solution I present earlier in this document (transactional resurrection, Section 6.4) proposes to take the synchronous abort, and then continue the aborted transaction / sequence of non-tx code until the abort can be handled more easily asynchronously.

Our original proposal to handle such cases was to propose a *roll-forward* mode for ASF[9] which we planned to publish as an update to the existing ASF specification. The roll-forward mode is started with a special flavour of the `SPECULATE` instruction; inside the roll-forward transaction, aborts will only set a flag and continue execution of the transaction. Transactional stores check the flag and if an earlier abort was detected, they will *not perform the store*. There is also a new instruction to check for a recorded abort (`VALIDATE`) and allow applications to defer handling of the abort to a more suitable time.

Transactional stores checking the flag simplifies the application logic, as otherwise, there would be a race between the abort happening and the application checking for the validity of the transaction only before / after the store, but not at the same time.

One challenge in this mode is the mixed nesting of roll-forward / roll-back transactions and tracking their state when a nested transaction ends: is the outer transaction of the roll-forward or roll-back kind?

7.4.2 Nested Abort Handlers

Nesting transactions in ASF (and most other BeHTMs) is conceptually simple flat nesting which keeps a nesting count and flattens all nested transactions into the outermost enclosing transaction. As a side-effect, only the outermost transaction will see the abort.

This is a nuisance for keeping statistics of transaction aborts, because if the innermost transaction is responsible for the conflict and could provide a separate, non-conflicting code path, it needs to know of the abort in order to change the selected code path.

[9]USPTO patent no 8,621,183

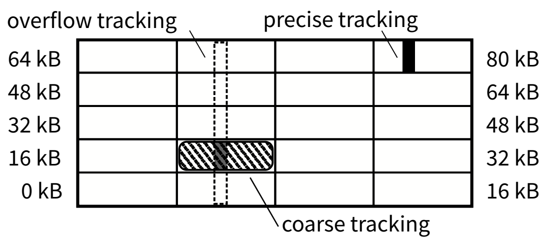


Figure 7.7: Tracking transactional read sets in two dimensions using a combination of different granularity mechanisms for additional capacity and precision. Precise tracking uses the normal cacheline-granularity HTM mechanism; overflow tracking marks the entire set as transactional when a read set entry was evicted from it; in combination with coarse tracking, precision loss is reduced because *both* structure need to indicate a conflict.

In addition, if nested transactions perform non-tx operations, they might want to undo them with a registered undo-action in case of an abort. The outermost handler, however, does not know of the undo-actions of the nested transaction which may be behind a library interface.

In our work on nesting abort handlers[10] we show how to reverse the handling of aborts: instead of invoking only the outermost abort handler, hardware invokes the *innermost* handler and a combination of hardware and software tracking (similar in technique to the transactional resurrection proposal described in Section 6.4) ensures that the innermost handler knows about the nesting hierarchy of abort handlers.

For that, nested `SPECULATE`s will push link information to the previous abort handler on the stack so that software can then perform a simple `POP` / `RET` sequence and link to the outer handler from the inner. Pushing the link information out to memory ensures that hardware does not have to provide for a large buffer tracking this information and that the information can persist a context switch so that the abort handler hierarchy can be walked when the application is then later switched to again.

7.4.3 Two-dimensional Tracking of Large Objects

One technique to extend the HTM's capacity for tracking the read set is to use a simple overflow mechanism: once a tx-read cache line is displaced from the conflict detection hardware structure (for example the L1 data cache), the entire set of the cache is marked as TX.R. That way, all remote conflicting memory snoops that access this set will cause a transactional abort. The addresses that map into the set depend on the cache geometry, and are distributed regularly in memory: usually[11] every $cache\_capacity/cache\_associativity$. For example, for a two-way set-associative L1 data cache with 32 kB capacity, the overflown set would (falsely) detect conflicts with all memory locations with an offset of integer multiplies of 16 kB.

Overall, this mechanism trades tracking precision over tracking capacity. Another mechanism that performs the same trade-off differently is using a more coarse grained tracking granularity; for example, instead of tracking cache-line-sized memory regions (64 byte), one could track larger blocks of 4 kB to increase the total tracked capacity.

In our invention[12], we propose to combine the two tracking mechanisms, i.e., once a cache-line is displaced from the cache and the entire index will conflict, a secondary tracking structure with larger granularity (such as the TLB with 4 kB page sizes) can be used to reduce the number of false positives caused by the aliasing of the equidistant addresses.

As is apparent from Figure 7.7, this is effectively tracking the transactional read set in two dimensions.

[10]USPTO patent application 20140181480
[11]if the lowest possible bits are used for indexing into the cache
[12]USPTO 8,612,694

7.4.4 Tracking Large Objects in the TLB / Page Tables

Using a separate, purpose-built tracking structure with different granularity is straightforward, but requires an additional hardware widget. Using the TLB directly has challenges because it is usually queried with *virtual* addresses and is also not consulted on external snoop messages. In two separate inventions[13], we have shown how to use the actual page-table data structure in memory and the fact that the AMD64 architecture maintains in hardware accessed and dirty bits for every page-table entry.

In a nutshell, the algorithm uses the normal ASF conflict detection mechanism (e.g. the L1 data cache), but instead of tracking conflicts on (all cache lines of) the accessed object, the conflict detection mechanism will monitor changes to the respective entry in the page-table (the last level PTE). Thanks to the hardware maintained accessed and dirty bits, a remote writer will need to write to the PTE to update the dirty bit, and thus cause a proxy conflict on the PTE with the original transactional reader.

In summary, this mechanism then can track accesses on page-granularity, supporting huge read sets with very little amount of actual conflict detection hardware; of course at the cost of losing tracking precision and potentially inducing false conflicts and aborts. The two inventions deal in more detail with challenges such exact marking / unmarking of the accessed / dirty bits.

7.4.5 Reducing Live-Lock by Ordering Transaction Memory Accesses

Two transactions can have mutually overlapping working sets and thus one of them may conflict abort, restart and then abort the other transaction, leading to the same, mirrored cycle. A system in such a condition is making progress on some level (the CPUs keep executing instructions), but failing to make progress on some higher level of abstraction (the transactions / operations they represent never complete)–a *live lock*.

There is a large body of related work on solving these issues, usually with specific contention management policies, delaying the restart of conflicting transactions probabilistically and also mechanisms for stricter scheduling of transactions.

In HTM implementations that use the cache coherence protocol for conflict detection, one option is to stall answers to incoming conflicting snoop messages in order to throttle the conflict, abort, restart, conflict loop and increase chances of the local transaction to commit in time. Unfortunately, stalling snoop responses in such a way may impede progress guarantees of the underlying coherence protocol, sometimes through long, obscure, system-wide resource dependencies; in other more obvious ways where two transactions decide to not respond to each others' snoop requests and thus deadlock the system.

In our invention[14], we carefully establish a non-cyclic order between memory accesses (for example by physical address), and allow transactions / CPUs to block snoop responses if their accesses are ordered in accordance. That way, in the mutual conflict case, one of the two transactions will be able to "lock" its working set and commit the transaction, clearing the livelock situation.

7.5 HTM Product Experiences

Most of my work in this PhD has been performed *before* HTM implementations were available commercially. Even then, a debate over the actual value of the feature arose, and it was clear that HTM is not a "silver bullet" for all synchronisation / parallelisation challenges [155, 246, 252].

IBM and Intel are currently leading in terms of having HTM implementations available, and have had these for considerable amounts of time (since 2013). Intel especially are pushing HTM as a differentiator

[13]USPTO 8,943,278 and 8,914,586
[14]USPTO application 20140181480

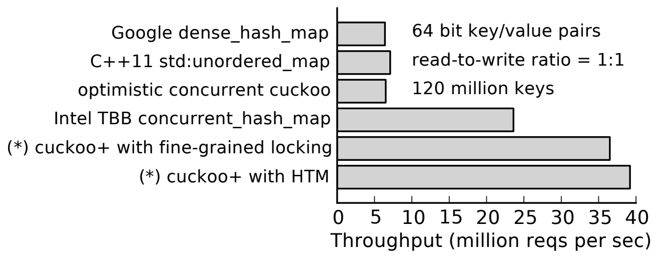


Figure 7.8: Throughput of different hash tables on a 4-core system. From [320].

to competing platforms that do not have HTM (most notably AMD and ARM), and claim significant performance gains in SAP HANA's radix tree and DPDK's hash table, of 2.2x and 11x respectively [335, 348]. However, on closer inspection, most (if not all) of these spectacular gains can be achieved by using different, already concurrent data structures, and / or converting the existing single-global-lock version to use fine-grained locking.

Li et al investigate the DPDK cuckoo-hash table in more detail in [320]. They break out the benefits of the different optimisation separately, and also convert the data structure to fine-grained locking. In Figure 7.8, it is obvious that that performance varies significantly between different hash tables. Comparing the performance of the fine-grained version and the optimised TSX version, it becomes clear that the most significant (and arguably only) benefit of HTM is the simplification / avoidance of the work required to convert the data structure to fine-grained locking / lock-free operation. The authors note on that point:

> Our results about TSX can be interpreted in two ways. On one hand, in almost all of our experiments, hardware transactional memory provided a modest but significant speedup over either global locking or our best-engineered fine-grained locking, and it was easy to use.
>
> ...
>
> On the other hand, the benefits of data structure engineering for efficient concurrent access con- tributed substantially more to improving performance, but also required deep algorithmic changes to the point of being a research contribution on their own.

With absolute performance removed from the list of HTM benefits (as opposed to the very valid performance vs application engineering trade-off), the other key remaining aspect of TM is *composability*. Going back to the detailed work in [320], however, we find that HTM performance of the application improved significantly only with significant *code motion* of application code out of the critical section / transaction. Such modification, however, loses the benefit when the entire macro data structure operation including the out-of-transaction prefix / suffix is part of an outer transaction. In my earlier work on ASF 1, I have speculated about possible ways to compose such operations by extracting and merging the prefix and transactional part of the operations separately [158, 159]. In the meantime, several authors have refined and formalised that concept [240, 314, 346], under such concepts as Consistency Oblivious Programming, Partitioned Transactions, and Optimistic Transactional Boosting.

With the remaining valid feature of TM being simpler development of fine-grained concurrent data structures, one should note that the overall complexity in the system remains, but is pushed into the hardware layer. This can for example be seen that Intel had to disable their HTM in the first *three* generations of products due to errors in the implementation [365, 366].

7.6 Summary

Despite significant work from both industry and academic researchers, and several commercially available implementations, transactional memory still has many open questions and available optimisation tweaks. In this chapter, I presented a small selection of work that I have undertaken towards understanding semantical challenges when using HTM and reasoning based on observing synchronised clocks, further added features and regularised ISAs to ease manual synthesis of flexible lock elision primitives, and further ISA extensions (nested abort handlers, roll-forward mode) and microachitectural improvements to provide better performance and larger capacities for HTM implementations.

In accordance with the industry research pipeline (research, patent, productise, publish), some of these ideas are only available publicly through patent applications and granted patents, while others are still in the patenting process and consequently cannot be discussed in this public thesis document. Still, despite a lot of work of me and other academic and industry researchers in the field mostly since 2006, there are still uncovered / unpublished extensions and optimisations twelve years later in 2018, and my employers (both AMD and ARM) continue the investigation; and I am sure that other companies are too.

Chapter 8

Conclusions

The key thesis of this work has been that one needs a detailed hardware substrate and ISA description to understand corner cases, feasibility, cost, and value of HTM implementations. Furthermore, I stipulated that despite and because of the higher level of detail, interesting solutions to both instruction set design and microarchitecture for HTM would be possible and new challenges could be uncovered–leading to solutions different from those presented in academic state of the art.

8.1 Summary

In the previous chapters, I have presented a summary of the state of the art in transactional memory and simulation (in Chapter 2); presented a production level ISA extension for HTM–AMD's Advanced Synchronization Facility–with background information and justifications for design decisions, challenges, and changes made due to experience from iterating through a full executable model, compiler, and real application stack (in Chapter 3). ASF provides full HTM functionality, but provides some additional features (limited capacity / progress guarantee, non-transactional accesses) and differences to "text book" style designs (no full register checkpoint). Then, I show various implementation options for that ISA extension in realistic and complex CPU cores that take into account interactions with the relevant CPU features, such as out-of-order execution, misspeculation, and complex memory hierarchies (Chapter 4); followed by a summary of ASF use cases and performance evaluation experiments, and a detailed tour through the simulator and challenges unique to the simulator implementation of ASF (in Chapter 5). Finally, I show new use cases and extensions to ASF, namely building communication channels on top of ASF and a mechanism to use the limited register checkpoint for transactions that can be resurrected after they have been aborted (Chapter 6); and present further extensions: decomposing lock elision primitives, roll-forward mode, nested abort handlers, two-dimensional conflict tracking, and using the TLB to track larger objects, and challenges: handling time sources as an implicit communication channel between elided critical sections (in Chapter 7).

My main contribution to the state of the art is the detailed level of microarchitectural, ISA, and system-level understanding for HTM that I have gained and made available to the research community through my work on the PTLsim and Marss86 simulation platforms [262, 304]. A further software artifact that I contributed to is the DTMC compiler toolchain; mainly through testing, and bug fixing thanks to detailed visibility into the whole system state in simulation. Additionally, my extensions to the Oracle Hotspot JVM uncovered several challenging real-world interactions that would break typical lock elision approaches [290]. These artifacts have been one leg of the foundation of the VELOX project that has taken a holistic application to transistor view of transactional memory.

From a commercial perspective, my main contributions are certainly the significant number of microarchitectural HTM implementation variants, their evaluation, and associated patent filings with AMD (27 patents pending, 19 patents granted as of writing), and the detailed ASF ISA extension that we published [186] (also available in Appendix A).

Academically, many of my contributions are part of papers at top-tier conferences and high-class relevant workshops. On top of those publications that I (co-)authored, several other publications use the ASF implementation in the simulator for further experimentation. The full list of publications that I have worked on is:

- Hardware Acceleration for Lock-Free Data Structures and Software-Transactional Memory (EPHAM 2008 [158], Appendix B.1)
- ASF: AMD64 Extension for Lock-free Data Structures and Transactional Memory (MICRO 2010 [214])
- Evaluation of AMD's Advanced Synchronization Facility Within a Complete Transactional Memory Stack (EuroSys 2010 [213])
- The Velox Transactional Memory Stack (IEEE Micro Journal [210])
- Implementing AMD's Advanced Synchronization Facility in an Out-of-Order x86 Core (TRANSACT 2010 [220], Appendix B.2)
- Compilation of Thoughts about AMD Advanced Synchronization Facility and First-Generation Hardware Transactional Memory Support (TRANSACT 2010 [215], Appendix B.3)
- Sane Semantics of Best-effort Hardware Transactional Memory (WTTM 2010 [221], Appendix B.5)
- From Lightweight Hardware Transactional Memory to Lightweight Lock Elision (TRANSACT 2011 [254], Appendix B.4)
- Delegation and Nesting in Best Effort Hardware Transactional Memory (SPAA 2012 [274])
- Safely Accessing Time Stamps in Transactions (WTTM 2012 [263], Appendix B.6)
- Between All and Nothing–Versatile Aborts in Hardware Transactional Memory (SPAA 2013 and TRANSACT 2015 [289, 337], Appendix B.7)

As already reviewed in Section 1.4, these are largely split into three phases; (1) baseline ISA, microarchitecture, and evaluation with full TM stack in 2010; (2) use cases and extensions (several of which did not turn into academic papers) until 2012, when the AMD Research office dissolved; and then further extensions with improved simulator models and extended use cases (2013 - 2015).

Comparing that to the trend in the field, this was after several of the seminal STM and HTM papers, but that time was not squandered, but instead allowed me and my collaborators to gain deeper insight into TM and related CPU architecture and micro-architecture concepts and challenges. Academic interest in HTM shifted, and commercial TM implementations became available, yet they were usually of quite simple nature (and often offering a single differenciating feature on top) around 2013. Overall, most of my contributions conincide with the peak of academic interest in TM, see Figure 2.2.

8.2 Thesis Evaluation

Reflecting back on the initial thesis of my work, my contributions, and the state of the art, my impression is that the thesis did hold; I have shown interesting extensions to baseline HTMs that have a small implementation footprint. For myself, I got a much higher level of appreciation for the complexities already in today's systems without HTM, and contributed to a detailed microarchitectural proof-of-concept and evaluation platform and ISA specification. During the course of the creation of each of those, we faced several non-obvious corner cases, precisely because we were thinking and experimenting on a very detailed level and had great applications and application experts that were stress-testing our systems. Therefore, I would strongly argue for the success of methodology of a high-level of detail, executable artifacts, and cross-stack understanding and collaboration.

8.3 Critical Reflection

Looking back at my experiences throughout my thesis work, they are very positive to the largest extent. Starting with the trust and flexibility of my thesis advisor, Prof Christof Fetzer, who was open to this somewhat unusual arrangement, and to the direction and trust I received inside of AMD thanks to my colleagues Dr Michael Hohmuth, Sherry Hurwitz, and Dr Martin Pohlack. Additionally, a significant amount of in-depth computer architecture knowledge came through a very collaborative atmosphere inside of AMD, both inside the Dresden office, but also globally across sites. Conversely, the collaboration with the VELOX researchers and the general Systems Engineering group at TU Dresden taught me a lot about transactional memory algorithms, compilers, and distributed systems. Finally, the VELOX project provided a great seed for connections across industry groups and universities around HTM, but also wider computer architecture, algorithms, and theory researchers.

With a lot of light, there comes a risk of shadow, too; I think the biggest impact on my work was the AMD Dresden office closing at the end of 2012, due to the strained financial situation at AMD. That caused a significant disruption to our work and meant that we could not publish and pursue some ideas that we were evaluating–some of these have made it into Chapter 7, however. One of those is the Oracle Hotspot ASF port, and a more detailed evaluation of it.

Looking back at my undergraduate degree[1] compared to what I know now after my PhD, I would have liked to learn more about computer architecture in my undergraduate degree–I think with the cfaed centre, the closer collaboration with the Faculty of Electrical Engineering, and a new chair for processor design, this will improve for future students at TU Dresden. Similarly, while the VELOX project grouped together many researchers from across the stack, in some cases it has only been during the write-up of my work that I have come across and appreciated work that has been done concurrently in the project, especially at BSC / UPC and Chalmers University. Getting young researchers to collaborate more closely across geographic regions and across layers in the stack is a challenge, but will undoubtedly increase the depth of understanding and education.

Another unfortunate aspect is that the VELOX website[2] has been offline, and several of the toolchains released through the VELOX project are not available publicly elsewhere, or hard to find[3]. Pushing open-sourced projects to reliable open-source hosting providers, such as GitHub, bitbucket, or GitLab would certainly keep code and its history available for future researchers, past the projects due date.

Finally, I personally underestimated the amount of effort, time, and mental focus required to condense my research work into this thesis document. Alongside starting a new job (at Arm Research), moving places to live (Dresden, Germany to Cambridge, UK), and buying a house, it has taken too much calendar time to complete the write-up. Partially due to my arrangements as "external" researcher and largely informal rules, writing up remained a low priority item on my to-do list. Some universities (I am now

[1]Diplom-Informatik (Masters in Computer Science) at TU Dresden
[2]`http://velox-project.eu`
[3]The DTMC compiler has been forked and extended at `https://github.com/basicthinker/PTMC` as the Persper Transactional Memory Compiler.

collaborating with) enforce checkpoints with a periodic write-up (first year report largely consisting of a state-of-the-art analysis, further reports adding research results incrementally)–essentially leaving the final write-up to a research update and refinement pass. I would encourage thesis advisers at TU Dresden Computer Science Faculty to consider such a regular model to support their PhD students.

8.4 Current State of the Eco-System

In the time I have been working on the topic of HTM (2007 to 2018), many aspects have changed. The number of publications at computer architecture and related conferences investigating transactional memory has reduced significantly, following the typical "hype curve" model. Several commercial implementations of HTM are now available, in all IBM architectures and the mainline performance Intel cores (IBM BlueGene/Q, IBM zSeries, IBM Power 8, Intel Haswell, Intel Broadwell, Intel Skylake, Intel Kaby Lake, Intel Coffee Lake); yet a significant fraction of those (Haswell, Broadwell, Skylake) had to be disabled due to implementation bugs ("errata"). Especially Intel has been marketing their TSX implementation aggressively with significant performance improvements quoted for SAP and DPDK workloads [348], and Intel-funded support for TSX in the glibc library [273, 317]. Production level compiler support in GCC for transactional memory is available, partially as a result of the work undertaken in the VELOX project.

Despite the (partial) availability of silicon and tool chains, there does not seem to be a great push towards using TM in applications; most use-cases use HTM for some form of (semi-transparent) lock elision. With a push to higher-level programming languages and abstractions, however, there may well be a good chance for using TM more in language runtimes, and query execution engines.

Going forward, I think it is clear that TM is not a silver bullet, but instead a powerful tool in the application programmer's toolbox. With more stable market penetration of (functioning) hardware, I would expect adoption to rise when application programmers want to unlock parallelism in their code through optimistic concurrency control. While the benefit of blindly and transparently applying TM and/or lock elision will be small, I strongly believe in the greater benefit of *incremental return*: changing code in transactions / elided locks to be more transaction friendly (removing conflicting global counters, removing system calls, looking after false conflicts) is not free, but produces instantly better performance. Restructuring code to either use fine-grained locking or lock-free algorithms often carries a higher upfront investment cost until performance improvements are seen, and furthermore, often complicates reasoning about the correctness of the code significantly.

As for market penetration of HTM, there is work at the other two big CPU vendors, AMD and Arm, on HTM; so a future where HTM is a standard feature of general purpose computing looks increasingly likely.

On the other hand, two big, current trends are pushing against that: (1) share nothing software, and (2) accelerators for future compute efficiency improvements. Looking at software, some solutions (Reddis vs memcached) use multiple processes on CMPs without sharing any memory between the different instances; similarly, data parallel programming models, such as PGAS, define the problem of data sharing between concurrent threads away through the programming language. In addition, thanks to further slowing of compute per Watt improvements in general purpose CPUs, a significant amount of computation will not happen on general purpose CPUs anymore, but instead on accelerators (GPUs, neural network accelerators). In those systems, the implementation of TM between general purpose CPUs and accelerators seems unlikely; instead, data sharing and transfer will need to be more carefully managed on a more coarse granularity.

Still, I believe there is room for flexible, easy to use optimistic concurrency control; especially in irregularly structured data sets, such as graphs, or sparse matrices, where static partitioning is impossible, and data replication would carry a too high storage cost.

As mentioned before, academic interest in TM has reduced since the peak in 2010; yet there are several recent publications on using and tuning applications on top of commercial HTM implementations, and working around limitations of HTM through thin software abstractions on top. Several works have suggested using transactional memory for non-coherent architectures, such as GPUs, and distributed systems, or for their failure atomicity properties in persistent memory (NVM) use cases. Several topics presented in Chapter 7 remain relevant and would benefit from additional evaluation and refinement.

For industry, once of the challenges that need solving (privately) is that of verifying that an HTM implementation is indeed correct. The experience with three Intel generations of TSX failing suggests that verification of the interactions and corner cases of HTM is not easy; my experience with my ASF implementations agrees: there are many racy interactions between the various speculation and coherence mechanisms that will not have an impact on single-location read-modify-write instructions, or the general memory consistency model, but will cause a transaction to not provide atomicity or isolation in the rarest circumstances.

But not just on the micro-architectural level, but also on the ISA, there seem to be many challenges defining how HTM should behave, especially with topics such as performance counters, trace, memory translation, and debug. Furthermore, the exact behaviour of transactions in a non-sequentially-consistent memory model, especially in weaker memory models such as IBM Power, or the presence of non-transactional accesses in ASF make that challenging [287, 370].

8.5 Post-thesis work

After the closing of the AMD Dresden office at the end of 2012, I moved to TU Dresden briefly and worked on three papers [274, 328, 337], and joined Arm Research in Cambridge, UK in August 2013. There, I worked on the gem5 simulator [241], implementing a snoop filter / directory using the experience gained in simulator code bases in general, and coherent memory systems in particular. I have worked on memory consistency model related issues, and picked up new projects such as low-power design (per-core DVFS, power modelling). At the time of writing, I am leading the Memory and Systems research group and look after general memory system architecture, and am consulting on HTM work. In total, since completing the bulk of HTM work, I have worked on the following publications: using turbo boosting for accelerating critical sections [301, 328], tracing and acceleration of simulation [356, 368, 382], power modelling [364, 375, 379], non-volatile memory [358, 371, 373], and future heterogeneous system composition [378].

The work in this thesis and the guidance of my advisers, mentors, and colleagues taught me many skills that are useful in my current role and will remain valuable throughout my life. First, this work has taught me how to dig through terabytes of logfile to find the needle in the haystack and not be afraid of drilling deep to find root causes for unexpected behaviour and bugs. Second, proposing a feature like HTM and trying to implement it in a realistic substrate has made me appreciate the general complexities and requirements of memory system design, and has given me a solid computer architecture and ISA design background (mainly through osmosis). I have been exposed to memory consistency models and their quirks, compilers, and generally a wide stack of software.

None of that would have been possible without this opportunity for work between industry and academia, and those who have put in considerable trust in my abilities, patience in explaining connections and backgrounds, and openness when discussing and improving wild ideas–many thanks!

Bibliography

[1] Robert M Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of research and Development*, 11(1):25–33, 1967.

[2] George H. Barnes, Richard M. Brown, Maso Kato, David J. Kuck, Daniel L. Slotnick, and Richard A. Stokes. The ILLIAC IV Computer. *IEEE Trans. Computers*, 17(8):746–757, 1968. doi: 10.1109/TC.1968.229158.

[3] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970. ISSN 0001-0782.

[4] F. Rubin. The Lee Path Connection Algorithm. *IEEE Trans. Computers*, 23(9):907–914, 1974. doi: 10.1109/T-C.1974.224054.

[5] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978. ISSN 0001-0782. doi: 10.1145/359545.359563.

[6] Intel. *Intel 8086 Family User's Manual*, October 1979. URL `http://matthieu.benoit.free.fr/cross/data_sheets/8086_family_Users_Manual.pdf`.

[7] Christos H. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the JACM*, 26(4):631–653, 1979.

[8] Jim Gray. The Transaction Concept: Virtues and Limitations (invited Paper). In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 144–154. IEEE Computer Society, 1981.

[9] Alastair J. W. Mayer. The Architecture of the Burroughs B5000: 20 Years Later and Still Ahead of the Times? *SIGARCH Comput. Archit. News*, 10(4):3–10, June 1982. ISSN 0163-5964. doi: 10.1145/641542.641543.

[10] James E. Smith. Decoupled Access/execute Computer Architectures. In Stephen A. Szygenda, John Hughes, Matt Blanton, Terry J. Wagner, Dennis J. Frailey, Tom Gunter, Chuck McLeavy, G. Jack Lipovski, and Miroslaw Malek, editors, *9th International Symposium on Computer Architecture (ISCA 1982), Austin, TX, USA, April 26-29, 1982*, pages 112–119. IEEE Computer Society, 1982. doi: 10.1145/800048.801719.

[11] Theo Härder and Andreas Reuter. Principles of Transaction-oriented Database Recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983. doi: 10.1145/289.291.

[12] Mark D. Hill and Alan Jay Smith. Experimental Evaluation of On-chip Microprocessor Cache Memories. In Dharma P. Agrawal, editor, *Proceedings of the 11th Annual Symposium on Computer Architecture, Ann Arbor, USA, June 1984*, pages 158–166. ACM, 1984. doi: 10.1145/800015.808178.

[13] Adam Levinthal and Thomas K. Porter. Chap - a SIMD Graphics Processor. In Hank Christiansen, editor, *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1984, Minneapolis, Minnesota, USA, July 23-27, 1984*, pages 77–82. ACM, 1984. doi: 10.1145/800031.808581.

[14] Steven R. Kunkel and James E. Smith. Optimal Pipelining in Supercomputers. In Hideo Aiso, editor, *Proceedings of the 13th Annual Symposium on Computer Architecture, Tokyo, Japan, June 1986*, pages 404–411. IEEE Computer Society, 1986. doi: 10.1145/17407.17403.

[15] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. In Joseph Y. Halpern, editor, *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing, Calgary, Alberta, Canada, August 11-13, 1986*, pages 229–239. ACM, 1986. doi: 10.1145/10590.10610.

[16] Thomas F. Knight. An Architecture for Mostly Functional Languages. In *LISP and Functional Programming*, pages 105–112, 1986.

[17] Christoph Scheurich and Michel Dubois. Correct Memory Operation of Cache-based Multiprocessors. In Daniel C. St. Clair, editor, *Proceedings of the 14th Annual International Symposium on Computer Architecture. Pittsburgh, PA, USA, June 1987*, pages 234–243, 1987. ISBN 0-8186-0776-9. doi: 10.1145/30350.30377. URL `http://dl.acm.org/citation.cfm?id=30350`.

[18] Eric H Jensen, Gary W Hagensen, and Jeffrey M Broughton. A New Approach to Exclusive Data Access in Shared Memory Multiprocessors. Technical report, Technical Report UCRL-97663, Lawrence Livermore National Laboratory, 1987.

[19] Maurice P. Herlihy. Impossibility and Universality Results for Wait-free Synchronization. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 276–290, New York, NY, USA, 1988. ACM. ISBN 0-89791-277-2. doi: 10.1145/62546.62593.

[20] Nigel P. Topham, Amos Omondi, and Roland N. Ibbett. On the Design and Performance of Conventional Pipelined Architectures. *The Journal of Supercomputing*, 1(4):353–393, 1988. doi: 10.1007/BF00128488.

[21] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[22] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John L. Hennessy. The Directory-based Cache Coherence Protocol for the DASH Multiprocessor. In Jean-Loup Baer, Larry Snyder, and James R. Goodman, editors, *Proceedings of the 17th Annual International Symposium on Computer Architecture, Seattle, WA, USA, June 1990*, pages 148–159. ACM, 1990. doi: 10.1145/325164.325132.

[23] Steve McGeady. The I960ca Superscalar Implementation of the 80960 Architecture. In *Intellectual Leverage: Thirty-Fifth IEEE Computer Society International Conference, Compcon Spring '90, San Francisco, California, USA, February 26 - March 2, 1992, Digest of Papers.*, pages 232–240. IEEE, 1990. doi: 10.1109/CMPCON.1990.63681.

[24] Henry Massalin and Calton Pu. A Lock-Free Multiprocessor OS Kernel. Technical Report CUCS-005-91, Columbia University, 1991.

[25] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1), 1991.

[26] Tse-Yu Yeh and Yale N. Patt. Two-level Adaptive Training Branch Prediction. In Yashwant K. Malaiya, editor, *Proceedings of the 24th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 24, Albuquerque, New Mexico, USA, November 18-20, 1991*, pages 51–61. ACM/IEEE, 1991. doi: 10.1145/123465.123475.

[27] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *In Proceedings of the 1991 International Conference on Parallel Processing*, pages 355–364, 1991.

[28] Maurice Herlihy. Wait-free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991. ISSN 0164-0925. doi: 10.1145/114005.102808.

[29] William W. Collier. *Reasoning about Parallel Architectures*. Prentice Hall, 1992. ISBN 978-0-13-766098-8.

[30] Henry Massalin and Calton Pu. A Lock-free Multiprocessor OS Kernel (abstract). *Operating Systems Review*, 26(2):8, 1992. doi: 10.1145/142111.993246.

[31] Yehuda Afek, Geoffrey M. Brown, and Michael Merritt. Lazy Caching. *ACM Trans. Program. Lang. Syst.*, 15(1):182–205, 1993. doi: 10.1145/151646.151651.

[32] André Seznec. A Case for Two-way Skewed-associative Caches. In Alan Jay Smith, editor, *Proceedings of the 20th Annual International Symposium on Computer Architecture. San Diego, CA, May 1993*, pages 169–178. ACM, 1993. doi: 10.1145/165123.165152.

[33] Per Stenström, Mats Brorsson, and Lars Sandberg. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In Alan Jay Smith, editor, *Proceedings of the 20th Annual International Symposium on Computer Architecture, San Diego, CA, USA, May 1993*, pages 109–118. ACM, 1993. doi: 10.1145/165123.165147.

[34] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In Alan Jay Smith, editor, *Proceedings of the 20th Annual International Symposium on Computer Architecture. San Diego, CA, May 1993*, pages 289–300. ACM, 1993. ISBN 0-8186-3810-9. doi: 10.1145/165123.165164. URL `http://dl.acm.org/citation.cfm?id=165123`.

[35] Janice M. Stone, Harold S. Stone, Philip Heidelberger, and John Turek. Multiple Reservations and the Oklahoma Update. *IEEE P&DT*, 1(4):58–71, 1993. doi: 10.1109/88.260295.

[36] André Seznec. Decoupled Sectored Caches: Conciliating Low Tag Implementation Cost and Low Miss Ratio. In David A. Patterson, editor, *Proceedings of the 21st Annual International Symposium on Computer Architecture. Chicago, IL, USA, April 1994*, pages 384–393. IEEE Computer Society, 1994. doi: 10.1109/ISCA.1994.288133.

[37] Keith I. Farkas, Norman P. Jouppi, and Paul Chow. How Useful Are Non-blocking Loads, Stream Buffers and Speculative Execution in Multiple Issue Processors? In *Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture (HPCA 1995), Raleigh, North Carolina, USA, January 22-25, 1995*, pages 78–89. IEEE Computer Society, 1995. doi: 10.1109/HPCA.1995.386553.

[38] Linley Gwennap. Intel's P6 Uses Decoupled Superscalar Design. *Microprocessor Report*, 9(2):9–15, 1995.

[39] Nir Shavit and Dan Touitou. Software Transactional Memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, Ottawa, ON, Canada, August 1995.

[40] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *Computer*, 29(12):66–76, 1996.

[41] Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, May 1996.

[42] Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun. A Single-chip Multiprocessor. *IEEE Computer*, 30(9):79–85, 1997. doi: 10.1109/2.612253.

[43] Alex Peleg, Sam Wilkie, and Uri C. Weiser. Intel MMX for Multimedia Pcs. *Commun. ACM*, 40(1): 24–38, 1997. doi: 10.1145/242857.242865.

[44] Mark Moir. Transparent Support for Wait-Free Transactions. In *Proceedings of the 11th International Workshop on Distributed Algorithms*, Saarbrucken, Germany, September 1997.

[45] Li Fan, Pei Cao, Jussara M. Almeida, and Andrei Z. Broder. Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol. In *SIGCOMM*, pages 254–265, 1998. doi: 10.1145/285237.285287.

[46] Venkata Krishnan and Josep Torrellas. An Direct-execution Framework for Fast and Accurate Simulation of Superscalar Processors. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques, Paris, France, October 12-18, 1998*, pages 286–293. IEEE Computer Society, 1998. doi: 10.1109/PACT.1998.727263.

[47] Paul E McKenney and John D Slingwine. Read-copy Update: Using Execution History to Solve Concurrency Problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[48] Lance Hammond, Mark Willey, and Kunle Olukotun. Data Speculation Support for a Chip Multiprocessor. In Dileep Bhandarkar and Anant Agarwal, editors, *ASPLOS-VIII Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA, October 3-7, 1998.*, pages 58–69. ACM Press, 1998. ISBN 1-58113-107-0. doi: 10.1145/291069.291020. URL `http://dl.acm.org/citation.cfm?id=291069`.

[49] Sridhar Gopal, T. N. Vijaykumar, James E. Smith, and Gurindar S. Sohi. Speculative Versioning Cache. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture, Las Vegas, Nevada, USA, January 31 - February 4, 1998*, pages 195–205. IEEE Computer Society, 1998. ISBN 0-8186-8323-6. doi: 10.1109/HPCA.1998.650559. URL `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5242`.

[50] Adi Yoaz, Mattan Erez, Ronny Ronen, and Stéphan Jourdan. Speculation Techniques for Improving Load Related Instruction Scheduling. In Allan Gottlieb and William J. Dally, editors, *Proceedings of the 26th Annual International Symposium on Computer Architecture, ISCA 1999, Atlanta, Georgia, USA, May 2-4, 1999*, pages 42–53. IEEE Computer Society, 1999. doi: 10.1109/ISCA.1999.765938.

[51] Pedro Marcuello and Antonio González. Clustered Speculative Multithreaded Processors. In Theodore S. Papatheodorou, Mateo Valero, Constantine D. Polychronopoulos, Yoichi Muraoka, and Jesús Labarta, editors, *Proceedings of the 13th international conference on Supercomputing, ICS 1999, Rhodes, Greece, June 20-25, 1999*, pages 365–372. ACM, 1999. ISBN 1-58113-164-X. doi: 10.1145/305138.305214.

[52] Venkata Krishnan and Josep Torrellas. A Chip-multiprocessor Architecture with Speculative Multithreading. *IEEE Trans. Computers*, 48(9):866–880, 1999. doi: 10.1109/12.795218.

[53] Alan D. Berenbaum and Joel S. Emer, editors. *27th International Symposium on Computer Architecture (ISCA 2000), June 10-14, 2000, Vancouver, Bc, Canada*, 2000. IEEE Computer Society. URL `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6892`.

[54] Marcelo H. Cintra, José F. Martínez, and Josep Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-memory Multiprocessors. In Berenbaum and Emer [53], pages 13–24. doi: 10.1109/ISCA.2000.854373. URL `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6892`.

[55] J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A Scalable Approach to Thread-level Speculation. In Berenbaum and Emer [53], pages 1–12. doi: 10.1109/ISCA.2000.854372. URL `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6892`.

[56] Daniel A. Jiménez and Calvin Lin. Dynamic Branch Prediction with Perceptrons. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture (HPCA'01), Nuevo Leone, Mexico, January 20-24, 2001*, pages 197–206. IEEE Computer Society, 2001. doi: 10.1109/HPCA.2001.903263.

[57] Darren J. Kerbyson, Henry J. Alme, Adolfy Hoisie, Fabrizio Petrini, Harvey J. Wasserman, and Michael L. Gittings. Predictive Performance and Scalability Modeling of a Large-scale Application. In Greg Johnson, editor, *Proceedings of the 2001 ACM/IEEE conference on Supercomputing, Denver, CO, USA, November 10-16, 2001, CD-ROM*, page 37. ACM, 2001. doi: 10.1145/582034.582071.

[58] Ravi Rajwar and James R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34th IEEE/ACM International Symposium on Microarchitecture*, Austin, TX, December 2001.

[59] Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. Design and Implementation of a Dynamic Optimization Framework for Windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 2001.

[60] Allan Hartstein and Thomas R. Puzak. The Optimum Pipeline Depth for a Microprocessor. In Yale N. Patt, Dirk Grunwald, and Kevin Skadron, editors, *29th International Symposium on Computer Architecture (ISCA 2002), 25-29 May 2002, Anchorage, AK, USA*, pages 7–13. IEEE Computer Society, 2002. doi: 10.1109/ISCA.2002.1003557.

[61] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, 2002. doi: 10.1109/2.982916.

[62] Ravi Rajwar and James R. Goodman. Transactional Lock-free Execution of Lock-based Programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 2002.

[63] André Seznec, Stephen Felix, Venkata Krishnan, and Yiannakis Sazeides. Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor. In Yale N. Patt, Dirk Grunwald, and Kevin Skadron, editors, *29th International Symposium on Computer Architecture (ISCA 2002), 25-29 May 2002, Anchorage, AK, USA*, pages 295–306. IEEE Computer Society, 2002. doi: 10.1109/ISCA.2002.1003587.

[64] José F Martínez and Josep Torrellas. Speculative Synchronization: Applying Thread-level Speculation to Explicitly Parallel Applications. In *ACM SIGOPS Operating Systems Review*, volume 36, pages 18–29. ACM, 2002.

[65] Peter Rundberg and Per Stenstrom. Reordered Speculative Execution of Critical Sections. In *Proceedings of the 2002 International Conference on Parallel Processing (ICPP '02)*, 2002.

[66] Tim Harris and Keir Fraser. Language Support for Lightweight Transactions. In *Proceedings of the 18th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2003.

[67] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free Synchronization: Double-ended Queues As an Example. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 522, Washington, D.C., USA, 2003. IEEE Computer Society. ISBN 0-7695-1920-2.

[68] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. In Elizabeth Borowsky and Sergio Rajsbaum, editors, *Proceedings of the Twenty-Second ACM Symposium on Principles of Distributed Computing, PODC 2003, Boston, Massachusetts, USA, July 13-16, 2003*, pages 92–101. ACM, 2003. doi: 10.1145/872035.872048.

[69] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using Simpoint for Accurate and Efficient Simulation. In Bill Cheng, Satish K. Tripathi, Jennifer Rexford, and William H. Sanders, editors, *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2003, June 9-14, 2003, San Diego, CA, USA*, pages 318–319. ACM, 2003. doi: 10.1145/781027.781076.

[70] Kai Richter, Marek Jersak, and Rolf Ernst. A Formal Approach to Mpsoc Performance Verification. *IEEE Computer*, 36(4):60–67, 2003. doi: 10.1109/MC.2003.1193230.

[71] Milo M. K. Martin, Mark D. Hill, and David A. Wood. Token Coherence: Decoupling Performance and Correctness. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 182–193, New York, N.Y., USA, 2003. ACM Press. ISBN 0-7695-1945-8. doi: 10.1145/859618.859640.

[72] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, October 2003.

[73] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, Anaheim, CA, February 2003.

[74] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional

Memory Coherence and Consistency. *SIGARCH Comput. Archit. News*, 32(2):102, 2004. ISSN 0163-5964. doi: 10.1145/1028176.1006711.

[75] Ronald N. Kalla, Balaram Sinharoy, and Joel M. Tendler. IBM Power5 Chip: A Dual-core Multithreaded Processor. *IEEE Micro*, 24(2):40–47, 2004. doi: 10.1109/MM.2004.1289290.

[76] Tejas Karkhanis and James E. Smith. A First-order Superscalar Processor Model. In *31st International Symposium on Computer Architecture (ISCA 2004), 19-23 June 2004, Munich, Germany*, pages 338–349. IEEE Computer Society, 2004. doi: 10.1109/ISCA.2004.1310786.

[77] Ilhyun Kim and Mikko H. Lipasti. Understanding Scheduling Replay Schemes. In *10th International Conference on High-Performance Computer Architecture (HPCA-10 2004), 14-18 February 2004, Madrid, Spain*, pages 198–209. IEEE Computer Society, 2004. doi: 10.1109/HPCA.2004.10011.

[78] Gabriel Marin and John M. Mellor-Crummey. Cross-architecture Performance Predictions for Scientific Applications Using Parameterized Models. In Edward G. Coffman Jr., Zhen Liu, and Arif Merchant, editors, *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2004, June 10-14, 2004, New York, NY, USA*, pages 2–13. ACM, 2004. doi: 10.1145/1005686.1005691.

[79] Advanced Micro Devices, Inc. AMD Releases AMD64 Simulator Application; SimNow, a High-performance X86 Platform Simulator, Extends Access to “Pacifica” Virtualization Technology. 2005. URL `http://ir.amd.com/news-releases/news-release-details/amd-releases-amd64-simulator-application-simnow-high-performance`.

[80] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded Transactional Memory. In *11th International Conference on High-Performance Computer Architecture (HPCA-11 2005), 12-16 February 2005, San Francisco, CA, USA*, pages 316–327. IEEE Computer Society, 2005. doi: 10.1109/HPCA.2005.41.

[81] Fabrice Bellard. Qemu, a Fast and Portable Dynamic Translator. In *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, pages 41–46. USENIX, 2005. URL `http://www.usenix.org/events/usenix05/tech/freenix/bellard.html`.

[82] Colin Blundell, E. Christopher Lewis, and Milo M. K. Martin. Deconstructing Transactional Semantics: The Subtleties of Atomicity. In *Proceedings of the Workshop on Duplicating, Deconstructing and Debunking*, Madison, WI, June 2005.

[83] Engin Ipek, Bronis R. de Supinski, Martin Schulz, and Sally A. McKee. An Approach to Performance Prediction for Parallel Applications. In José C. Cunha and Pedro D. Medeiros, editors, *Euro-Par 2005, Parallel Processing, 11th International Euro-Par Conference, Lisbon, Portugal, August 30 - September 2, 2005, Proceedings*, volume 3648 of *Lecture Notes in Computer Science*, pages 196–205. Springer, 2005. doi: 10.1007/11549468_24.

[84] Austen McDonald, JaeWoong Chung, Hassan Chafi, Chi Cao Minh, Brian D. Carlstrom, Lance Hammond, Christos Kozyrakis, and Kunle Olukotun. Characterization of TCC on Chip-multiprocessors. In *14th International Conference on Parallel Architecture and Compilation Techniques (PACT 2005), 17-21 September 2005, St. Louis, MO, USA*, pages 63–74. IEEE Computer Society, 2005. doi: 10.1109/PACT.2005.11.

[85] Eliot Moss and Antony L. Hosking. Nested Transactional Memory: Model and Preliminary Architecture Sketches. In *Proceedings of the Workshop on Synchronization and Concurrency in Object-Oriented Languages*, San Diego, CA, October 2005.

[86] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing Transactional Memory. In *Proceedings of the 32nd International Symposium on Computer Architecture*, Madison, WI, June 2005.

[87] William N. Scherer III and Michael L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, Las Vegas, NV, July 2005.

[88] Adrian Cristal, Oliverio J. Santana, Francisco Cazorla, Marco Galluzzi, Tanausu Ramirez, Miquel Pericas, and Mateo Valero. Kilo-instruction Processors: Overcoming the Memory Wall. *IEEE Micro*, 25(3):48–57, 2005. ISSN 0272-1732. doi: 10.1109/MM.2005.53.

[89] Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Adaptive Software Transactional Memory. In Pierre Fraigniaud, editor, *Distributed Computing, 19th International Conference, DISC 2005, Cracow, Poland, September 26-29, 2005, Proceedings*, volume 3724 of *Lecture Notes in Computer Science*, pages 354–368. Springer, 2005. ISBN 3-540-29163-6. doi: 10.1007/11561927_26.

[90] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable Memory Transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, N.Y., USA, 2005. ACM Press. ISBN 1-59593-080-9. doi: 10.1145/1065944.1065952.

[91] Tim Harris. Exceptions and Side-effects in Atomic Blocks. *Sci. Comput. Program.*, 58(3):325–343, 2005. ISSN 0167-6423. doi: 10.1016/j.scico.2005.03.005.

[92] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Redd, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 26th ACM Conference on Programming Language Design and Implementation*, Chicago, IL, June 2005.

[93] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005. ISSN 0163-5964. doi: 10.1145/1105734.1105747.

[94] Mark Moir. Hybrid Transactional Memory, July 2005. Unpublished manuscript.

[95] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and Runtime Support for Efficient Software Transactional Memory. In Michael I. Schwartzbach and Thomas Ball, editors, *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, pages 26–37. ACM, 2006. doi: 10.1145/1133981.1133985.

[96] Laura Carrington, Allan Snavely, and Nicole Wolter. A Performance Prediction Framework for Scientific Applications. *Future Generation Comp. Syst.*, 22(3):336–346, 2006. doi: 10.1016/j.future.2004.11.019.

[97] P. J. Joseph, Kapil Vaswani, and Matthew J. Thazhuthaveetil. Construction and Use of Linear Regression Models for Processor Performance Analysis. In *12th International Symposium on High-Performance Computer Architecture, HPCA-12 2006, Austin, Texas, February 11-15, 2006*, pages 99–108. IEEE Computer Society, 2006. doi: 10.1109/HPCA.2006.1598116.

[98] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid Transactional Memory. In *Proceedings of the 11th ACM Symposium on Principles and Practice of Parallel Programming*, New York, NY, March 2006.

[99] Benjamin C. Lee and David M. Brooks. Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction. In John Paul Shen and Margaret Martonosi, editors, *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 185–194. ACM, 2006. doi: 10.1145/1168857.1168881.

[100] Yingmin Li, Benjamin C. Lee, David M. Brooks, Zhigang Hu, and Kevin Skadron. CMP Design Space Exploration Subject to Physical Constraints. In *12th International Symposium on High-Performance Computer Architecture, HPCA-12 2006, Austin, Texas, February 11-15, 2006*, pages 17–28. IEEE Computer Society, 2006. doi: 10.1109/HPCA.2006.1598109.

[101] Virendra J. Marathe, Michael Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the Overhead of Nonblocking Software Transactional Memory. Technical Report TR 893, Department of Computer Science, University of Rochester, March 2006.

[102] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based Transactional Memory. In *12th International Symposium on High-Performance Computer Architecture, HPCA-12 2006, Austin, Texas, USA, February 11-15, 2006*, pages 254–265. IEEE Computer Society, 2006. doi: 10.1109/HPCA.2006.1598134.

[103] Torvald Riegel, Pascal Felber, and Christof Fetzer. A Lazy Snapshot Algorithm with Eager Validation. In *Proceedings of the 20th International Symposium on Distributed Computing*, Stockholm, Sweden, September 2006.

[104] Arrvindh Shriraman, Michael Spear, Hemayet Hossain, Sandhya Dwarkadas, and Michael L. Scott. An Integrated Hardware-Software Approach to Flexible Transactional Memory. Technical Report TR 910, Department of Computer Science, University of Rochester, December 2006.

[105] Arrvindh Shriraman, Virendra J. Marathe, Sandhya Dwarkadas, Michael L. Scott, David Eisenstat, Christopher Heriot, William N. Scherer III, and Michael Spear. Hardware Acceleration of Software Transactional Memory. In *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Ottawa, ON, Canada, June 2006.

[106] Travis Skare and Christos Kozyrakis. Early Release: Friend or Foe. In *Workshop on Transactional Memory Workloads*, volume 77, 2006.

[107] corbet. Counting on the Time Stamp Counter. `http://lwn.net/Articles/209101/`, November 2006. URL `https://lwn.net/Articles/209101/`.

[108] Dan Grossman, Jeremy Manson, and William Pugh. What Do High-level Memory Models Mean for Transactions? In Antony L. Hosking and Ali-Reza Adl-Tabatabai, editors, *Proceedings of the 2006 workshop on Memory System Performance and Correctness, San Jose, California, USA, October 11, 2006*, pages 62–69. ACM, 2006. ISBN 1-59593-578-9. doi: 10.1145/1178597.1178609.

[109] Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Brad Calder, and Osvaldo Colavin. Unbounded Page-based Transactional Memory. In John Paul Shen and Margaret Martonosi, editors, *Proceedings of the 12th International*

Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006, pages 347–358. ACM, 2006. ISBN 1-59593-451-0. doi: 10.1145/1168857.1168901.

[110] Nicholas Riley and Craig B. Zilles. Hardware Tansactional Memory Support for Lightweight Dynamic Language Evolution. In Peri L. Tarr and William R. Cook, editors, *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 998–1008. ACM, 2006. ISBN 1-59593-491-X. doi: 10.1145/1176617.1176758.

[111] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid Transactional Memory. In *ASPLOS*, 2006.

[112] Maurice Herlihy, Victor Luchangco, and Mark Moir. A Flexible Framework for Implementing Software Transactional Memory. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming languages, systems, and applications*, pages 253–262, New York, N.Y., USA, 2006. ACM Press. ISBN 1-59593-348-4. doi: 10.1145/1167473.1167495.

[113] Austen McDonald, JaeWoong Chung, Brian D. Carlstrom, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural Semantics for Practical Transactional Memory. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*. June 2006.

[114] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: A High Performance Software Transactional Memory System for a Multi-core Runtime. In *PPoPP*, 2006.

[115] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, 2006. ISSN 0272-1732. doi: 10.1109/MM.2006.82.

[116] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *DISC*, 2006.

[117] Robert Ennals. Software Transactional Memory Should Not Be Obstruction-free. Technical Report IRC-TR-06-052, intel, 2006.

[118] Michelle Moravan, Jayaram Bobba, Kevin Moore, Luke Yen, Mark Hill, Ben Liblit, Michael Swift, and David Wood. Supporting Nested Transactional Memory in LogTM. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 2006.

[119] Bratin Saha, Ali-Reza Adl-Tabatabai, and Quinn Jacobson. Architectural Support for Software Transactional Memory. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 185–196. IEEE, 2006.

[120] Craig Zilles and Lee Baugh. Extending Hardware Transactional Memory to Support Non-busy Waiting and Non-transactional Actions. In *TRANSACT: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[121] Colin Blundell, Joe Devietti, E. Christopher Lewis, and Milo M. K. Martin. Making the Fast Case Common and the Uncommon Case Simple in Unbounded Transactional Memory. In Tullsen and Calder [148], pages 24–34. ISBN 978-1-59593-706-3. doi: 10.1145/1250662.1250667. URL `http://dl.acm.org/citation.cfm?id=1250662`.

[122] Pat Conway and Bill Hughes. The AMD Opteron Northbridge Architecture. *IEEE Micro*, 27(2): 10–21, 2007. doi: 10.1109/MM.2007.43.

[123] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. ISBN 0-12-370490-1.

[124] Owen S. Hofmann, Donald E. Porter, Christopher J. Rossbach, Hany E. Ramadan, and Emmett Witchel. Solving Difficult HTM Problems Without Difficult Hardware. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, Portland, OR, August 2007.

[125] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. Kvm: The Linux Virtual Machine Monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230. Dttawa, Dntorio, Canada, 2007.

[126] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Grasso Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In Dean M. Tullsen and Brad Calder, editors, *34th International Symposium on Computer Architecture (ISCA 2007), June 9-13, 2007, San Diego, California, USA*, pages 69–80. ACM, 2007. doi: 10.1145/1250662.1250673.

[127] Naveen Neelakantam, Ravi Rajwar, Suresh Srinivas, Uma Srinivasan, and Craig Zilles. Hardware Atomicity for Reliable Software Speculation. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, CA, June 2007.

[128] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 89–100. ACM, 2007. doi: 10.1145/1250734.1250746.

[129] Yang Ni, Vijay Menon, Ali-Reza Adl-Tabatabai, Antony Hosking, Rick Hudson, Eliot Moss, Bratin Saha, and Tatiana Shpeisman. Open Nesting in Software Transactional Memory. In *Proceedings of the 12th ACM Symposium on Principles and Practice of Parallel Programming*, San Jose, CA, March 2007.

[130] Marek Olszewski, Jeremy Cutler, and J. Gregory Steffan. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, Brasov, Romania, September 2007.

[131] Swaroop Sridhar, Jonathan S. Shapiro, and Prashanth P. Bungale. Hdtrans: A Low-overhead Dynamic Translator. *SIGARCH Computer Architecture News*, 35(1):135–140, 2007. doi: 10.1145/1241601.1241602.

[132] Cheng Wang, Wei-Yu Chen, Youfeng Wu, Bratin Saha, and Ali-Reza Adl-Tabatabai. Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. In *Proceedings of the 2007 International Symposium on Code Generation and Optimization*, San Jose, CA, March 2007.

[133] Katherine A. Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham, Paul Hargrove, Paul N. Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Michael L. Welcome, and Tong Wen. Productivity and Performance Using Partitioned Global Address Space Languages. In Marc Moreno Maza and Stephen M. Watt, editors, *Parallel Symbolic Computation, PASCO 2007, International Workshop, 27-28 July 2007,*

University of Western Ontario, London, Ontario, Canada, pages 24–32. ACM, 2007. doi: 10.1145/1278177.1278183.

[134] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, and Michael M. Swift David A. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, Phoenix, AZ, February 2007.

[135] Matt Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *Proceedings of the 2007 IEEE International Symposium on Performance Analysis of Systems and Software*, San Jose, CA, April 2007.

[136] *Software Optimization Guide for AMD Family 10h Processors*. Advanced Micro Devices, Inc., 3.05 edition, January 2007.

[137] Lee Baugh and Craig Zilles. An Analysis of I/O and Syscalls in Critical Sections and Their Implications for Transactional Memory. In *TRANSACT 2007*, 2007.

[138] Hassan Chafi, Jared Casper, Brian D. Carlstrom, Austen McDonald, Chi Cao Minh, Woongki Baek, Christos Kozyrakis, and Kunle Olukotun. A Scalable, Non-blocking Approach to Transactional Memory. In *13th International Symposium on High Performance Computer Architecture (HPCA)*. February 2007.

[139] Njuguna Njoroge, Jared Casper, Sewook Wee, Yuriy Teslyar, Daxia Ge, Christos Kozyrakis, and Kunle Olukotun. ATLAS: a Chip-multiprocessor with Transactional Memory Support. In Rudy Lauwereins and Jan Madsen, editors, *2007 Design, Automation and Test in Europe Conference and Exposition, DATE 2007, Nice, France, April 16-20, 2007*, pages 3–8. EDA Consortium, San Jose, CA, USA, 2007. ISBN 978-3-9810801-2-4. doi: 10.1109/DATE.2007.364558. URL `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4211748`.

[140] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. Stmbench7: A Benchmark for Software Transactional Memory. In Paulo Ferreira, Thomas R. Gross, and Luís Veiga, editors, *Proceedings of the 2007 EuroSys Conference, Lisbon, Portugal, March 21-23, 2007*, pages 315–324. ACM, 2007. ISBN 978-1-59593-636-3. doi: 10.1145/1272996.1273029.

[141] Michael L. Scott, Michael F. Spear, Luke Dalessandro, and Virendra J. Marathe. Delaunay Triangulation with Transactions and Barriers. In *IEEE 10th International Symposium on Workload Characterization, IISWC 2007, Boston, MA, USA, 27-29 September, 2007*, pages 107–113. IEEE Computer Society, 2007. ISBN 978-1-4244-1561-8. doi: 10.1109/IISWC.2007.4362186. URL `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4362166`.

[142] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance Pathologies in Hardware Transactional Memory. In Tullsen and Calder [148], pages 81–91. ISBN 978-1-59593-706-3. doi: 10.1145/1250662.1250674. URL `http://dl.acm.org/citation.cfm?id=1250662`.

[143] Daniel Sánchez, Luke Yen, Mark D. Hill, and Karthikeyan Sankaralingam. Implementing Signatures for Transactional Memory. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-40 2007), 1-5 December 2007, Chicago, Illinois, USA*, pages 123–133. IEEE Computer Society, 2007. ISBN 0-7695-3047-8. doi: 10.1109/MICRO.2007.20. URL `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4408231`.

[144] Nehir Sönmez, Cristian Perfumo, Srdjan Stipic, Adrián Cristal, Osman S. Unsal, and Mateo Valero. Unreadtvar: Extending Haskell Software Transactional Memory for Performance. In Marco T. Morazán, editor, *Proceedings of the Eighth Symposium on Trends in Functional Programming, TFP 2007, New York City, New York, USA, April 2-4. 2007.*, volume 8 of *Trends in Functional Programming*, pages 89–104. Intellect, 2007. ISBN 978-1-84150-196-3.

[145] Cristian Perfumo, Nehir Sonmez, Adrian Cristal, Osman S. Unsal, Mateo Valero, and Tim Harris. Dissecting Transactional Executions in Haskell. In *TRANSACT 2007*, 2007.

[146] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing Isolation and Ordering in STM. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 78–88, New York, N.Y., USA, 2007. ACM Press. ISBN 978-1-59593-633-2. doi: 10.1145/1250734.1250744.

[147] Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization Techniques for Software Transactional Memory. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, pages 338–339, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-616-5. doi: 10.1145/1281100.1281161.

[148] Dean M. Tullsen and Brad Calder, editors. *34th International Symposium on Computer Architecture (ISCA 2007), June 9-13, 2007, San Diego, California, USA*, 2007. ACM. ISBN 978-1-59593-706-3. URL `http://dl.acm.org/citation.cfm?id=1250662`.

[149] Luke Dalessandro, Virendra J Marathe, Michael F Spear, and Michael L Scott. Capabilities and Limitations of Library-based Software Transactional Memory in C++. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, volume 5, page 49, 2007.

[150] Pascal Felber, Christof Fetzer, Ulrich Müller, Torvald Riegel, Martin Süßkraut, and Heiko Sturzrehm. Transactifying Applications Using an Open Compiler Framework. In *TRANSACT*, August 2007.

[151] Yossi Lev, Mark Moir, and Dan Nussbaum. Phtm: Phased Transactional Memory. In *Workshop on Transactional Computing (Transact)*, 2007.

[152] Yannis Smaragdakis, Anthony Kay, Reimer Behrends, and Michal Young. Transactions with Isolation and Cooperation. In *Proceedings of the 22nd ACM Conference on Object Oriented Programming, Systems, Languages, and Applications*, Montreal, Quebec, Canada, October 2007.

[153] Michael F Spear, Arrvindh Shriraman, Hemayet Hossain, Sandhya Dwarkadas, and Michael L Scott. Alert-on-update: A Communication Aid for Shared Memory Multiprocessors. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 132–133. ACM, 2007.

[154] Michael F Spear, Arrvindh Shriraman, Luke Dalessandro, Sandhya Dwarkadas, and Michael L Scott. Nonblocking Transactions without Indirection Using Alert-on-update. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 210–220. ACM, 2007.

[155] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software Transactional Memory: Why Is It Only a Research Toy? *Commun. ACM*, 51(11):40–46, 2008.

[156] Kunal Agrawal, Jeremy Fineman, and Jim Sukha. Nested Parallelism in Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, February 2008.

[157] Jayaram Bobba, Neelam Goyal, Mark D. Hill, Michael M. Swift, and David A. Wood. Tokentm: Efficient Execution of Large Transactions with Hardware Transactional Memory. In *35th International Symposium on Computer Architecture (ISCA 2008), June 21-25, 2008, Beijing, China*, pages 127–138. IEEE Computer Society, 2008. doi: 10.1109/ISCA.2008.24.

[158] Stephan Diestelhorst and Michael Hohmuth. Hardware Acceleration for Lock-Free Data Structures and Software-Transactional Memory. In *Proceedings of the Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods*, Boston, MA, April 2008.

[159] Stephan Diestelhorst. Hardwarebeschleunigung Für Software Transactional Memory. mathesis, TU Dresden, Germany, January 2008.

[160] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, February 2008.

[161] Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo. Intel Avx: New Frontiers in Performance Improvements and Energy Efficiency. *Intel white paper*, 19:20, 2008.

[162] Aamer Jaleel, Robert S Cohn, Chi-Keung Luk, and Bruce Jacob. Cmp$im: A Pin-based On-the-fly Multi-core Cache Simulator. In *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA*, pages 28–36, 2008.

[163] Yossi Lev and Jan-Willem Maessen. Split Hardware Transactions: True Nesting of Transactions Using Best-Effort Hardware Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, February 2008.

[164] Virendra J. Marathe, Michael F. Spear, and Michael L. Scott. Scalable Techniques for Transparent Privatization in Software Transactional Memory. In *2008 International Conference on Parallel Processing, ICPP 2008, September 8-12, 2008, Portland, Oregon, USA*, pages 67–74. IEEE Computer Society, 2008. doi: 10.1109/ICPP.2008.69.

[165] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard Hudson, Bratin Saha, and Adam Welc. Practical Weak-Atomicity Semantics for Java STM. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[166] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford Transactional Applications for Multi-processing. In David Christie, Alan Lee, Onur Mutlu, and Benjamin G. Zorn, editors, *4th International Symposium on Workload Characterization (IISWC 2008), Seattle, Washington, USA, September 14-16, 2008*, pages 35–46. IEEE Computer Society, 2008. doi: 10.1109/IISWC.2008.4636089.

[167] Hany E Ramadan, Christopher J Rossbach, and Emmett Witchel. Dependence-aware Transactional Memory for Increased Concurrency. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 246–257. IEEE Computer Society, 2008.

[168] Hany E. Ramadan, Christopher J. Rossbach, Donald E. Porter, Owen S. Hofmann, Bhandari Aditya, and Emmett Witchel. MetaTM/TxLinux: Transactional Memory for an Operating System. *IEEE Micro*, 28(1):42–51, 2008. doi: 10.1109/MM.2008.10.

[169] Simone Raoux, Geoffrey W. Burr, Matthew J. Breitwisch, Charles T. Rettner, Yi-Chou Chen, Robert M. Shelby, Martin Salinga, Daniel Krebs, Shih-Hung Chen, Hsiang-Lan Lung, and Chung Hon Lam. Phase-change Random Access Memory: A Scalable Technology. *IBM Journal of Research and Development*, 52(4-5):465–480, 2008. doi: 10.1147/rd.524.0465.

[170] Torvald Riegel, Christof Fetzer, and Pascal Felber. Automatic Data Partitioning in Software Transactional Memories. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[171] Arrvindh Shriraman, Sandhya Dwarkadas, and Michael L. Scott. Flexible Decoupled Transactional Memory Support. In *Proceedings of the 35th International Symposium on Computer Architecture*, Beijing, China, June 2008.

[172] Michael Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. Ordering-Based Semantics for Software Transactional Memory. In *Proceedings of the 12th International Conference On Principles Of DIstributed Systems*, Luxor, Egypt, December 2008.

[173] Michael M Swift, Haris Volos, Neelam Goyal, Luke Yen, Mark D Hill, and David A Wood. Os Support for Virtualizing Hardware Transactional Memory. In *Procs. of the 3rd ACM SIGPLAN Workshop on Transactional Computing*, 2008.

[174] Richard Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin Lee. Kicking the Tires of Software Transactional Memory: Why the Going Gets Tough. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[175] Siddhartha Chatterjee and Michael L. Scott, editors. *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, Salt Lake City, Ut, Usa, February 20-23, 2008*, 2008. ACM. ISBN 978-1-59593-795-7. URL `http://dl.acm.org/citation.cfm?id=1345206`.

[176] Cristian Perfumo, Nehir Sönmez, Srdjan Stipic, Osman S. Unsal, Adrián Cristal, Tim Harris, and Mateo Valero. The Limits of Software Transactional Memory (STM): Dissecting Haskell STM Applications on a Many-core Environment. In Alex Ramírez, Gianfranco Bilardi, and Michael Gschwind, editors, *Proceedings of the 5th Conference on Computing Frontiers, 2008, Ischia, Italy, May 5-7, 2008*, pages 67–78. ACM, 2008. ISBN 978-1-60558-077-7. doi: 10.1145/1366230.1366241.

[177] Mohammad Ansari, Christos Kotselidis, Ian Watson, Chris C. Kirkham, Mikel Luján, and Kim Jarvis. Lee-tm: A Non-trivial Benchmark Suite for Transactional Memory. In Anu G. Bourgeois and Si-Qing Zheng, editors, *Algorithms and Architectures for Parallel Processing, 8th International Conference, ICA3PP 2008, Cyprus, June 9-11, 2008, Proceedings*, volume 5022 of *Lecture Notes in Computer Science*, pages 196–207. Springer, 2008. ISBN 978-3-540-69500-4. doi: 10.1007/978-3-540-69501-1_21.

[178] Luke Yen, Stark C. Draper, and Mark D. Hill. Notary: Hardware Techniques to Enhance Signatures. In *41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-41 2008),*

November 8-12, 2008, Lake Como, Italy, pages 234–245. IEEE Computer Society, 2008. ISBN 978-1-4244-2836-6. doi: 10.1109/MICRO.2008.4771794. URL `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4757685`.

[179] Rachid Guerraoui and Michal Kapalka. On the Correctness of Transactional Memory. In Chatterjee and Scott [175], pages 175–184. ISBN 978-1-59593-795-7. doi: 10.1145/1345206.1345233. URL `http://dl.acm.org/citation.cfm?id=1345206`.

[180] Phil McGachey, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Vijay Menon, Bratin Saha, and Tatiana Shpeisman. Concurrent GC Leveraging Transactional Memory. In Chatterjee and Scott [175], pages 217–226. ISBN 978-1-59593-795-7. doi: 10.1145/1345206.1345238. URL `http://dl.acm.org/citation.cfm?id=1345206`.

[181] Maurice Herlihy and Eric Koskinen. Transactional Boosting: A Methodology for Highly-concurrent Transactional Objects. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 207–216, New York, N.Y., USA, 2008. ACM. ISBN 978-1-59593-795-7. doi: 10.1145/1345206.1345237.

[182] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. Ordering-based Semantics for Software Transactional Memory. In *OPODIS '08: Proceedings of the 12th International Conference on Principles of Distributed Systems*, pages 275–294, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-92220-9. doi: 10.1007/978-3-540-92221-6_19.

[183] Lee Baugh, Naveen Neelakantam, and Craig Zilles. Using Hardware Memory Protection to Build a High-performance, Strongly-atomic Hybrid Transactional Memory. *ACM SIGARCH Computer Architecture News*, 36(3):115–126, 2008.

[184] Dave Dice, Maurice Herlihy, Doug Lea, Yossi Lev, Victor Luchangco, Wayne Mesard, Mark Moir, Kevin Moore, and Dan Nussbaum. Applications of the Adaptive Transactional Memory Test Platform. In *TRANSACT '08: 3rd Workshop on Transactional Computing*, February 2008.

[185] Enrique Vallejo, Tim Harris, Adrián Cristal, O Unsal, and Mateo Valero. Hybrid Transactional Memory to Accelerate Safe Lock-based Transactions. In *Workshop on transactional computing (TRANSACT)*, 2008.

[186] Advanced Micro Devices, Inc. Advanced Synchronization Facility: Proposed Architectural Specificaiton. Technical Report Publication #45432, rev. 2.1, Advanced Micro Devices, Inc, March 2009. Available as developer.amd.com/assets/45432-ASF_Spec_2.1.pdf.

[187] Geoffrey Blake, Ronald Dreslinski, and Trevor Mudge. Proactive Transaction Scheduling for Contention Management. In *Proceedings of the 42nd IEEE/ACM International Symposium on Microarchitecture*, December 2009. ISBN 978-1-60558-798-1.

[188] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay. Rock: A High-performance Sparc CMT Processor. *IEEE Micro*, 29(2):6–16, 2009. doi: 10.1109/MM.2009.34.

[189] Maria Couceiro, Paolo Romano, Nuno Carvalho, and Luís E. T. Rodrigues. D2STM: Dependable Distributed Software Transactional Memory. In *2009 15th IEEE Pacific Rim International Symposium on Dependable Computing, PRDC 2009, Shanghai, China, 16-18 November 2009*, pages 307–313. IEEE Computer Society, 2009. doi: 10.1109/PRDC.2009.55.

[190] Own Hofmann, Christopher Rossbach, and Emmett Witchel. Maximum Benefit from a Minimal HTM. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, Long Beach, CA, March 2009.

[191] Ryan Johnson, Manos Athanassoulis, Radu Stoica, and Anastasia Ailamaki. A New Look at the Roles of Spinning and Blocking. In Peter A. Boncz and Kenneth A. Ross, editors, *Proceedings of the Fifth International Workshop on Data Management on New Hardware, DaMoN 2009, Providence, Rhode Island, USA, June 28, 2009*, pages 21–26. ACM, 2009. doi: 10.1145/1565694.1565700.

[192] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting Phase Change Memory As a Scalable Dram Alternative. In Stephen W. Keckler and Luiz André Barroso, editors, *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, pages 2–13. ACM, 2009. doi: 10.1145/1555754.1555758.

[193] Scott Owens, Susmit Sarkar, and Peter Sewell. A Better X86 Memory Model: X86-TSO. In *TPHOLs '09: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, pages 391–407, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03358-2. doi: 10.1007/978-3-642-03359-9_27.

[194] Donald Porter, Owen Hofmann, Christopher Rossbach, Alexander Benn, and Emmett Witchel. Operating System Transactions. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, October 2009.

[195] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable High Performance Main Memory System Using Phase-change Memory Technology. In Stephen W. Keckler and Luiz André Barroso, editors, *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, pages 24–33. ACM, 2009. doi: 10.1145/1555754.1555760.

[196] Sasa Tomic, Cristian Perfumo, Chinmay Eishan Kulkarni, Adrià Armejach, Adrián Cristal, Osman S. Unsal, Tim Harris, and Mateo Valero. Eazyhtm: Eager-lazy Hardware Transactional Memory. In David H. Albonesi, Margaret Martonosi, David I. August, and José F. Martínez, editors, *42st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42 2009), December 12-16, 2009, New York, New York, USA*, pages 145–155. ACM, 2009. doi: 10.1145/1669112.1669132.

[197] Haris Volos, Adam Welc, Ali-Reza Adl-Tabatabai, Tatiana Shpeisman, Xinmin Tian, and Ravi Narayanaswamy. NePalTM: Design and Implementation of Nested Parallelism for Transactional Memory Systems. In *23rd European Conference on Object-Oriented Programming*, Genova, Italy, July 2009.

[198] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology. In Stephen W. Keckler and Luiz André Barroso, editors, *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, pages 14–23. ACM, 2009. doi: 10.1145/1555754.1555759.

[199] Cliff Click. Azul's Experiences with Hardware Transactional Memory. In *HP Labs - Bay Area Workshop on Transactional Memory*, January 2009.

[200] Shantanu Gupta, Florin Sultan, Srihari Cadambi, Franjo Ivancic, and Martin Rötteler. Using Hardware Transactional Memory for Data Race Detection. In *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*, pages 1–11. IEEE, 2009. doi: 10.1109/IPDPS.2009.5161006. URL `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5136864`.

[201] David Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early Experience with a Commercial Hardware Transactional Memory Implementation. In *ASPLOS*, 2009.

[202] Vladimir Gajinov, Ferad Zyulkyarov, Osman S. Unsal, Adrian Cristal, Eduard Ayguade, Tim Harris, and Mateo Valero. QuakeTM: Parallelizing a Complex Sequential Application Using Transactional Memory. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 126–135, New York, N.Y., USA, 2009. ACM. ISBN 978-1-60558-498-0. doi: 10.1145/1542275.1542298.

[203] J. Rubén Titos Gil, Manuel E. Acacio, and José Manuel García Carrasco. Speculation-based Conflict Resolution in Hardware Transactional Memory. In *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*, pages 1–12. IEEE, 2009. doi: 10.1109/IPDPS.2009.5161021.

[204] Seunghwa Kang and David A. Bader. An Efficient Transactional Memory Algorithm for Computing Minimum Spanning Forest of Sparse Graphs. In *PPoPP*, 2009.

[205] Marc Lupon, Grigorios Magklis, and Antonio González. FASTM: A Log-based Hardware Transactional Memory with Fast Abort Recovery. In *PACT 2009, Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques, 12-16 September 2009, Raleigh, North Carolina, USA*, pages 293–302. IEEE Computer Society, 2009. doi: 10.1109/PACT.2009.19.

[206] Ferad Zyulkyarov, Vladimir Gajinov, Osman S. Unsal, Adrián Cristal, Eduard Ayguadé, Tim Harris, and Mateo Valero. Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 25–34, New York, N.Y., USA, 2009. ACM. ISBN 978-1-60558-397-6. doi: 10.1145/1504176.1504183.

[207] Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Elastic Transactions. In *Proceedings of the 23rd International Symposum on Distributed Computing (DISC'09)*, volume 5805 of *LNCS*, pages 93–107. Springer-Verlag, September 2009.

[208] Hui Zeng, Matt Yourst, Kanad Ghose, and Dmitry Ponomarev. Mptlsim: A Simulator for X86 Multicore Processors. In *Proceedings of the 46th Annual Design Automation Conference*, pages 226–231. ACM, 2009.

[209] *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. AMD Corp., 3.17 edition, June 2010.

[210] Yehuda Afek, Ulrich Drepper, Pascal Felber, Christof Fetzer, Vincent Gramoli, Michael Hohmuth, Etienne Riviere, Per Stenström, Osman S. Unsal, Walther Maldonado, Derin Harmanci, Patrick Marlier, Stephan Diestelhorst, Martin Pohlack, Adrián Cristal, Ibrahim Hur, Aleksandar Dragojevic, Rachid Guerraoui, Michal Kapalka, Sasa Tomic, Guy Korland, Nir Shavit, Martin Nowack, and Torvald Riegel. The Velox Transactional Memory Stack. *IEEE Micro*, 30(5):76–87, 2010. doi: 10.1109/MM.2010.80.

[211] Woongki Baek, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. Implementing and Evaluating Nested Parallel Transactions in Software Transactional Memory. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.

[212] Joao Barreto, Aleksandar Dragojevic, Paulo Ferreira, Rachid Guerraoui, and Michal Kapalka. Leveraging Parallel Nesting in Transactional Memory. In *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming*, Bangalore, India, January 2010.

[213] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne Riviere. Evaluation of AMD's Advanced Synchronization Facility within a Complete Transactional Memory Stack. In *Proceedings of the EuroSys2010 Conference*, Paris, France, April 2010.

[214] Jaewoong Chung, Luke Yen, Stephan Diestelhorst, Martin Pohlack, Michael Hohmuth, Dan Grossman, and David Christie. ASF: AMD64 Extension for Lock-free Data Structures and Transactional Memory. In *Proceedings of the 43rd IEEE/ACM International Symposium on Microarchitecture*, Atlanta, GA, December 2010.

[215] Jaewoong Chung, David Christie, Martin Pohlack, Stephan Diestelhorst, Michael Hohmuth, and Luke Yen. Compilation of Thoughts about AMD Advanced Synchronization Facility and First-Generation Hardware Transactional Memory Support. In *Proceedings of the 5th ACM SIGPLAN Workshop on Transactional Computing*, Paris, France, April 2010.

[216] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor. *IEEE Micro*, 30(2):16–29, 2010. doi: 10.1109/MM.2010.31.

[217] Luke Dalessandro, Michael Spear, and Michael L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming*, Bangalore, India, January 2010.

[218] Luke Dalessandro, David Dice, Michael L. Scott, Nir Shavit, and Michael F. Spear. Transactional Mutex Locks. In Pasqua D'Ambra, Mario Rosario Guarracino, and Domenico Talia, editors, *Euro-Par 2010 - Parallel Processing, 16th International Euro-Par Conference, Ischia, Italy, August 31 - September 3, 2010, Proceedings, Part II*, volume 6272 of *Lecture Notes in Computer Science*, pages 2–13. Springer, 2010. doi: 10.1007/978-3-642-15291-7_2.

[219] Dave Dice, Yossi Lev, Virendra Marathe, Mark Moir, Marek Olszewski, and Dan Nussbaum. Simplifying Concurrent Algorithms by Exploiting Hardware TM. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.

[220] Stephan Diestelhorst, Martin Pohlack, Michael Hohmuth, Dave Christie, Jae-Woong Chung, and Luke Yen. Implementing AMD's Advanced Synchronization Facility in an Out-of-Order x86 Core. In *Proceedings of the 5th ACM SIGPLAN Workshop on Transactional Computing*, Paris, France, April 2010.

[221] Stephan Diestelhorst, Michael Hohmuth, and Martin Pohlack. Sane Semantics of Best-effort Hardware Transactional Memory. In *2nd Workshop on the Theory of Transactional Memory (WTTM)*, 2010.

[222] Francois Carouge and Michael Spear. A Scalable Lock-Free Universal Construction with Best Effort Transactional Hardware. In *Proceedings of the 24th International Symposium on Distributed Computing*, Cambridge, MA, September 2010.

[223] Cesare Ferri, Samantha Wood, Tali Moreshet, R. Iris Bahar, and Maurice Herlihy. Embedded-tm: Energy and Complexity-effective Hardware Transactional Memory for Embedded Multicore Systems. *J. Parallel Distrib. Comput.*, 70(10):1042–1052, 2010. doi: 10.1016/j.jpdc.2010.02.003.

[224] Tim Harris, James R. Larus, and Ravi Rajwar. *Transactional Memory, 2nd Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010. doi: 10.2200/S00272ED1V01Y201006CAC011.

[225] Sungpack Hong, Tayo Oguntebi, Jared Casper, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. Eigenbench: A Simple Exploration Tool for Orthogonal TM Characteristics. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Atlanta, GA, December 2010.

[226] Guy Korland, Nir Shavit, and Pascal Felber. Deuce: Noninvasive Software Transactional Memory in Java. *Transactions on HiPEAC*, 5(2):43, 2010.

[227] Daniel Lupei, Bogdan Simion, Don Pinto, Matthew Misler, Mihai Burcea, William Krick, and Cristiana Amza. Transactional Memory Support for Scalable and Transparent Parallelization of Multiplayer Games. In Christine Morin and Gilles Muller, editors, *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010*, pages 41–54. ACM, 2010. doi: 10.1145/1755913.1755919.

[228] Marc Lupon, Grigorios Magklis, and Antonio González. A Dynamically Adaptable Hardware Transactional Memory. In *43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2010, 4-8 December 2010, Atlanta, Georgia, USA*, pages 27–38. IEEE Computer Society, 2010. doi: 10.1109/MICRO.2010.23.

[229] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald III, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A Distributed Parallel Simulator for Multicores. In Matthew T. Jacob, Chita R. Das, and Pradip Bose, editors, *16th International Conference on High-Performance Computer Architecture (HPCA-16 2010), 9-14 January 2010, Bangalore, India*, pages 1–12. IEEE Computer Society, 2010. doi: 10.1109/HPCA.2010.5416635.

[230] Mathias Payer and Thomas R. Gross. Generating Low-overhead Dynamic Binary Translators. In Gadi Haber, Dilma Da Silva, and Ethan L. Miller, editors, *Proceedings of of SYSTOR 2010: The 3rd Annual Haifa Experimental Systems Conference, Haifa, Israel, May 24-26, 2010*, ACM International Conference Proceeding Series. ACM, 2010. doi: 10.1145/1815695.1815724.

[231] Paolo Romano, Luís E. T. Rodrigues, Nuno Carvalho, and João P. Cachopo. Cloud-tm: Harnessing the Cloud with Distributed Transactional Memories. *Operating Systems Review*, 44(2):1–6, 2010. doi: 10.1145/1773912.1773914.

[232] Michael Spear. Lightweight, Robust Adaptivity for Software Transactional Memory. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.

[233] Fuad Tabba. Adding Concurrency in Python Using a Commercial Processor's Hardware Transactional Memory Support. *SIGARCH Computer Architecture News*, 38(5):12–19, 2010. doi: 10.1145/1978907.1978911.

[234] Gil Tene. Azul Puts the Zing in Java, December 2010. URL `https://www.infoq.com/interviews/gil-tene-azul-zing`.

[235] Takayuki Usui, Reimer Behrends, Jacob Evans, and Yannis Smaragdakis. Adaptive Locks: Combining Transactions and Locks for Efficient Concurrency. *J. Parallel Distrib. Comput.*, 70(10):1009–1023, 2010. doi: 10.1016/j.jpdc.2010.02.006.

[236] Colin Blundell, Arun Raghavan, and Milo M. K. Martin. RETCON: Transactional Repair without Replay. In André Seznec, Uri C. Weiser, and Ronny Ronen, editors, *37th International Symposium on Computer Architecture (ISCA 2010), June 19-23, 2010, Saint-Malo, France*, pages 258–269.

ACM, 2010. ISBN 978-1-4503-0053-7. doi: 10.1145/1815961.1815995. URL `http://dl.acm.org/citation.cfm?id=1815961`.

[237] Stephan Diestelhorst. Complications of Transactional Memory Implementations in Modern Micprocessors. Presented as a keynote at VELOX General Assembly and Seminar in Champery, Switzerland, 2010.

[238] Pascal Felber, Christof Fetzer, Patrick Marlier, Martin Nowack, and Torvald Riegel. Brief Announcement: Hybrid Time-based Transactional Memory. In Nancy Lynch and Alexander Shvartsman, editors, *Distributed Computing*, volume 6343 of *Lecture Notes in Computer Science*, pages 124–126. Springer Berlin / Heidelberg, 2010. doi: 10.1007/978-3-642-15763-9_11.

[239] Jake Edge. The Trouble with the Tsc. `http://lwn.net/Articles/388188`, May 2010.

[240] Yehuda Afek, Hillel Avni, and Nir Shavit. Towards Consistency Oblivious Programming. In Antonio Fernández Anta, Giuseppe Lipari, and Matthieu Roy, editors, *Principles of Distributed Systems - 15th International Conference, OPODIS 2011, Toulouse, France, December 13-16, 2011. Proceedings*, volume 7109 of *Lecture Notes in Computer Science*, pages 65–79. Springer, 2011. doi: 10.1007/978-3-642-25873-2_6.

[241] Nathan L. Binkert, Bradford M. Beckmann, Gabriel Black, Steven K. Reinhardt, Ali G. Saidi, Arkaprava Basu, Joel Hestness, Derek Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib Bin Altaf, Nilay Vaish, Mark D. Hill, and David A. Wood. The Gem5 Simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, 2011. doi: 10.1145/2024716.2024718.

[242] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-core Simulation. In Scott Lathrop, Jim Costa, and William Kramer, editors, *Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12-18, 2011*, pages 52:1–52:12. ACM, 2011. doi: 10.1145/2063384.2063454.

[243] Jared Casper, Tayo Oguntebi, Sungpack Hong, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. Hardware Acceleration of Transactional Memory on Commodity Systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, Calif., March 2011.

[244] Luke Dalessandro, François Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael F. Spear. Hybrid Norec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory. In Rajiv Gupta and Todd C. Mowry, editors, *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, pages 39–52. ACM, 2011. doi: 10.1145/1950365.1950373.

[245] Aleksandar Dragojevic, Maurice Herlihy, Yossi Lev, and Mark Moir. On the Power of Hardware Transactional Memory to Simplify Memory Management. In Cyril Gavoille and Pierre Fraigniaud, editors, *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, PODC 2011, San Jose, CA, USA, June 6-8, 2011*, pages 99–108. ACM, 2011. doi: 10.1145/1993806.1993821.

[246] Aleksandar Dragojevic, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why STM Can Be More Than a Research Toy. *Commun. ACM*, 54(4):70–77, 2011. doi: 10.1145/1924421.1924440.

[247] Cesare Ferri, Andrea Marongiu, Benjamin Lipton, R. Iris Bahar, Tali Moreshet, Luca Benini, and Maurice Herlihy. Soc-tm: Integrated HW/SW Support for Transactional Memory Programming on Embedded Mpsocs. In Robert P. Dick and Jan Madsen, editors, *Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2011, part of ESWeek '11 Seventh Embedded Systems Week, Taipei, Taiwan, 9-14 October, 2011*, pages 39–48. ACM, 2011. doi: 10.1145/2039370.2039380.

[248] Wilson W. L. Fung, Inderpreet Singh, Andrew Brownsword, and Tor M. Aamodt. Hardware Transactional Memory for GPU Architectures. In Carlo Galuzzi, Luigi Carro, Andreas Moshovos, and Milos Prvulovic, editors, *44rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2011, Porto Alegre, Brazil, December 3-7, 2011*, pages 296–307. ACM, 2011. doi: 10.1145/2155620.2155655.

[249] J. Rubén Titos Gil, Anurag Negi, Manuel E. Acacio, José M. García, and Per Stenström. ZEBRA: a Data-centric, Hybrid-policy Hardware Transactional Memory Design. In David K. Lowenthal, Bronis R. de Supinski, and Sally A. McKee, editors, *Proceedings of the 25th International Conference on Supercomputing, 2011, Tucson, AZ, USA, May 31 - June 04, 2011*, pages 53–62. ACM, 2011. doi: 10.1145/1995896.1995906.

[250] The IBM Blue Gene Team. The Blue Gene/Q Compute Chip. In *Proceedings of the 23rd Hot Chips Conference*, Palo Alto, CA, August 2011.

[251] Gokcen Kestor, Vasileios Karakostas, Osman S. Unsal, Adrián Cristal, Ibrahim Hur, and Mateo Valero. RMS-TM: a Comprehensive Benchmark Suite for Transactional Memory Systems. In Samuel Kounev, Vittorio Cortellessa, Raffaela Mirandola, and David J. Lilja, editors, *ICPE'11 - Second Joint WOSP/SIPEW International Conference on Performance Engineering, Karlsruhe, Germany, March 14-16, 2011*, pages 335–346. ACM, 2011. doi: 10.1145/1958746.1958795.

[252] Victor Pankratius and Ali-Reza Adl-Tabatabai. A Study of Transactional Memory vs. Locks in Practice. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, San Jose, CA, June 2011.

[253] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. MARSS: a Full System Simulator for Multicore X86 Cpus. In Leon Stok, Nikil D. Dutt, and Soha Hassoun, editors, *Proceedings of the 48th Design Automation Conference, DAC 2011, San Diego, California, USA, June 5-10, 2011*, pages 1050–1055. ACM, 2011. doi: 10.1145/2024724.2024954.

[254] Martin Pohlack and Stephan Diestelhorst. From Lightweight Hardware Transactional Memory to Lightweight Lock Elision. In *Proceedings of the 6th ACM SIGPLAN Workshop on Transactional Computing*, San Jose, CA, June 2011.

[255] Torvald Riegel, Patrick Marlier, Martin Nowack, Pascal Felber, and Christof Fetzer. Optimizing Hybrid Transactional Memory: The Importance of Nonspeculative Operations. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, June 2011.

[256] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011. doi: 10.2200/S00346ED1V01Y201104CAC016.

[257] *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions*. Advanced Micro Devices, Inc., 3.18 edition, March 2012.

[258] *Intel Architecture Instruction Set Extensions Programming Reference*. Intel Corp., 319433-012a edition, February 2012.

[259] *Z/architecture Principles of Operations*, sa22-7832-09 edition, September 2012. URL `http://publibfi.boulder.ibm.com/epubs/pdf/dz9zr009.pdf`.

[260] AMD. Simnow(tm) Simulator, 2012. URL `https://developer.amd.com/simnow-simulator/`.

[261] David Dice, Virendra J. Marathe, and Nir Shavit. Lock Cohorting: A General Technique for Designing NUMA Locks. In J. Ramanujam and P. Sadayappan, editors, *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*, pages 247–256. ACM, 2012. doi: 10.1145/2145816.2145848.

[262] Stephan Diestelhorst. PTLsim-ASF Repository, October 2012. URL `https://github.com/stephand/ptlsim/tree/ptlsim_asf`.

[263] Stephan Diestelhorst and Martin Pohlack. Safely Accessing Time Stamps in Transactions. *4th Workshop on the Theory of Transactional Memory*, July 2012.

[264] Michael Ferdman, Almutaz Adileh, Yusuf Onur Koçberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In Tim Harris and Michael L. Scott, editors, *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, pages 37–48. ACM, 2012. doi: 10.1145/2150976.2150982.

[265] Wilson Wai Lun Fung, Inderpreet Singh, Andrew Brownsword, and Tor M. Aamodt. Kilo TM: Hardware Transactional Memory for GPU Architectures. *IEEE Micro*, 32(3):7–16, 2012. doi: 10.1109/MM.2012.16.

[266] Wilson WL Fung, Inderpreet Singh, and Tor M Aamodt. Kilo Tm Correctness: Aba Tolerance and Validation-commit Indivisibility. Technical report, Technical report, University of British Columbia, 2012. http://www. ece. ubc. ca/ aamodt/papers/wwlfung. tr2012. pdf, 2012.

[267] Ruud A. Haring, Martin Ohmacht, Thomas W. Fox, Michael Gschwind, David L. Satterfield, Krishnan Sugavanam, Paul Coteus, Philip Heidelberger, Matthias A. Blumrich, Robert W. Wisniewski, Alan Gara, George L.-T. Chiu, Peter A. Boyle, Norman H. Christ, and Changhoan Kim. The IBM Blue Gene/q Compute Chip. *IEEE Micro*, 32(2):48–60, 2012. doi: 10.1109/MM.2011.108.

[268] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, February 2012. URL `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372`.

[269] Balaji Iyengar, Gil Tene, Michael Wolf, and Edward F. Gehringer. The Collie: A Wait-free Compacting Collector. In Martin T. Vechev and Kathryn S. McKinley, editors, *International Symposium on Memory Management, ISMM '12, Beijing, China, June 15-16, 2012*, pages 85–96. ACM, 2012. doi: 10.1145/2258996.2259009.

[270] Christian Jacobi, Timothy J. Slegel, and Dan F. Greiner. Transactional Memory Architecture and Implementation for IBM System Z. In *45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2012, Vancouver, BC, Canada, December 1-5, 2012*, pages 25–36. IEEE Computer Society, 2012. doi: 10.1109/MICRO.2012.12.

[271] Christian Jacobi, Timothy Slegel, and Dan Greiner. Transactional Memory Architecture and Implementation for IBM System z. In *45th Int. Symp. On Microarchitecture*, 2012.

[272] Andi Kleen. Adding Lock Elision to Linux. In *Linux Plumbers Conference, August*, 2012.

[273] Andi Kleen. Tsx Lock Elision for Glibc (github Repository), December 2012. URL `https://github.com/andikleen/glibc`.

[274] Yujie Liu, Stephan Diestelhorst, and Michael Spear. Delegation and Nesting in Best Effort Hardware Transactional Memory. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, Pittsburgh, PA, June 2012.

[275] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia L. Lawall, and Gilles Muller. Remote Core Locking: Migrating Critical-section Execution to Improve the Performance of Multithreaded Applications. In Gernot Heiser and Wilson C. Hsieh, editors, *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, pages 65–76. USENIX Association, 2012. URL `https://www.usenix.org/conference/atc12/technical-sessions/presentation/lozi`.

[276] M Mitran and V Vokhshori. Evaluating the Zec12 Transactional Execution Facility. *IBM Systems Magazine*, 2012.

[277] Jeff Preshing. A Look Back at Single-threaded Cpu Performance. February 2012. URL `http://preshing.com/20120208/a-look-back-at-single-threaded-cpu-performance/`.

[278] James Reinders. Transactional Synchronization in Haswell. Blog Post, February 2012. URL `https://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell`.

[279] Daniel Sánchez and Christos Kozyrakis. SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding. In *18th IEEE International Symposium on High Performance Computer Architecture, HPCA 2012, New Orleans, LA, USA, 25-29 February, 2012*, pages 129–140. IEEE Computer Society, 2012. doi: 10.1109/HPCA.2012.6168950.

[280] Martin Schindewolf, Barna L. Bihari, John C. Gyllenhaal, Martin Schulz, Amy Wang, and Wolfgang Karl. What Scientific Applications Can Benefit from Hardware Transactional Memory? In Jeffrey K. Hollingsworth, editor, *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*, page 90. IEEE/ACM, 2012. doi: 10.1109/SC.2012.113.

[281] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raúl Silvera, and Maged M. Michael. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In Pen-Chung Yew, Sangyeun Cho, Luiz DeRose, and David J. Lilja, editors, *International Conference on Parallel Architectures and Compilation Techniques, PACT '12, Minneapolis, MN, USA - September 19 - 23, 2012*, pages 127–136. ACM, 2012. doi: 10.1145/2370816.2370836.

[282] Lihang Zhao, Woojin Choi, and Jeff Draper. SEL-TM: Selective Eager-lazy Management for Improved Concurrency in Transactional Memory. In *26th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2012, Shanghai, China, May 21-25, 2012*, pages 95–106. IEEE Computer Society, 2012. doi: 10.1109/IPDPS.2012.19.

[283] Andrew Danowitz, Kyle Kelley, James Mao, John P. Stevenson, and Mark Horowitz. Cpu Db: Recording Microprocessor History. *Commun. ACM*, 55(4):55–63, April 2012. ISSN 0001-0782. doi: 10.1145/2133806.2133822.

[284] Intel Architecture Instruction Set Extensions Programming Reference. Intel Corp., February 2012.

[285] Ravi Rajwar. Personal communication, 2012.

[286] *Intel(r) Core(tm) I7-4650u Processor*. Intel, June 2013. URL `https://ark.intel.com/products/75114/Intel-Core-i7-4650U-Processor-4M-Cache-up-to-3_30-GHz`.

[287] Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Q. Le. Robust Architectural Support for Transactional Memory in the Power Architecture. In Avi Mendelson, editor, *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*, pages 225–236. ACM, 2013. doi: 10.1145/2485922.2485942.

[288] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything You Always Wanted to Know about Synchronization but Were Afraid to Ask. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 33–48. ACM, 2013. doi: 10.1145/2517349.2522714.

[289] Stephan Diestelhorst, Martin Nowack, Michael F. Spear, and Christof Fetzer. Brief Announcement: Between All and Nothing - Versatile Aborts in Hardware Transactional Memory. In Guy E. Blelloch and Berthold Vöcking, editors, *25th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '13, Montreal, QC, Canada - July 23 - 25, 2013*, pages 108–110. ACM, 2013. doi: 10.1145/2486159.2486165.

[290] Stephan Diestelhorst. Extending Openjdk's Hotspot Jvm with Support for Lock Elision with Asf. Bitbucket Code Repository, August 2013. URL `https://bitbucket.org/stephand/hotspot-asfsle/branch/ASF-SLE`.

[291] Wilson W. L. Fung and Tor M. Aamodt. Energy Efficient GPU Transactional Memory Via Space-time Optimizations. In Matthew K. Farrens and Christos Kozyrakis, editors, *The 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-46, Davis, CA, USA, December 7-11, 2013*, pages 408–420. ACM, 2013. doi: 10.1145/2540708.2540743.

[292] Victor Luchangco, Jens Maurer, Mark Moir, Hans Boehm, Justin Gottschlich, Maged Michael, Torvald Riegel, Michael Scott, Tatiana Shpeisman, Michael Spear, and Michael Wong. *Transactional Memory Support for C++*, volume N3718. August 2013. URL `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3718.pdf`.

[293] Anurag Negi and J. Rubén Titos Gil. Scin-cache: Fast Speculative Versioning in Multithreaded Cores. *TACO*, 9(4):58:1–58:26, 2013. doi: 10.1145/2400682.2400717.

[294] Wenjia Ruan, Yujie Liu, and Michael F. Spear. Boosting Timestamp-based Transactional Memory by Exploiting Hardware Cycle Counters. *TACO*, 10(4):40:1–40:21, 2013. doi: 10.1145/2541228.2555297.

[295] Daniel Sánchez and Christos Kozyrakis. Zsim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. In Avi Mendelson, editor, *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*, pages 475–486. ACM, 2013. doi: 10.1145/2485922.2485963.

[296] C. Kevin Shum, Fadi Busaba, and Christian Jacobi. IBM Zec12: The Third-generation High-frequency Mainframe Microprocessor. *IEEE Micro*, 33(2):38–47, 2013. doi: 10.1109/MM.2013.9.

[297] Inderpreet Singh, Arrvindh Shriraman, Wilson W. L. Fung, Mike O'Connor, and Tor M. Aamodt. Cache Coherence for GPU Architectures. In *19th IEEE International Symposium on High Performance Computer Architecture, HPCA 2013, Shenzhen, China, February 23-27, 2013*, pages 578–590. IEEE Computer Society, 2013. doi: 10.1109/HPCA.2013.6522351.

[298] Alexandre Skyrme and Noemi Rodriguez. From Locks to Transactional Memory: Lessons Learned from Porting a Real-world Application. In *the 8th ACM SIGPLAN Workshop on Transactional Computing*, 2013.

[299] Vesna Smiljkovic, Martin Nowack, Neboja Miletic, Tim Harris, Osman S. Ünsal, Adrián Cristal, and Mateo Valero. Tm-dietlibc: A Tm-aware Real-world System Library. In *27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2013, Cambridge, MA, USA, May 20-24, 2013*, pages 1266–1274. IEEE Computer Society, 2013. doi: 10.1109/IPDPS.2013.45.

[300] Jons-Tobias Wamhoff, Christof Fetzer, Pascal Felber, Etienne Rivière, and Gilles Muller. Fastlane: Improving Performance of Software Transactional Memory for Low Thread Counts. In Alex Nicolau, Xiaowei Shen, Saman P. Amarasinghe, and Richard W. Vuduc, editors, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, pages 113–122. ACM, 2013. doi: 10.1145/2442516.2442528.

[301] Jons-Tobias Wamhoff, Stephan Diestelhorst, Christof Fetzer, Patrick Marlier, Pascal Felber, and Dave Dice. Selective Core Boosting: The Return of the Turbo Button. techreport ISSN 1430-211X, TUD-Fl13-02, TU Dresden, November 2013.

[302] Zhichao Yan, Hong Jiang, Yujuan Tan, and Dan Feng. An Integrated Pseudo-associativity and Relaxed-order Approach to Hardware Transactional Memory. *TACO*, 9(4):42:1–42:26, 2013. doi: 10.1145/2400682.2400701.

[303] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance Evaluation of Intel Transactional Synchronization Extensions for High-performance Computing. In William Gropp and Satoshi Matsuoka, editors, *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*, pages 19:1–19:11. ACM, 2013. ISBN 978-1-4503-2378-9. doi: 10.1145/2503210.2503232. URL `http://dl.acm.org/citation.cfm?id=2503210`.

[304] Stephan Diestelhorst. Marss86 ASF Repository, June 2013. URL `https://bitbucket.org/stephand/marss86-asf/branch/master`.

[305] Alexander Matveev and Nir Shavit. Reduced Hardware Transactions: A New Approach to Hybrid Transactional Memory. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, pages 11–22. ACM, 2013.

[306] Dan Alistarh, Patrick Eugster, Maurice Herlihy, Alexander Matveev, and Nir Shavit. Stacktrack: An Automated Transactional Approach to Concurrent Memory Reclamation. In Dick C. A. Bulterman, Herbert Bos, Antony I. T. Rowstron, and Peter Druschel, editors, *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, pages 25:1–25:14. ACM, 2014. doi: 10.1145/2592798.2592808.

[307] Hillel Avni and Adi Suissa-Peleg. COP Composition Using Transaction Suspension in the Compiler. In Fabian Kuhn, editor, *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, volume 8784 of *Lecture Notes in Computer*

Science, pages 550–552. Springer, 2014. URL http://link.springer.com/content/pdf/bbm%3A978-3-662-45174-8%2F1.pdf.

[308] Hillel Avni and Bradley C Kuszmaul. Improving Htm Scaling with Consistency-oblivious Programming. In *9th Workshop on Transactional Computing, TRANSACT*, volume 14, 2014.

[309] Dhruva R. Chakrabarti, Hans-Juergen Boehm, and Kumud Bhandari. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In Andrew P. Black and Todd D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 433–452. ACM, 2014. doi: 10.1145/2660193.2660224.

[310] Dave Dice, Alex Kogan, Yossi Lev, Timothy Merrifield, and Mark Moir. Adaptive Integration of Hardware and Software Lock Elision Techniques. In Guy E. Blelloch and Peter Sanders, editors, *26th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '14, Prague, Czech Republic - June 23 - 25, 2014*, pages 188–197. ACM, 2014. doi: 10.1145/2612669.2612696.

[311] Dave Dice, Timothy L Harris, Alex Kogan, Yossi Lev, and Mark Moir. Pitfalls of Lazy Subscription. In *Proceedings of the 6th Workshop on the Theory of Transactional Memory, Paris, France*, 2014.

[312] Nuno Diegues and Paolo Romano. Self-tuning Intel Transactional Synchronization Extensions. In Xiaoyun Zhu, Giuliano Casale, and Xiaohui Gu, editors, *11th International Conference on Autonomic Computing, ICAC '14, Philadelphia, PA, USA, June 18-20, 2014.*, pages 209–219. USENIX Association, 2014. URL https://www.usenix.org/conference/icac14/technical-sessions/presentation/diegues.

[313] Marco Elver and Vijay Nagarajan. TSO-CC: Consistency Directed Cache Coherence for TSO. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*, pages 165–176. IEEE Computer Society, 2014. doi: 10.1109/HPCA.2014.6835927.

[314] Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. Optimistic Transactional Boosting. In José E. Moreira and James R. Larus, editors, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, pages 387–388. ACM, 2014. doi: 10.1145/2555243.2555283.

[315] Azeem Jiva and Vladimir Kozlov. Use Intel Rtm Instructions for Locks. OpenJDK Bug Report, March 2014. URL https://bugs.openjdk.java.net/browse/JDK-8031320.

[316] Tomas Karnagel, Roman Dementiev, Ravi Rajwar, Konrad Lai, Thomas Legler, Benjamin Schlegel, and Wolfgang Lehner. Improving In-memory Database Index Performance with Intel® Transactional Synchronization Extensions. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*, pages 476–487. IEEE Computer Society, 2014. doi: 10.1109/HPCA.2014.6835957.

[317] Andi Kleen. Lock Elision in Glibc, May 2014. URL https://01.org/blogs/2014/lock-elision-glibc.

[318] Andi Kleen. Scaling Existing Lock-based Applications with Lock Elision. *ACM Queue*, 12(1):20, 2014. doi: 10.1145/2576966.2579227.

[319] Viktor Leis, Alfons Kemper, and Thomas Neumann. Exploiting Hardware Transactional Memory in Main-memory Databases. In Isabel F. Cruz, Elena Ferrari, Yufei Tao, Elisa Bertino, and Goce Trajcevski, editors, *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 580–591. IEEE Computer Society, 2014. doi: 10.1109/ICDE.2014.6816683.

[320] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In Dick C. A. Bulterman, Herbert Bos, Antony I. T. Rowstron, and Peter Druschel, editors, *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, pages 27:1–27:14. ACM, 2014. doi: 10.1145/2592798.2592820.

[321] Yutao Liu, Yubin Xia, Haibing Guan, Binyu Zang, and Haibo Chen. Concurrent and Consistent Virtual Machine Introspection with Hardware Transactional Memory. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*, pages 416–427. IEEE Computer Society, 2014. doi: 10.1109/HPCA.2014.6835951.

[322] Rei Odaira, José G. Castaños, and Hisanobu Tomari. Eliminating Global Interpreter Locks in Ruby through Hardware Transactional Memory. In José E. Moreira and James R. Larus, editors, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, pages 131–142. ACM, 2014. doi: 10.1145/2555243.2555247.

[323] Rei Odaira and Takuya Nakaike. Thread-level Speculation on Off-the-shelf Hardware Transactional Memory. In *2014 IEEE International Symposium on Workload Characterization, IISWC 2014, Raleigh, NC, USA, October 26-28, 2014*, pages 212–221. IEEE Computer Society, 2014. doi: 10.1109/IISWC.2014.6983060.

[324] Oracle. Tools Enhancements in Jdk 8, 2014. URL `https://docs.oracle.com/javase/8/docs/technotes/tools/enhancements-8.html`.

[325] Carl G. Ritson, Tomoharu Ugawa, and Richard E. Jones. Exploring Garbage Collection with Haswell Hardware Transactional Memory. In David Grove and Samuel Z. Guyer, editors, *International Symposium on Memory Management, ISMM '14, Edinburgh, United Kingdom, June 12, 2014*, pages 105–115. ACM, 2014. doi: 10.1145/2602988.2602992.

[326] Wenjia Ruan, Yujie Liu, and Michael Spear. Stamp Need Not Be Considered Harmful. In *Ninth ACM SIGPLAN Workshop on Transactional Computing*, 2014.

[327] Wenjia Ruan, Trilok Vyas, Yujie Liu, and Michael Spear. Transactionalizing Legacy Code: An Experience Report Using Gcc and Memcached. In *ACM SIGPLAN Notices*, volume 49, pages 399–412. ACM, 2014.

[328] Jons-Tobias Wamhoff, Stephan Diestelhorst, Christof Fetzer, Patrick Marlier, Pascal Felber, and Dave Dice. The TURBO Diaries: Application-controlled Frequency Scaling Explained. In Garth Gibson and Nickolai Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.*, pages 193–204. USENIX Association, 2014. URL `https://www.usenix.org/conference/atc14/technical-sessions/presentation/wamhoff`.

[329] Zhaoguo Wang, Hao Qian, Jinyang Li, and Haibo Chen. Using Restricted Transactional Memory to Build a Scalable In-memory Database. In Dick C. A. Bulterman, Herbert Bos, Antony I. T. Rowstron, and Peter Druschel, editors, *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, pages 26:1–26:15. ACM, 2014. doi: 10.1145/2592798.2592815.

[330] Richard M Yoo, Sandhya Viswanathan, Vivek R Deshpande, Christopher J Hughes, and Shirish Aundhe. Early Experience on Transactional Execution of Java Tm Programs Using Intelrg Transactional Synchronization Extensions. In *TRANSACT 2014 9th ACM SIGPLAN Workshop on Transactional Computing*, 2014.

[331] Irina Calciu, Tatiana Shpeisman, Gilles Pokam, and Maurice Herlihy. Improved Single Global Lock Fallback for Best-effort Hardware Transactional Memory. *9th ACM SIGPLAN Wkshp. on Transactional Computing*, 2014.

[332] Irina Calciu, Justin Gottschlich, Tatiana Shpeisman, Gilles Pokam, and Maurice Herlihy. Invyswell: A Hybrid Transactional Memory for Haswell's Restricted Transactional Memory. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 187–200. ACM, 2014.

[333] Dan Alistarh, Justin Kopinsky, Petr Kuznetsov, Srivatsan Ravi, and Nir Shavit. Inherent Limitations of Hybrid Transactional Memory. In *Distributed Computing*, pages 185–199. Springer, 2015.

[334] Hillel Avni, Eliezer Levy, and Avi Mendelson. Hardware Transactions in Nonvolatile Memory. In Yoram Moses, editor, *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, volume 9363 of *Lecture Notes in Computer Science*, pages 617–630. Springer, 2015. doi: 10.1007/978-3-662-48653-5_41.

[335] Roman Dementiev. Add Support for Htm Lock Elision for X86. DPDK-dev Mailing list, June 2015. URL `http://dpdk.org/ml/archives/dev/2015-June/018566.html`.

[336] Nuno Diegues and Paolo Romano. Self-tuning Intel Restricted Transactional Memory. *Parallel Computing*, 50:25–52, 2015. doi: 10.1016/j.parco.2015.10.001.

[337] Stephan Diestelhorst, Martin Nowack, M Spear, and C Fetzer. Between All and Nothing–versatile Aborts in Hardware Transactional Memory. In *10th Workshop on Transactional Computing (TRANSACT15)*, 2015.

[338] Le Guan, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang. Protecting Private Keys against Memory Disclosure Attacks Using Hardware Transactional Memory. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 3–19. IEEE Computer Society, 2015. doi: 10.1109/SP.2015.8.

[339] Bradley C. Kuszmaul. Supermalloc: A Super Fast Multithreaded Malloc for 64-bit Machines. In Antony L. Hosking and Michael D. Bond, editors, *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management, ISMM 2015, Portland, OR, USA, June 13-14, 2015*, pages 41–55. ACM, 2015. doi: 10.1145/2754169.2754178.

[340] Hung Q. Le, G. L. Guthrie, Derek Williams, Maged M. Michael, Brad Frey, William J. Starke, Cathy May, Rei Odaira, and Takuya Nakaike. Transactional Memory Support in the IBM POWER8 Processor. *IBM Journal of Research and Development*, 59(1), 2015. doi: 10.1147/JRD.2014.2380199.

[341] Darko Makreshanski, Justin J. Levandoski, and Ryan Stutsman. To Lock, Swap, or Elide: On the Interplay of Hardware Transactional Memory and Lock-free Indexing. *PVLDB*, 8(11):1298–1309, 2015. URL `http://www.vldb.org/pvldb/vol8/p1298-makreshanski.pdf`.

[342] Maged M. Michael. Portability Issues in Hardware Transactional Memory Implementations. In Michael Philippsen, Pascal Felber, Michael L. Scott, and J. Eliot B. Moss, editors, *Concurrent computing in the many-core era*, volume 5 of *Dagstuhl Reports*. Schloss Dagstuhl – Leibniz-Zentrum

für Informatik, Dagstuhl Publishing, Germany, 2015. URL `http://drops.dagstuhl.de/opus/volltexte/2015/5010/pdf/dagrep_v005_i001_p001_s15021.pdf`.

[343] Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged M. Michael, and Hisanobu Tomari. Quantitative Comparison of Hardware Transactional Memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In Deborah T. Marr and David H. Albonesi, editors, *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, pages 144–157. ACM, 2015. doi: 10.1145/2749469.2750403.

[344] Siddharth Nilakantan, Karthik Sangaiah, Ankit More, Giordano Salvador, Baris Taskin, and Mark Hempstead. Synchrotrace: Synchronization-aware Architecture-agnostic Traces for Light-weight Multicore Simulation. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2015, Philadelphia, PA, USA, March 29-31, 2015*, pages 278–287. IEEE Computer Society, 2015. doi: 10.1109/ISPASS.2015.7095813.

[345] Amy Wang, Matthew Gaudet, Peng Wu, Martin Ohmacht, José Nelson Amaral, Christopher Barton, Raúl Silvera, and Maged M. Michael. Software Support and Evaluation of Hardware Transactional Memory on Blue Gene/Q. *IEEE Trans. Computers*, 64(1):233–246, 2015. doi: 10.1109/TC.2013.190.

[346] Lingxiang Xiang and Michael L. Scott. Software Partitioning of Hardware Transactions. In Albert Cohen and David Grove, editors, *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, pages 76–86. ACM, 2015. doi: 10.1145/2688500.2688506.

[347] Minjia Zhang, Jipeng Huang, Man Cao, and Michael D. Bond. Low-overhead Software Transactional Memory with Progress Guarantees and Strong Semantics. In Albert Cohen and David Grove, editors, *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, pages 97–108. ACM, 2015. doi: 10.1145/2688500.2688510.

[348] intel. Ask for More from Your Data. Technical report, 2015. URL `https://www.intel.com/content/www/us/en/big-data/sap-hana-real-time-analytics-solution-brief.html`.

[349] Yehuda Afek, Alexander Matveev, Oscar R Moll, and Nir Shavit. Amalgamated Lock-elision. In *Distributed Computing*, pages 309–324. Springer, 2015.

[350] Wenjia Ruan and Michael Spear. Hybrid Transactional Memory Revisited. In *International Symposium on Distributed Computing*, pages 215–231. Springer, 2015.

[351] *Arm Architecture Reference Manual Armv8, for Armv8-a Architecture Profile*. Arm Limited, a.j edition, June 2016. URL `https://developer.arm.com/docs/ddi0487/a/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile`.

[352] *Arm Architecture Reference Manual Supplement Armv8.1, for Armv8-a Architecture Profile*. Arm Limited, a.b edition, June 2016. URL `https://developer.arm.com/products/architecture/a-profile/docs/ddi0557/latest/arm-architecture-reference-manual-supplement-armv81-for-armv8-a-architecture-profile`.

[353] *Ibm Power Isa(tm) Version 2.07 B*. IBM, June 2016. URL `https://openpowerfoundation.org/?resource_lib=ibm-power-isa-version-2-07-b`.

[354] Trevor Brown and Hillel Avni. Phytm: Persistent Hybrid Transactional Memory. *PVLDB*, 10(4): 409–420, 2016. URL `http://www.vldb.org/pvldb/vol10/p409-brown.pdf`.

[355] Diego Didona, Nuno Diegues, Anne-Marie Kermarrec, Rachid Guerraoui, Ricardo Neves, and Paolo Romano. Proteustm: Abstraction Meets Performance in Transactional Memory. In Tom Conte and Yuanyuan Zhou, editors, *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, pages 757–771. ACM, 2016. doi: 10.1145/2872362.2872385.

[356] Radhika Jagtap, Stephan Diestelhorst, and Andreas Hansson. Elastic Traces for Fast and Accurate System Performance Exploration. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2016, Uppsala, Sweden, April 17-19, 2016*, pages 147–148. IEEE Computer Society, 2016. doi: 10.1109/ISPASS.2016.7482084.

[357] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 380–392. ACM, 2016. doi: 10.1145/2976749.2978321.

[358] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali G. Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. Delegated Persist Ordering. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, pages 58:1–58:13. IEEE Computer Society, 2016. doi: 10.1109/MICRO.2016.7783761.

[359] Tobias Maier, Peter Sanders, and Roman Dementiev. Concurrent Hash Tables: Fast *and* General?(!). In Rafael Asenjo and Tim Harris, editors, *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, pages 34:1–34:2. ACM, 2016. doi: 10.1145/2851141.2851188.

[360] Marius Muench, Fabio Pagani, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna, and Davide Balzarotti. Taming Transactions: Towards Hardware-assisted Control Flow Integrity Using Transactional Memory. In Fabian Monrose, Marc Dacier, Gregory Blanc, and Joaquín García-Alfaro, editors, *Research in Attacks, Intrusions, and Defenses - 19th International Symposium, RAID 2016, Paris, France, September 19-21, 2016, Proceedings*, volume 9854 of *Lecture Notes in Computer Science*, pages 24–48. Springer, 2016. doi: 10.1007/978-3-319-45719-2_2.

[361] Nikos Nikoleris, Andreas Sandberg, Erik Hagersten, and Trevor E. Carlson. Coolsim: Eliminating Traditional Cache Warming with Fast, Virtualized Profiling. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2016, Uppsala, Sweden, April 17-19, 2016*, pages 149–150. IEEE Computer Society, 2016. doi: 10.1109/ISPASS.2016.7482085.

[362] Mohamed M. Saad, Roberto Palmieri, Ahmed Hassan, and Binoy Ravindran. Extending TM Primitives Using Low Level Semantics. In Christian Scheideler and Seth Gilbert, editors, *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*, pages 109–120. ACM, 2016. doi: 10.1145/2935764.2935794.

[363] Dimitris Siakavaras, Konstantinos Nikas, Georgios I. Goumas, and Nectarios Koziris. Massively Concurrent Red-black Trees with Hardware Transactional Memory. In *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016, Herak-*

lion, Crete, Greece, February 17-19, 2016, pages 127–134. IEEE Computer Society, 2016. doi: 10.1109/PDP.2016.65.

[364] Matthew J. Walker, Stephan Diestelhorst, Andreas Hansson, Domenico Balsamo, Geoff V. Merrett, and Bashir M. Al-Hashimi. Thermally-aware Composite Run-time CPU Power Models. In *26th International Workshop on Power and Timing Modeling, Optimization and Simulation, PATMOS 2016, Bremen, Germany, September 21-23, 2016*, pages 17–24. IEEE, 2016. doi: 10.1109/PATMOS.2016.7833420.

[365] *6th Generation Intel (r) Processor Family Specification Update*. Intel, document number: 332689-011 en edition, June 2017. URL `https://www.intel.com/content/www/us/en/processors/core/desktop-6th-gen-core-family-spec-update.html`.

[366] *Desktop 4th Generation Intel(r) Core(tm) Processor Family, Desktop Intel (r) Pentium (r) Processor Family, and Desktop Intel (r) Celeron (r) Processor Family Specification Update*. Intel, revision 037us edition, March 2017. URL `https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/4th-gen-core-family-desktop-specification-update.pdf`.

[367] *Intel(r) 64 and Ia-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*. Intel, October 2017. URL `https://software.intel.com/en-us/articles/intel-sdm`.

[368] Mohammad Alian, Umur Darbaz, Gábor Dózsa, Stephan Diestelhorst, Daehoon Kim, and Nam Sung Kim. Dist-gem5: Distributed Simulation of Computer Clusters. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2017, Santa Rosa, CA, USA, April 24-25, 2017*, pages 153–162. IEEE, 2017. doi: 10.1109/ISPASS.2017.7975287.

[369] Arm. Modelling Arm-based Socs and Subsystems with Fast Models, 2017. URL `https://developer.arm.com/products/system-design/fast-models`.

[370] Nathan Chong, Tyler Sorensen, and John Wickerson. The Semantics of Transactions and Weak Memory in X86, Power, Armv8, and C++. *CoRR*, abs/1710.04839, 2017. URL `http://arxiv.org/abs/1710.04839`.

[371] Andrés Amaya García, René de Jong, William Wang, and Stephan Diestelhorst. Composing Lifetime Enhancing Techniques for Non-volatile Main Memories. In *Proceedings of the International Symposium on Memory Systems, MEMSYS 2017, Alexandria, VA, USA, October 02 - 05, 2017*, pages 363–373. ACM, 2017. doi: 10.1145/3132402.3132411.

[372] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. Nvthreads: Practical Persistence for Multi-threaded Applications. In Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic, editors, *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 468–482. ACM, 2017. doi: 10.1145/3064176.3064204.

[373] Aasheesh Kolli, Vaibhav Gogte, Ali G. Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Language-level Persistency. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*, pages 481–493. ACM, 2017. doi: 10.1145/3079856.3080229.

[374] Maged Michael. Personal Communication. 2017.

[375] Basireddy Karunakar Reddy, Matthew J. Walker, Domenico Balsamo, Stephan Diestelhorst, Bashir M. Al-Hashimi, and Geoff V. Merrett. Empirical CPU Power Modelling and Estimation in the Gem5 Simulator. In *27th International Symposium on Power and Timing Modeling, Optimization and Simulation, PATMOS 2017, Thessaloniki, Greece, September 25-27, 2017*, pages 1–8. IEEE, 2017. doi: 10.1109/PATMOS.2017.8106988.

[376] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanaël Prémillieu, Alastair Reid, Alejandro Rico, and Paul Walker. The ARM Scalable Vector Extension. *IEEE Micro*, 37(2):26–39, 2017. doi: 10.1109/MM.2017.35.

[377] Alejandro Villegas, Rafael Asenjo, Angeles G. Navarro, Oscar G. Plata, Rafael Ubal, and David R. Kaeli. Hardware Support for Scratchpad Memory Transactions on GPU Architectures. In Francisco F. Rivera, Tomás F. Pena, and José Carlos Cabaleiro, editors, *Euro-Par 2017: Parallel Processing - 23rd International Conference on Parallel and Distributed Computing, Santiago de Compostela, Spain, August 28 - September 1, 2017, Proceedings*, volume 10417 of *Lecture Notes in Computer Science*, pages 273–286. Springer, 2017. doi: 10.1007/978-3-319-64203-1_20.

[378] Ilias Vougioukas, Andreas Sandberg, Stephan Diestelhorst, Bashir M Al-Hashimi, and Geoff V Merrett. Nucleus: Finding the Sharing Limit of Heterogeneous Cores. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):152, 2017.

[379] Matthew J. Walker, Stephan Diestelhorst, Andreas Hansson, Anup Das, Sheng Yang, Bashir M. Al-Hashimi, and Geoff V. Merrett. Accurate and Stable Run-time Power Modeling for Mobile and Embedded Cpus. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 36(1):106–119, 2017. doi: 10.1109/TCAD.2016.2562920.

[380] Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Elastic Transactions. *J. Parallel Distrib. Comput.*, 100:103–127, 2017. doi: 10.1016/j.jpdc.2016.10.010.

[381] Karl Rupp. 42 Years of Microprocessor Trend Data. February 2018. URL `https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/`.

[382] Karthik Sangaiah, Michael Lui, Radhika Jagtap, Stephan Diestelhorst, Siddharth Nilakantan, Ankit More, Baris Taskin, and Mark Hempstead. Synchrotrace: Synchronization-aware Architecture-agnostic Traces for Lightweight Multicore Simulation of CMP and HPC Workloads. *TACO*, 15(1): 2:1–2:26, 2018. doi: 10.1145/3158642.

[383] Eric Koskinen and Maurice Herlihy. Concurrent Non-commutative Boosted Transactions. PODC 09 submission.

[384] Advanced Micro Devices, Inc. AMD Advanced Synchronization Facility Proposal. Available as developer.amd.com/CPU/ASF/Pages/default.aspx.

Appendix A

ASF 2.0 Specification

Preliminary Information — Subject to Change

Advanced Synchronization Facility

Proposed Architectural Specification

Publication #	**45432**	Revision:	**2.1**
Issue Date:	**March 2009**		

Advanced Micro Devices

Contents

Revision History

Date	Revision	Description
March 2009	2.1	Minor typographic and language corrections
August 2008	2.0	Initial public release

Chapter 1 Introduction

The Advanced Synchronization Facility (ASF) is an AMD64 extension to allow user- and system-level code to modify a set of memory objects atomically without requiring expensive traditional synchronization mechanisms.

The ASF extension provides an inexpensive primitive from which higher-level synchronization mechanisms can be synthesized: for example, multi-word compare-and-exchange, load-locked-store-conditional, lock-free data structures, lock-based data structures that do not suffer from priority inversion, and primitives for software-transactional memory.

ASF is both more flexible and less expensive than existing atomic memory modification primitives. Instead of offering new instructions with hardwired semantics (such as compare-and-exchange for two independent memory locations), ASF only exposes a mechanism for atomically updating multiple independent memory locations and allows software to implement the intended synchronization semantics.

1.1 Overview

ASF works by allowing software to declare speculative regions that specify and modify a set of protected memory locations. Modifications made to protected memory become visible to other CPUs either all at once (when the speculative region finishes successfully) or never (if the speculative region is aborted).

Unlike traditional critical sections, ASF speculative regions do not require mutual exclusion. Multiple ASF speculative regions that may access the same memory locations can be active at the same time on different processors, allowing greater parallelism. When ASF detects conflicting accesses to protected memory, it aborts the speculative region and notifies software, which can retry the operation as desired.

ASF protects memory at cache-line granularity. Despite cache-line size being an implementation detail, software does not have to be concerned with cache lines and can instead work on the level of memory objects, as long as all of the following constraints are met, which are supported by all ASF-capable CPU implementations:

- ASF-protected memory objects have a size of up to 64 bytes and are naturally aligned. (All ASF-capable implementations have a cache-line size of at least 64 bytes.)
- The speculative region does not reference more than four objects (the architecturally guaranteed minimum; more may be supported on a model-specific basis).
- Memory objects protected using ASF do not share cache lines with memory objects that should not be so protected. (False sharing may lead to unwanted protection, exceptions, and unnecessary aborts.)

1.1.1 ASF guarantees

In more detail, ASF guarantees forward progress for speculative regions, provided the following conditions hold:

- The speculative region does not exceed ASF's guaranteed capacity: up to four cacheable memory regions with a size and alignment of 64 bytes. (See Section 1.6 for details.)
- No interrupt or exception is delivered while executing the speculative region.
- There are no conflicting memory accesses from other CPUs.

If one of these conditions does not hold, the speculative region will be aborted (explained in more detail in Section 1.4).

1.1.2 ASF limitations

ASF has the following limitations:

- ASF supports only a limited form of nested speculative regions. (Refer to Section 1.5 for details.)
- ASF only operates on cacheable data and has a weakened memory-access-ordering model in certain respects. Memory ordering can be controlled as necessary via existing fence instructions. (Refer to Section 6.5 for details.)

1.2 Speculative region structure

ASF introduces a set of new instructions for denoting the beginning and end of a speculative region and for protecting memory objects. Additionally, ASF speculative regions first need to specify which memory objects should be protected using special declarator instructions.

Once a set of memory objects is protected, a speculative region can modify these memory objects speculatively. If a speculative region completes successfully, all such modifications become visible to all CPUs simultaneously and *atomically*. Otherwise, the modifications are discarded.

An ASF speculative region has the following structure:

1. The speculative region is entered with the SPECULATE instruction.
2. SPECULATE always writes an ASF status code of zero in rAX and sets the rFLAGS register accordingly. This status code distinguishes between the initial entry into a speculative region and an abort situation. SPECULATE also remembers the address of the instruction following the SPECULATE instruction as the landmark to which control is transferred on an abort.
3. SPECULATE is followed by instructions that check the status code and jump to an error handler if it is not zero (typically JNZ).
4. Declarator instructions (memory-load forms of LOCK MOVx, LOCK PREFETCH, and LOCK PREFETCHW instructions) are used to specify locations for atomic access – memory that ASF should protect. The MOV forms also perform the specified register load.
5. The speculative region (standard x86 instructions) is executed.

6. Once a memory location has been protected using a declarator, it can be read using regular x86 instructions. However, to modify protected memory locations, the speculative region uses memory-store forms of LOCK MOVx instructions. (It is an error to use regular memory-updating instructions for protected memory locations. Doing so results in a #GP exception.)
7. The COMMIT instruction denotes the end of the speculative region and causes the modifications to the protected lines to become visible to the rest of the system.
8. An ABORT instruction is available to programmatically terminate the speculative region with abort rather than commit semantics.

Note that the two declarators LOCK PREFETCH and LOCK PREFETCHW differ from non-LOCK-prefixed prefetches in that they need to check the specified memory address for translation faults and memory-access permission and generate a page fault if unsuccessful. This behavior is necessary because ASF needs to establish a valid translation before it starts monitoring the protected memory location.

Example

The following example code implements compare-and-exchange on two independent memory locations using ASF (dubbed “DCAS” for “double compare-and-swap”). (This code uses immediate retry as the recovery strategy. A real implementation might have a more elaborate recovery strategy, for example, exponential backoff.)

```
; DCAS Operation:
; IF ((mem1 = RAX) && (mem2 = RBX))
; {
;   mem1 = RDI
;   mem2 = RSI
;   RCX = 0
; }
; ELSE
; {
;   RAX = mem1
;   RBX = mem2
;   RCX = 1
; }
; (R8, R9 modified)
;
DCAS:
    MOV      R8, RAX
    MOV      R9, RBX
retry:
    SPECULATE                   ; Speculative region begins
    JNZ      retry              ; Page fault, interrupt, or contention
    MOV      RCX, 1             ; Default result, overwritten on success
    LOCK MOV RAX, [mem1]        ; Specification begins
    LOCK MOV RBX, [mem2]
    CMP      R8, RAX            ; DCAS semantics
    JNZ      out
    CMP      R9, RBX
    JNZ      out
    LOCK MOV [mem1], RDI         ; Update protected memory
    LOCK MOV [mem2], RSI
    XOR      RCX, RCX            ; Success indication
out:
    COMMIT                       ; End of speculative region
```

1.3 Unprotected memory

ASF only protects memory lines that have been specified using declarator instructions. All other memory remains unprotected and can be modified inside a speculative region using standard x86 instructions. These modifications retain their standard behavior, that is, they become visible to other CPUs immediately and in program order.

1.4 Speculative region aborts

Speculative regions can be aborted at any point because of contention, far control transfers (including those caused by interrupts and faults), or software aborts.

Speculative-region aborts discard modifications to the contents of the protected lines, causing them to be unobservable by other CPUs. However, ASF does not roll back modifications to unprotected memory. Software must be written to accommodate these modifications. In many cases this will simply be a matter of reentering the initialization sequence leading up to the speculative region.

Aborts do not roll back register state (except for the instruction and stack pointers, as described later in this section). Software must be written to handle or ignore modified register contents in case of an abort, or it must avoid modifying them in the speculative regions.

Before an interrupt or exception handler returns, operating-system code or other processes may have executed in the interim. This is of no consequence for the interrupted software as no ASF-related state is maintained across context switches. Other processes may even have executed ASF speculative regions that inspected or modified any of the locations targeted by the interrupted speculative region. The interrupted software will have its speculative region aborted and simply needs to re-inspect the state of the shared data structure as it attempts its speculative region again.

ASF is unusual in that SPECULATE has rollback semantics (much like C's setjmp interface): Speculative-region aborts reset the instruction and stack pointers to the values they had after SPECULATE was first executed. The rAX register is also written with a nonzero status code that provides details of the abort condition, and rFLAGS is set accordingly. The subsequent instructions can inspect the status code or rFLAGS register and direct the control flow (via a conditional jump) to the error handler.

1.5 Nested speculative regions

ASF supports composing a speculative region out of pseudo-nested speculative regions by flattening a hierarchy of SPECULATE-COMMIT pairs into just one speculative region. All pseudo-nested speculative regions share ASF's resources; nested COMMIT instructions do not release any protected lines. For nesting to work, all memory that is protected in the outermost SPECULATE-COMMIT pair plus all nested SPECULATE-COMMIT pairs must fit within ASF's limits for protected lines.

Because of the flattening, memory locations protected in a nested speculative region remain protected in outer speculative regions. Therefore, outer speculative regions need to use LOCK MOV for updating memory locations protected by an inner speculative region. (Use of regular memory-update instructions for protected lines results in a #GP exception.)

To detect nesting, ASF maintains an internal nesting count that is incremented by SPECULATE and decremented by COMMIT. A nested SPECULATE does not define a new checkpoint for rollback. Instead, aborts always roll back to the first SPECULATE that started the speculative region.

1.6 Capacity

A given ASF implementation will have certain capacity constraints caused by hardware limitations, such as the number of locations that can be simultaneously monitored for contention, or the number of stores that can be handled speculatively. There are two aspects to this: a minimum guaranteed capacity, and a larger reference-pattern dependent capacity.

A speculative region is guaranteed to complete, in the absence of disturbances such as faults, interrupts, or contention, as long as the number of protected locations does not exceed the minimum guaranteed capacity, regardless of where in the cacheable address space those locations are. An implementation may also provide a capacity beyond the minimum that can vary depending on which locations are referenced.

For example, an implementation may require that all protected locations simultaneously reside in the data cache for the duration of the speculative region, and if a protected line is displaced from the cache because of replacement, the speculative region is aborted. Hence, a speculative region that happened to reference N+1 locations that all mapped to the same index in an N-way associative data cache would never be able to complete. In this case, the minimum guaranteed capacity would be determined by the cache’s associativity.

For more random reference patterns, a speculative region could however reference many locations before the associativity at any one cache index is exceeded and a protected line is displaced, and hence could often operate successfully on a much larger data set than the guaranteed minimum. However, there would be no guarantee for any given set of references that it would not hit a hardware limitation. In such a scenario, software must provide an alternate means for completing the intended operation in case the ASF hardware cannot handle it – for example employing a global lock (see Section -Lock for a specific example). ASF provides an indication to software of when such a limitation has been hit, distinguishing it from transient conditions which might not be encountered on a retry. (Refer to Section 6.1 for details.)

Implementations may use monitoring and store buffering mechanisms which are not tied to cache associativity. In any event, all ASF implementations architecturally guarantee a minimum capacity of four cache lines. The actual minimum of a given implementation (which may be higher) is reported by CPUID.

For some use cases, the odds that a speculative region with a larger number of reads will succeed can be increased through the use of RELEASE instructions, which remove designated cache lines from the monitored set, lowering the chances of hitting a hardware capacity limit. This could be applicable in such cases as walking a long linked list, where each successive element can be dropped once it has been traversed without being modified.

Chapter 2 Terminology

(Terms set in *italics* are defined in a separate glossary entry.)

ABORT

Instruction that voluntarily aborts a speculative region. See also *Abort* and Section 5.5.

Abort

A condition that causes a *speculative region* to fail. Different abort conditions are distinguished by an abort status code written to rAX when the abort is signaled. In case of an abort, the contents of protected lines and the instruction and stack pointers are rolled back to the values they had when SPECULATE was executed. Aborts in nested speculative regions roll back to the SPECULATE instruction that started the outermost speculative region.

ASF configuration MSR

A model-specific register (MSR) configuring ASF.

Cache line

Aside from their use to reduce memory-access latencies, ASF uses the cache-coherency protocol for detecting *contention*. Therefore, the granularity for ASF memory protection is the size of a cache line.

See also *memory line*.

COMMIT

Instruction that denotes the end of a *speculative region*. See Section 5.4.

Contention

Conflicting memory accesses that usually cause a speculative region to abort. See Section 6.2.

CPU

In this specification, the term “CPU” refers to one logical CPU (one hardware thread executing x86 instructions), irrespective of how these logical CPUs are packaged. (Its use is synonymous to terms like “CPU core” and “x86 thread,” which are not used in this specification.)

Declarator

Instruction that declares a location for atomic access (*protected lines*) during a *speculative region*: LOCK MOVx (load), LOCK PREFETCH, and LOCK PREFETCHW. See Section 5.2.

Far control transfer

A (voluntary or involuntary) control-flow diversion to another privilege level or another code segment. Far control transfers include far-call, far-jump, far-ret, and interrupts. See Section 6.4.

Imprecise exception

Exceptions occurring in a *speculative region* cause an *abort*, rolling back execution flow to the instruction following *SPECULATE* before the exception handler is called. Consequently, the instruction and stack pointers reported to the exception handler do not correspond to the fault site, making the exception imprecise. See Section 6.4.1.1.

Memory line

A region of physical memory that has the same size and alignment as a *cache line*.

Protected line

A *memory line* that is protected during a *speculative region*. ASF maintains atomicity of updates to all protected lines as long as no other CPU contends for it (see *contention*). Otherwise, the speculative region is *aborted*.

RELEASE

Instruction that allows ASF to release one *protected line* before the end of a *speculative region*. Protected lines that have been modified cannot be released.

SPECULATE

Instruction that starts an ASF *speculative region*. In case the speculative region is aborted, the instruction and stack pointer are rolled back to the post-SPECULATE instruction values, and modifications to *protected lines* are discarded.

Speculative region

An ASF speculative region starts with the execution of the *SPECULATE* instruction and ends either when the *COMMIT* instruction is executed or when the speculative region is *aborted*.

Transactional store

Instructions that write to *protected lines*, in particular, memory-store variants of LOCK MOVx. No other instructions are allowed to write to protected lines. See Section 5.3.

Chapter 3 CPUID identification

To determine whether ASF is present and which capabilities it has, use the CPUID instruction.

3.1 Detecting ASF presence and capabilities

CPUID <= EAX = ***TBD***

Return: ASF capabilities, according to the following table:

Register	Bits	Meaning
EDX	31:1	Reserved.
EDX	0	**ASF**: Set to 1 if the CPU supports ASF.
EBX	31:16	Reserved.
EBX	15:0	**ASFCapacity**: ASF capacity. The minimum number of different protected lines in an ASF speculative region that this implementation supports. If ASF is present, this value is always greater than or equal to 4. Note that an ASF implementation might support more than the number of protected lines reported by ASFCapacity under certain conditions; see Section 1.6.

3.2 Detecting the cache-line size

CPUID <= EAX = 0000_0001h

Return: Various information (refer to the CPUID specification for details). Includes:

Register	Bits	Meaning
EBX	15:8	**CLFlush**: Cache-line size. Specifies the size of a cache line in quadwords. (A quadword has a size of eight bytes.) AMD64 implementations supporting ASF always have a cache-line size of at least 8 quadwords (64 bytes).

Chapter 4 Model-specific registers

4.1 ASF configuration MSR

MSR ***TBD*** – ASF_CFG

ASF configuration MSR

This MSR defines ASF's current operating mode.

Bits	Meaning
63:1	Reserved – MBZ
0	**ASFFault**: Fault when ASF capacity is exceeded. When this bit it set to 1, declarators generate a #GP(0) fault when software attempts to protect more lines than supported by ASF. Otherwise, the #GP is suppressed. Instead, the speculative region is aborted and the abort status code is set to *ASF_CAPACITY*.

This MSR is read–write. Its reset value is 0.

4.2 ASF exception IP MSR

MSR ***TBD*** – ASF_EXCEPTION_IP

ASF exception IP MSR

In the case of an exception in a speculative region that causes an abort, ASF saves the rIP of the original fault or trap site in this MSR before aborting the speculative region. (This rIP value is the one that would have been put in the exception frame if the rollback had not happened.) The rIP actually reported in the exception frame is the address of the instruction following the initial SPECULATE instruction due to the rollback.

A bit on the exception handler's stack frame indicates whether the ASF_EXCEPTION_IP MSR contains a valid value. Refer to Section 6.4.1.1 for details.

Bits	Meaning
63:0	**ExceptionIP**: rIP at which an exception or fault occurred before the speculative region was aborted.

This MSR is read-only.

4.3 Debug-control MSR

MSR 0000_01D9 – DebugCtlMSR

The following bit is added to this MSR:

Bits	Meaning
TBD	**DebugAbort**: If set to 1, #DB debug traps abort speculative regions. Otherwise, ASF speculative regions act as an interrupt shadow for debug traps: #DB traps in ASF speculative regions are deferred until after the speculative region has ended. Read-write. Defaults to 0.

Chapter 5 Instructions

This section describes the instructions added to the AMD64 architecture to support ASF. All of these instructions raise #UD if ASF is not implemented or if bit 0 of the ASF_CFG MSR is 0.

5.1 SPECULATE

5.1.1 Instruction

Mnemonic

```
SPECULATE
```

Opcode

TBD

5.1.2 Description

SPECULATE starts an ASF speculative region.

The exact operation of SPECULATE differs depending on whether it is the initial SPECULATE of a top-level speculative region or a nested SPECULATE.

The initial instance of SPECULATE records the (partial) checkpoint to which execution returns if the speculative region is aborted. The checkpoint consists of the values the instruction and stack pointers will have after SPECULATE has completed execution (hence on an abort, control transfers to whatever instruction follows SPECULATE). SPECULATE also clears the rAX register, sets rFLAGS accordingly, and sets the nesting level (an internal processor state variable) to 1.

If an instance of SPECULATE is encountered within an ASF speculative region, it does not checkpoint the instruction and stack pointers but it does clear rAX and set rFLAGS. It also increments the nesting level. An ASF abort will transfer control to the checkpoint recorded by the initial instance of SPECULATE.

The maximum nesting level is 256. If this level is exceeded, SPECULATE raises #GP(0).

5.1.3 Operation

```
IF (NEST_LEVEL = 256)
{
  EXCEPTION [#GP(0)]
}
rAX = 0
NEST_LEVEL += 1
IF (NEST_LEVEL = 1)
{
  SAVED_rSP = rSP
  SAVED_rIP = rIP of next instruction
}
```

5.1.4 Flags affected

SF, ZF, AF, PF, CF set according to result in rAX. OF is set to 0.

5.2 LOCK MOVx (load), PREFETCH, and PREFETCHW

5.2.1 Instruction

Mnemonic

```
LOCK MOV reg,mem
```

Opcodes

F0 8A/r, F0 8B/r, F0 A0, F0 A1

Mnemonic

```
LOCK MOV{D,DQA,DQU,Q} xmm,mem
```

Opcodes

F0 66 0F 6E/r, F0 66 0F 6F/r, F0 F3 0F 6F/r, F0 F3 0F 7E/r

Mnemonic

```
LOCK PREFETCH mem
```

Opcode

F0 0F 0D/0

Mnemonic

```
LOCK PREFETCHW mem
```

Opcode

F0 0F 0D/1

5.2.2 Description

These memory-reference instructions, called *declarators*, are used to specify locations for which atomic access is desired.

Declarators work like their counterparts without the LOCK prefix, with the following additional operation:

Each declarator adds the memory line containing the first byte of the referenced memory object to the set of protected lines. Software must ensure that unaligned memory accesses do not span both protected and unprotected lines; otherwise, the atomicity of data accesses to these memory objects is not guaranteed.

Unlike prefetches without a LOCK prefix, LOCK PREFETCH and LOCK PREFETCHW also check the specified memory address for translation faults and memory-access permission (read or write, respectively) and, if unsuccessful, generate a page-fault or general-protection exception as appropriate. Also, LOCK PREFETCH and LOCK PREFETCHW generate a #DB exception when they reference a memory address for which a data breakpoint has been configured.

A declarator referencing a line that has already been protected is permitted and behaves like a regular memory reference. It does not change the protected status of the line.

Once a memory line has been protected using a declarator, it can be modified speculatively (but cannot be modified nonspeculatively) within the speculative region. See Section 5.3 for instructions that can update protected lines, and Section 6.5 for memory access ordering rules.

If the number of declarators issued in the current speculative region exceeds ASF's maximum supported capacity, the behavior depends on the setting of MSR ASF_CFG[ASFFault]. If that bit is set to 1, a #GP(0) is generated. Otherwise, the #GP is suppressed. Instead, the speculative region is aborted and the abort status code is set to *ASF_CAPACITY*.

Declarators are not allowed outside of speculative regions and result in #UD in this case.

LOCK MOVx from memory with a caching type other than WB (writeback) is not supported by ASF and results in #GP(0).

LOCK MOVD into MMX registers is not supported and results in #UD.

(#GP and #UD, like all interrupts, also abort the speculative region. See Section 6.4.)

5.2.3 Operation

```
IF (CPU not in speculative region)
{
  EXCEPTION [#UD]
}
IF (instruction = LOCK PREFETCHW)
{
  translate memory address and check for write permission
    // Generates #GP or #PF if necessary
}
ELSE
{
  translate memory address and check for read permission
    // Generates #GP or #PF if necessary
}
IF (line already protected)
{
  perform conventional memory reference operation
  EXIT
}
IF (address refers to non-WB memory type)
{
  EXCEPTION [#GP(0)]
}
IF (ASF capacity overflow)
{
  IF (ASF_CFG[ASFFault])
  {
    EXCEPTION [#GP(0)]
  }
  ELSE
  {
    abort speculative region (ASF_CAPACITY, 1, 0) // See Section 6.1
    EXIT
  }
}
execute memory reference and handle contention // See Section 6.2
add line to set of protected lines
```

5.2.4 Flags affected

None.

5.3 LOCK MOVx (store)

5.3.1 Instruction

Mnemonic

```
LOCK MOV mem,reg/imm
```

Opcodes

F0 88/r, F0 89/r, F0 A2, F0 A3, F0 C6/0i, F0 C7/0i

Mnemonic

```
LOCK MOV{D,DQA,DQU,Q} mem,xmm
```

Opcodes

F0 66 0F 7E/r, F0 66 0F 7F/r, F0 F3 0F 7F/r, F0 66 0F D6/r

5.3.2 Description

These memory-store instructions are used to store data into protected lines. The lines must already have been protected by a declarator instruction (see Section 5.2); if not, these store instructions result in #GP(0).

Updates to protected lines do not become visible to other CPUs until the COMMIT instruction is executed. If the speculative region is aborted, these updates will be discarded and cannot be observed from other CPUs.

There are no other instructions to store data into protected lines. Attempting to modify protected lines using regular move instructions or other memory-updating instructions results in #GP(0).

LOCK MOVx store instructions are not allowed outside of speculative regions and result in #UD in this case.

Software must ensure that unaligned memory accesses resulting from LOCK MOVx store instructions do not span both protected and unprotected lines; otherwise, #GP(0) is generated.

LOCK MOVD from MMX registers is not supported and results in #UD.

(#GP and #UD, like all interrupts, also abort the speculative region. See Section 6.4.)

5.3.3 Operation

```
IF (CPU not in speculative region)
```

```
{
  EXCEPTION [#UD]
}
translate memory address and check for write permission
  // Generates #PF if necessary
IF (access spanning protected and unprotected line
    || ! line already protected)
{
  EXCEPTION [#GP(0)]
}
execute memory reference and handle contention // See Section 6.2
```

5.3.4 Flags affected

None.

5.4 COMMIT

5.4.1 Instruction

Mnemonic

```
COMMIT
```

Opcode

TBD

5.4.2 Description

Denotes end of an ASF speculative region.

When the COMMIT belongs to a pseudo-nested speculative region (a nested SPECULATE-COMMIT pair), COMMIT decrements the nesting count and exits without releasing any protected lines.

When COMMIT ends a speculative region (nest count is equal to 1), this instruction releases all protected lines. Modified protected lines will be committed and made visible to other CPUs.

COMMIT sets the rAX register to zero and sets rFLAGS according to the value in rAX. (Future enhancements to ASF may result in COMMIT setting rAX to a value other than zero.)

When encountered outside of a speculative region, the COMMIT instruction raises #GP(0).

5.4.3 Operation

```
IF (CPU not in speculative region)  // NEST_LEVEL = 0
{
  EXCEPTION [#GP(0)]
}
rAX = 0
NEST_LEVEL -= 1
IF (NEST_LEVEL = 0)
{
  commit protected lines
  release protected lines
  end speculative region
}
```

5.4.4 Flags affected

SF, ZF, AF, PF, CF set according to result in rAX. OF is set to 0.

5.5 ABORT

5.5.1 Instruction

Mnemonic

`ABORT`

Opcode

TBD

5.5.2 Description

Aborts an ASF speculative region.

In a speculative region, ABORT discards modifications to previously protected lines and releases all protected lines. The contents of the AX register are copied into the SoftwareAbort field of the abort code in rAX. The abort status-code field will be set to *ASF_ABORT*.

Refer to Section 6.1 for a further description of abort behavior.

When encountered outside an ASF speculative region, the ABORT instruction generates #GP(0).

5.5.3 Operation

```
IF (CPU not in speculative region)
{
  EXCEPTION [#GP(0)]
}
abort speculative region (ASF_ABORT, 0, AX) // See Section 6.1
```

5.5.4 Flags affected

None.

5.6 RELEASE

5.6.1 Instruction

Mnemonic

```
RELEASE mem
```

Opcode

TBD

5.6.2 Description

The RELEASE instruction is a hint that allows ASF to remove an unmodified protected line (referenced by the specified memory address) from a speculative region's set of protected lines.

RELEASE can be used to circumvent ASF's capacity limitations when traversing potentially long chains of pointers. However, as the instruction does not guarantee that the specified protected line will actually be released, software must be designed to fall back to a different code path when the capacity limit is reached.

RELEASE never releases protected lines that have been modified within the speculative region. The circumstances under which RELEASE releases unmodified protected lines are implementation specific.

If RELEASE does release a protected line, then another CPU accessing data contained in that memory line will no longer cause ASF contention. Otherwise, ASF continues to monitor the protected line for contention.

RELEASE does not consider the number of declarators that were used to protect the memory line. In other words, a protected line might be released even if it was specified using more than one declarator.

When attempting to release a line that is not in the current set of protected lines, the instruction is a no-op.

When encountered outside an ASF speculative region, the instruction generates #GP(0).

5.6.3 Operation

```
IF (CPU not in speculative region)
{
  EXCEPTION [#GP(0)]
}
IF (referenced line not in set of protected lines)
{
  EXIT
}
IF (referenced line has not been modified in speculative region)
{
  IF (implementation-specific conditions)
  {
    release referenced line
  }
}
```

5.6.4 Flags affected

None.

Chapter 6 Operation in ASF speculative regions

6.1 Aborts

6.1.1 Description

ASF automatically aborts a speculative region when one of the following conditions occurs:

- Contention for memory that is included in the set of protected lines (see Section 6.2)
- A condition that results in a far control transfer (see Section 6.4)
- Explicit abort (by executing ABORT)
- Other implementation-specific conditions

Aborts discard any modifications to currently locked cache lines and release all protected cache lines. Conditions that cause an abort also set the abort status code to a nonzero value. This code is passed to software in the rAX register, and rFLAGS is set accordingly, when the abort is signaled.

ASF signals aborts by rolling back rSP and rIP to the instruction following the SPECULATE instruction that initiated the speculative region. The conditional jump following SPECULATE can then jump to a recovery routine.

The other registers (general-purpose registers, floating-point registers, XMM registers) are not restored during a roll back. The only way software can rely on the contents of a register after a roll back is by not modifying it in the speculative region. Otherwise, software must be written to ignore, in the case of an abort, the contents of any registers the speculative region might have modified.

When an abort is signaled, the rAX register is always nonzero and has the following layout:

Bits	Meaning
63:32	(In 64-bit mode:) Set to zero
31:16	**SoftwareAbort**: 16-bit value passed to the ABORT instruction. Zero if no ABORT instruction was encountered.
15:8	**NestLevel**: Nesting level in which the abort occurred (equivalent to ASF's nesting count minus 1). Zero if the aborted speculative region has not been nested.
7	**HardError**: If set to 0, the speculative region has been aborted because of a transient error (such as contention) and can be retried. If set to 1, a hard error (such as a capacity overrun) has been detected, requiring a different recovery method.

6:0	**StatusCode**: The reason for the abort. See following table.

The StatusCode and HardError fields have the following meaning:

Status code	Hard error	Meaning
0	0	Success (no abort)
ASF_CONTENTION	0	Speculative region was aborted because of contention.
ASF_ABORT	0	Speculative region aborted using ABORT instruction.
ASF_FAR	0	Speculative region aborted by an exception or an interrupt.
ASF_DISALLOWED_OP	1	Speculative region aborted because of a disallowed instruction. (This status code indicates a programming error.)
ASF_CAPACITY	1	ASF capacity exceeded. The number of declarators exceeded the hardware's capacity for handling them atomically.
Other value	0	Spurious error
Other value	1	Hard error

"*ASF_CONTENTION*", "*ASF_ABORT*", and so on, are symbolic constants that will be defined in a later revision of this document.

Note that it is possible for interrupt handlers to modify the abort status code in rAX when they detect that an ASF speculative region has been aborted (through the Imprecise bit in the rFLAGS image on the stack or in the VMCB; see Section 6.4.1.1). For example, interrupt handlers can convey additional information in the SoftwareAbort field according to a software convention, and exception handlers can set the HardError flag if necessary.

6.1.2 Operation

```
abort speculative region (status_code, hard_error, software_code):
    STATUS_CODE = (status_code
                  | (hard_error << 7)
                  | ((NEST_LEVEL - 1) << 8)
                  | ((software_code & FFFFh) << 16))
    undo modifications to protected lines
    release protected lines
    NEST_LEVEL = 0
    rSP = SAVED_rSP
```

```
rIP = SAVED_rIP
rAX = STATUS_CODE
set EFLAGS.{SF,ZF,AF,PF,CF,OF} according to rAX
```

6.2 Contention

6.2.1 Description

Contention is interference that other CPUs cause when they access memory that has previously been protected. ASF aborts speculative regions under certain types of contention.

The following table summarizes how ASF handles contention in the case where CPU A performs an operation while CPU B is in a speculative region with the line protected by ASF:

CPU A mode	**CPU A operation**	**CPU B cache-line state**	
		Protected Shared	**Protected Owned ***
Speculative region	LOCK MOVx (load)	OK	B aborts
Speculative region	LOCK MOVx (store)	B aborts	B aborts
Speculative region	LOCK PREFETCH	OK	B aborts
Speculative region	LOCK PREFETCHW	B aborts	B aborts
Speculative region	COMMIT	OK	OK
Any	Read operation	OK	B aborts
Any	Write operation	B aborts	B aborts
Any	Prefetch operation	OK	B aborts
Any	PREFETCHW	B aborts	B aborts

"Owned *" – Modified or owned

6.2.2 Operation

Memory references:

```
     // "CPU A" refers to the current CPU executing the memory reference;
     // "CPU B" refers to another CPU
     IF (memory reference contends with CPU B's ASF speculative region)
     {
       CPU B -> abort speculative region (ASF_CONTENTION, 0, 0) // See
Section 6.1
     }
     execute memory reference
```

6.3 Disallowed instructions

Privileged instructions (those that must be executed at CPL = 0), instructions that cause a far control transfer or an exception, and all instructions that can be intercepted by an SVM hypervisor are not allowed in an ASF speculative region. This includes:

- FAR JMP, FAR CALL, FAR RET
- SYSCALL, SYSRET, SYSENTER, SYSEXIT
- INT, INT1, INT3, INTO, IRET, RSM
- BOUND, UD2
- PUSHF, POPF, PAUSE, HLT, CPUID, MONITOR, MWAIT, RDTSC, RDTSCP, RDPMC
- IN, OUT
- SIDT, SLDT, SGDT, STR, SMSW
- All privileged instructions
- All SVM instructions

Attempting to execute these instructions causes an #GP fault, which will be handled as a far control transfer (as described in the next section).

6.4 Far control transfers

6.4.1 Description

All far control transfers lead to an abort of the ASF speculative region. Far control transfers include traps, faults, exceptions, NMIs, SMIs, unmasked interrupts, and disallowed instructions converted into an exception (see previous section).

Instructions that directly or indirectly cause a far control transfer, described in Section 6.3, are not allowed inside ASF speculative regions and will generate a #GP exception.

After aborting the speculative region, discarding any modified protected lines, and rolling back rIP and rSP, ASF changes the abort status code to *ASF_FAR* and then executes the far control transfer. Upon return from the far control transfer (or the fault handler invoked by the #GP caused by a disallowed instruction), the conditional jump following SPECULATE can jump to a recovery routine.

6.4.1.1 Imprecise exception reporting

Exceptions, like all other far control transfers, cause ASF speculative regions to be aborted. Therefore, the rIP of the interrupted program that is pushed to the exception-handler stack does not correspond to the instruction that caused the fault or trap (unless a fault occurred at the first instruction after SPECULATE). For this reason, exceptions occurring while an ASF speculative region was active are called *imprecise exceptions.*

ASF saves the rIP of the original fault or trap site in the ASF_EXCEPTION_IP MSR.

To signal that an imprecise exception has occurred, ASF uses a flag (Imprecise) in the rFLAGS register image in the exception-handler stack frame or, in case the exception was intercepted, in the VMCB.RFLAGS state-save-area field. The Imprecise flag bit in the rFLAGS register cannot be set (it always reads as zero). Instructions that read rFLAGS from memory (such as IRET, POPF, and VMRUN) mask out the Imprecise bit when restoring the rFLAGS register. VMRUN uses VMCB.RFLAGS[Imprecise] only when injecting an exception or interrupt into a virtual machine: If the bit is set, the injected event will be marked as imprecise.

Specifically, the rFLAGS image on the exception-handler stack and VMCB.RFLAGS are extended as follows:

Bit	Meaning
31	**Imprecise**: When this bit is set, the exception has aborted an ASF speculative region, and the rIP pushed to the stack or saved to the VMCB may not correspond to the fault site. ASF sets this bit when rolling back an aborted speculative region; in that case, the rIP points to the instruction following the SPECULATE instruction. The ASF_EXCEPTION_IP MSR reports the original fault or trap site. This bit is not available to interrupt handlers invoked through a 16-bit interrupt or trap gate.

The ASF_EXCEPTION_IP MSR will be overwritten every time an imprecise exception occurs. To fully support ASF applications, operating systems should read this value as soon as possible and pass it on to user-level exception handlers.

6.4.1.2 Debug traps

When MSR DebugCtlMSR[DebugAbort] is cleared to 0, debug traps (#DB, caused by hardware breakpoints or single stepping) are deferred until the speculative region ends. Otherwise, they behave like the other far control transfers and abort the speculative region. (Please note that debug traps that abort a speculative region are signaled as imprecise exceptions – see previous subsection.)

In case of a debug trap, the debug-status register (DR6) reflects the conditions valid when the configured breakpoint was hit. If the #DB exception is deferred, DR6 reflects the condition immediately and is not protected from being overwritten before the exception is delivered. In addition, DR6 can accumulate additional breakpoint information throughout the rest of the speculative region.

6.4.1.3 Page faults

In case of page faults, CR2 (page-fault linear address) contains the actual page-fault address before the rollback.

6.4.2 Operation

```
    IF (far control transfer because of a disallowed instruction)
    {
      tmp_hard_error = 1
      tmp_status_code = ASF_DISALLOWED_OP
    }
    ELSE
    {
      tmp_hard_error = 0
      tmp_status_code = ASF_FAR
    }
    IF (CPU in speculative region)
    {
      tmp_rIP = rIP
      abort speculative region (tmp_status_code, tmp_hard_error, 0) // See
Section 6.1
      IF (far control transfer = exception)
      {
        MSR ASF_EXCEPTION_IP = tmp_RIP
        if (exception intercepted)
        {
          VMCB.RFLAGS |= 2^31
        }
        ELSE
        {
          rFLAGS stack-image value |= 2^31
        }
      }
    }
    execute far control transfer
```

6.5 Memory access ordering

ASF speculative regions have a different memory access ordering model in that modifications to protected lines cannot be observed from other CPUs until successful completion of a COMMIT instruction, at which point they become visible at once.

While writes to memory locations in unprotected lines become visible in program order, the total order of memory accesses in both protected and unprotected lines after COMMIT or RELEASE (observed from another CPU) is implementation specific. If a stronger ordering model is desired, software needs to insert LFENCE, SFENCE, and MFENCE instructions. For example, if all unprotected memory writes should become visible before all protected ones, software can use SFENCE immediately before COMMIT.

For all other memory modifications, the standard ordering rules apply. In particular, writes occurring before SPECULATE always become visible before all writes in the speculative region – both protected and unprotected ones (not considering incompatible caching types).

Example

Consider the following program:

```
    MOV      [mem1], 0
    SPECULATE
    JNZ      error
    LOCK MOV RAX, [mem3]
    MOV      [mem2], 0
    LOCK MOV [mem3], 0
    MOV      [mem4], 0
    COMMIT
    MOV      [mem5], 0
```

This program can expose any of the following memory write orders (assuming the speculative region is not aborted):

1. mem1, mem2, mem3, mem4, mem5
2. mem1, mem2, mem4, mem3, mem5

Inserting SFENCE just before COMMIT forces the order to be the second one.

Even though writes to protected memory are held pending and do not become visible to other CPUs before COMMIT, all writes (to protected or unprotected memory) appear to be in program order on the executing CPU.

Conventional LOCK-prefixed instructions (such as LOCK CMPXCHG; or XCHG, which has implicit LOCK semantics) have unchanged behavior, including fencing semantics. However, note that it is not possible to use conventional LOCK-prefixed instructions to manipulate ASF-protected memory (only LOCK MOV store instructions can be used to update protected memory).

6.6 Updating Accessed and Dirty bits in page-table entries

When executing an ASF speculative region, the CPU updates the Accessed and Dirty bits of the referenced page-table entries as it would if no speculative region were active. Speculative modifications to protected memory locations thus leads to a set Dirty bit even if the modifications are later discarded because of an abort.

The behavior caused by protecting memory lines (using declarator instructions) containing active page tables (memory lines accessed and updated by the CPU's page-table walker) is undefined.

Chapter 7 ASF usage models

ASF provides considerable flexibility in the construction of synchronization methods. Some basic usage examples are provided here for illustration.

7.1 Lock-free synchronization primitives

7.1.1 Double-word compare and swap

Double compare-and-swap (DCAS) is a primitive that allows atomic manipulation of pointer-based data structures such as doubly linked lists, queues, and trees.

Unlike the example in Section 1.2, this version of DCAS does not implicitly retry in case of contention or aborts, but leaves the retry (and the backoff) to application code. (Also, in that case it does not return the current memory values.)

```
; DCAS Operation:
; IF ((mem1 = RAX) && (mem2 = RBX))
; {
;   mem1 = RDI
;   mem2 = RSI
;   RCX = 0
; }
; ELSE
; {
;   RCX = 1
; }
; (RAX, RBX, R8, R9 modified)
;
DCAS:
    MOV      R8, RAX
    MOV      R9, RBX
    MOV      RCX, 1
    SPECULATE                    ; speculative region begins
    JNZ      fail                ; Bail out if rolled back
    LOCK MOV RAX, [mem1]         ; Specification begins
    LOCK MOV RBX, [mem2]
    CMP      R8, RAX             ; DCAS semantics
    JNZ      out
    CMP      R9, RBX
    JNZ      out
    LOCK MOV [mem1], RDI         ; Update protected memory
    LOCK MOV [mem2], RSI
    XOR      RCX, RCX
```

```
out:
    COMMIT                      ; End of speculative region
fail:
```

7.1.2 Load locked, store conditional

Load locked (LL) and store conditional (SC) are a pair of primitives that allow a store to occur only if a previously loaded memory operand has not been changed.

Typical LL and SC instructions cannot directly be translated to ASF because ASF always rolls back program flow to a point before the first memory reference. However, programs using LL and SC can be expressed using ASF as follows:

```
    SPECULATE                   ; LL/SC section begins
    JNZ      ll_sc_failed
    LOCK MOV RAX, [mem]
    ...
 ; compute new value for mem in RAX
    ...
    LOCK MOV [mem], RAX
    COMMIT                      ; End of speculative region
    ...
; Error handling
ll_sc_failed:
    ...
```

In addition to traditional LL/SC semantics, ASF also supports pipelined LL/SC sequences:

```
    SPECULATE                   ; LL/SC section begins
    JNZ      ll_sc_failed
    LOCK MOV RAX, [mem1]
    LOCK MOV RBX, [mem2]
    LOCK MOV RCX, [mem3]
    ...
    LOCK MOV [mem1], RAX
    LOCK MOV [mem2], RBX
    LOCK MOV [mem3], RCX
    COMMIT                      ; End of speculative region
```

7.2 Lock-free data structures

ASF can be used to construct large speculative regions for manipulating lock-free data structures for which simple primitives such as DCAS are not sufficient or not convenient.

7.2.1 LIFO list manipulation

Lock-free LIFO lists are common linked-list structures where elements can be added or removed at the front of the list by manipulating a list header structure with a single compare-and-exchange instruction, such as CMPXCHG8B. One typical use of such structures is for maintaining a pool of buffers, where a buffer can be popped off the list as needed, and pushed back on when done with. The compare-and-exchange manipulation of the header structure allows the list to be manipulated simultaneously by competing threads without needing a global lock to provide mutually-exclusive access.

This benefit comes with a couple of constraints:

1. The list header must include a version number along with the pointer to the top-most element of the list in order to avoid the *A–B–A problem* where a concurrently executing, or interrupted, pop operation could erroneously modify the header and break the list: When popping off a list element A, the header is updated to point to the second element B, the pointer for which is read from the link field of element A. However, in the time between reading the pointer to B and updating the list header, the header might have been modified multiple times. It might again point to A, but this time A's next pointer might reference a different second element C. A compare-and-exchange operation comparing the list header to A would succeed and change it to B, which might not even be an element of the list anymore. A version number that is incremented each time the header is manipulated, and is included in the compare-and-exchange operation, prevents such erroneous matching on stale values. This requires a compare-and-exchange operation that is larger than the pointer size, and a version field that is large enough that wrap-around causing a false match is sufficiently unlikely.
2. Multiple elements cannot be removed from the list in one operation (although multiple elements can be pushed on in one operation). This latter constraint comes from the fact that one cannot safely walk the list to find the Nth element to point the header to when removing N-1 elements, because other threads can be altering the list at the same time, pushing elements on and/or popping them off.

ASF solves both of these issues. ASF eliminates the need for a version number because it allows the header to be monitored while the top-most element's link value is read and ultimately used to update the header. Any intervening manipulation of the header, or interrupt of the sequence, causes the operation to abort. Because element A's link to B is read and used to update the pointer atomically, the A-B-A problem does not exist.

ASF also allows multiple elements to be removed from the list in a single operation. Because the header can be continuously monitored while the list is being walked to find the Nth element, any manipulation of the list during this time will be detected and the operation aborted. With the use of the RELEASE instruction, there is a better chance that the list could be walked without exceeding the hardware's minimum guaranteed capacity. In any event, a suitable response to a hardware capacity limitation, or high contention, would be to simply resort to popping elements from the list one at a time, as is done today.

The following example code demonstrates single-element push and pop using ASF.

```
; PUSH_ELEM Operation:
;   (INPUT: element ptr in RAX)
;   (INPUT: list ptr in RBX)
; RAX->next = RBX->head
; RBX->head = RAX
; (RDX modified)
;
PUSH_ELEM:
retry:
    SPECULATE
    JNZ      retry
    LOCK MOV RDX, [RBX + head]
    MOV      [RAX + next], RDX
    LOCK MOV [RBX + head], RAX
    COMMIT
    RET

; POP_ELEM Operation:
;   (INPUT: list ptr in RBX)
;   (RETURN: element ptr in RAX)
; RAX = RBX->head
; IF (RBX->head != 0)
; {
;   RBX->head = RAX->next
; }
; (RDX modified)
;
POP_ELEM:
retry:
    SPECULATE
    JNZ      retry
    LOCK MOV RAX, [RBX + head]
    TEST     RAX, RAX
    JZ       end
    MOV      RDX, [RAX + next]
    LOCK MOV [RBX + head], RDX
end:
    COMMIT
    RET
```

7.2.2 FIFO queue

The following example presents a FIFO queue implemented using ASF. Without ASF, lock-free FIFO queues supporting multiple readers and writers have considerably higher overhead.

Note that ASF speculative regions can safely dereference pointers once they have been protected: Pointer modifications on other CPUs (for example when elements are removed from the list) will abort the speculative region.

```
; ENQUEUE Operation:
;   (INPUT: element ptr in RAX)
;   (INPUT: list ptr in RBX)
; RAX->next = 0
; IF (RBX->tail != 0)
; {
;   tmp_ptr_next = & RBX->tail->next
; } ELSE {
;   tmp_ptr_next = & RBX->head
; }
; *tmp_ptr_next = RAX
; RBX->tail = RAX
;
ENQUEUE:
    MOV      [RAX + next], 0
retry:
    SPECULATE
    JNZ      retry
    LOCK PREFETCH [RBX + head]
    LOCK MOV RCX, [RBX + tail]
    TEST     RCX, RCX
    JZ       empty_list
    LOCK PREFETCHW [RCX + next]
    LEA      RCX, [RCX + next]
    JMP      ok
empty_list:
    LEA      RCX, [RBX + head]
ok:
    LOCK MOV [RCX], RAX
    LOCK MOV [RBX + tail], RAX
    COMMIT
    RET

; DEQUEUE Operation:
;   (INPUT: list ptr in RBX)
;   (RETURN: element ptr in RAX)
; RAX = RBX->head
; IF (RBX->head != 0)
; {
;   RBX->head = RAX->next
;   IF (RBX->head = 0)
;   {
;     RBX->tail = 0
;   }
; }
;
DEQUEUE:
retry:
    SPECULATE
    JNZ      retry
    LOCK MOV RAX, [RBX + head]
    LOCK PREFETCH [RBX + tail]
    TEST     RAX, RAX
    JZ       end
    LOCK MOV RDX, [RAX + next]
    LOCK MOV [RBX + head], RDX
    TEST     RDX, RDX
    JNZ      end
    LOCK MOV [RBX + tail], RDX
end:
    COMMIT
    RET
```

7.2.3 Speculative region composition

ASF allows composing large speculative regions out of smaller ones. In effect, ASF flattens the hierarchy of SPECULATE-COMMIT pairs into one large speculative region.

For example, a speculative region that removes a piece of data from one FIFO queue and puts it on another one can be composed of the routines presented in the previous subsections as follows:

```
; DEQUEUE_ENQUEUE Operation:
;   (INPUT: remove-list ptr in RBX)
;   (INPUT: insert-list ptr in RAX)
DEQUEUE_ENQUEUE:
retry_spec:
    SPECULATE
    JNZ      retry_spec
    PUSH     RAX
    CALL     DEQUEUE
    POP      RBX
    CALL     ENQUEUE
    COMMIT
    RET
```

7.3 Coexistence with lock-based critical sections

ASF can be used in conjunction with traditional non-ASF lock-based critical sections by including a read declarator that refers to the lock variable and checking the value of the variable before proceeding. The ASF speculative region tests and monitors the lock variable without modifying it.

For example, consider a data structure such as a B-tree. Concurrent users of the B-tree perform frequent insert and delete operations in a lock-free manner using ASF. Occasionally the B-tree needs rebalancing for efficiency, but such an operation would be beyond ASF’s capacity. A global lock associated with the B-tree solves this problem in a straightforward manner: Each ASF speculative region that operates on the B-tree first initiates monitoring of the lock variable with a LOCK MOV and examines the current value of the lock. If the lock variable is set (indicating that some other thread is rebalancing the B-tree), the speculative region commits without doing any modifications (or programmatically aborts using the ABORT instruction) and then retries, effectively spinning on the lock until it clears.

The code that implements the rebalancing operation does not use ASF. It is a traditional lock-based critical section. It acquires the lock with (for example) a test-and-set-bit operation on the lock variable. The resulting write to the lock variable forces any active ASF speculative regions to abort, and upon retry they see that the lock variable is set and wait for it to clear. The rebalancing procedure need not be concerned with other operations that may be in progress and can be executed at any time.

```
; Delete operation
del_btree:
retry:
    SPECULATE
    JNZ      retry
    LOCK MOV EAX, btree_lock     ; Check and monitor global lock
    TEST     EAX, EAX            ; Rebalance in process?
    JE       nolock              ; No
    MOV      EAX, ABORT_REBALANCING ; Software abort code
    COMMIT                       ; Abort speculative region
    JMP      retry
nolock:
 ; Do the real work of deleting, using ASF
 ; If a rebalance starts, this section aborts
    ...
    COMMIT                       ; Delete finished
;
;----------------------------------------------------------------------
;
; Rebalance (does not use ASF)
rebalance_btree:
    LOCK BTS btree_lock, 0       ; Acquire rebalance lock
    JC       done                ; Another thread is rebalancing
 ; Do the rebalancing work
    ...
    MOV      btree_lock, 0       ; Release the lock
done:
```

This technique can also be used more generally as a fallback position for handling reference-pattern-dependent capacity limitations or even contention situations. Depending on the specific use case, it may be subject to some limitations of a traditional critical section, such as not being able to (easily) abort a partially completed update, or provide strong isolation in the face of non-mutex-based and non-ASF-based accesses to the shared data.

Appendix B

Workshop Papers

Hardware acceleration for lock-free data structures and software-transactional memory

Stephan Diestelhorst[*]
Technische Universität Dresden
Systems Engineering Group
Dresden, Germany
stephan.diestelhorst@inf.tu-dresden.de

Michael Hohmuth
Advanced Micro Devices
Operating System Research Center
Dresden, Germany
michael.hohmuth@amd.com

ABSTRACT

In this paper, we report on a new CPU-architecture extension proposal, named Advanced Synchronization Facility (ASF), which is geared toward accelerating and easing lock-free programming and software transactional memory (STM). We present an initial performance simulation and usability study of ASF's application to a lock-free data structure (a singly linked list) and to accelerating a state-of-the-art STM system, TinySTM. Our results indicate that ASF can significantly increase the throughput and scaling behavior of these workloads: Single-thread performance increased by up to 15 %, and the factor of scaling to eight CPUs increased by up to 20 %.

1. INTRODUCTION

Future CPU generations will no longer be able to increase their single-thread performance exponentially. Instead, CPUs will scale the number of processing cores. In consequence, software will no longer get faster execution speeds automatically with each hardware upgrade, but will have to be adapted to the higher level of parallelism exposed by the CPU. Existing parallelization techniques get more and more complex with an increasing number of execution threads, which is why the software industry is looking for new, less complex parallel programming paradigms.

Transactional memory is a promising programming model that provides transactions (known from database technology) that take the burden for synchronizing concurrent data access off programmers' backs. However, today's software implementations of transactional memory, known as *Software Transactional Memory* (*STM*), still inflict too much overhead for synchronization and bookkeeping, making STMs impractical for the CPU count to be expected in the near future. One way to reduce this overhead is to accelerate STMs with new hardware mechanisms.

Another promising programming paradigm is that of lock-free data structures. Many authors have shown that lock-free algorithms perform and scale well and are robust against deadlocks, but to date these algorithms have been limited by incomplete hardware support: Lock-free programming relies on atomically modifying a set of memory locations using instructions like test-and-set and compare-and-swap (CAS). However, these instructions typically operate on only one or two words of memory and have a high latency, making lock-free programming impractical for more complex data structures or when low latency is required.

In this paper, we introduce a new hardware acceleration mechanism, *Advanced Synchronization Facility* (*ASF*). ASF is an experimental AMD64 architecture extension originally intended for the acceleration of lock-free algorithms. We evaluate ASF in the contexts of lock-free data structures and STMs.

Our evaluation results indicate that ASF has excellent potential for lock-free data structure acceleration. For an integer set implemented as a singly linked list, the ASF-based implementation has both better single-thread performance and better scalability than a lock-based implementation and a conventional lock-free version based on CAS.

We also applied ASF to the implementation of an STM system, TinySTM [4]. In two STM workloads we examined, a red-black-tree-based integer set and a singly-linked-list-based one, we observed a significant increase in multiprocessor throughput. Single-thread performance was comparable to or better than the baseline STM system in each case.

This paper is organized as follows. Section 2 presents related work. In Section 3, we introduce ASF and demonstrate how it simplifies lock-free programming. Section 4 applies ASF to the example applications used in our performance study: a lock-free linked-list implementation and TinySTM. In Section 5, we describe our simulation and performance measurement environment and compare the ASF-accelerated applications developed in Section 4 to their conventional counterparts. We conclude the paper in Section 6 with an outlook on future research directions.

2. BACKGROUND AND RELATED WORK

2.1 Lock-free data structures

Lock-free data structures do not use locks to coordinate concurrent accesses, avoiding most drawbacks of traditional locks, such as deadlock and priority inversion. Herlihy [7] showed that, given an atomic CAS primitive, all concurrent data structures can be implemented in a lock-free manner. Despite this general proof, only few lock-free implementations exist, such as the singly linked list, introduced by Valois [16]. Harris' later attempt [5] fixes bugs and is conceptually simpler, suggesting that correct and well-performing lock-free implementations are not trivial to find.

ASF aims at making lock-free programming significantly easier by

[*]Stephan Diestelhorst contributed to this work while interning at Advanced Micro Devices.

Revision 1.1 (Aug 14, 2008)

providing a mechanism that is both more powerful and more flexible than traditional primitives such as CAS.

2.2 Transactional memory

Herlihy and Moss proposed transactional memory in [9], implemented in hardware. Recently, a large number of software implementations (STM) have been developed [8, 3, 6, 4], but despite steady improvements, they are still about an order of magnitude worse than native hardware in single-threaded performance. Hardware support to reduce this penalty has been proposed earlier. To keep architectural extensions modest, proposals primarily either (1) restrain the size of supported hardware transactions (e. g., HyTm [2, 10], PhTM [11]), or (2) limit the offered expressiveness (e. g., LogTM-SE [17], SigTM [14]), or both (HASTM [15]).

Each of these hardware approaches is accompanied by software that works around the limitations and provides the interface and features of STM: flexibility, expressiveness, and large transaction sizes.

ASF in contrast has a broader scope than only the acceleration of transactional memory and can be implemented with moderate hardware extensions. The result is a mechanism that has relatively small capacity (compared to those listed under (1)) and richer expressiveness (than those listed under (2)), but requires a more static setup than hardware proposals under both (1) and (2) and STMs.

2.3 Simulation

Evaluation of new hardware-extension proposals requires simulators that can provide accurate timing information. Besides the internal tools employed by CPU vendors, various tools model the microarchitecture of a modern out-of-order processor [1, 12, 13]. This paper uses PTLsim [18], because unlike other solutions it is freely available and largely supports the AMD64 instruction set. In addition, it supports *co-simulation*, transparent switching between simulation and native hardware to quickly execute uninteresting parts of the application under test. PTLsim also supports full-system simulation and features a rich CPU model, which provides detailed architectural statistics.

3. ADVANCED SYNCHRONIZATION FACILITY (ASF)

3.1 Overview

ASF is an experimental AMD64 extension that allows user- and system-level code to modify a set of memory objects atomically without requiring expensive synchronization mechanisms.

The ASF extension provides an inexpensive primitive from which higher-level synchronization mechanisms can be synthesized: for example, multi-word compare-and-exchange, load-locked-store-conditional, lock-free data structures, and primitives for software-transactional memory.

ASF is both more flexible and faster than existing lock-free atomic memory-modification approaches. Instead of offering new instructions with hardwired semantics (such as compare-and-exchange for two independent memory locations), ASF only exposes a mechanism for atomically updating multiple independent memory locations and allows software to implement the intended synchronization semantics.

ASF works by allowing software to declare critical sections that modify a specified set of protected memory locations. Protected memory that critical sections modify will become visible to other CPUs[1] either all at once (when the critical section finishes successfully) or never (if the critical section is aborted). CPUs can protect and speculatively modify up to 8 memory objects that can each be at most cache-line sized and need to be size-aligned. When ASF detects conflicting accesses to one of these objects, it aborts the critical section.

Unlike traditional critical sections, ASF critical sections do not require mutual exclusion. Multiple ASF critical sections on different CPUs can be active at the same time, allowing greater parallelism.

3.2 Critical section structure

ASF critical sections consist of two phases. In the first phase, the specification phase, software declares which memory objects should be protected. The second phase, the atomic phase, can modify these memory objects speculatively. If the atomic phase completes successfully, all such modifications become visible to all CPUs simultaneously and atomically. Otherwise, modifications to protected memory objects are discarded.

ASF introduces a set of new instructions that denote the beginning and end of ASF phases. An ASF critical section has the following structure:

- The specification phase is entered when the first declarator instruction, or *declarator*, (LOCK MOV, LOCK PREFETCH, and LOCK PREFETCHW instructions) occurs. Declarators are used to declare memory that ASF should protect.
- A VALIDATE instruction can be used in the specification phase to check whether any of the previously declared memory locations has been invalidated by a concurrent write operation.
- The ACQUIRE instruction denotes the end of the specification phase and the beginning of the atomic phase. ACQUIRE has a return code that signals whether the atomic phase has been entered successfully, and also sets the rFLAGS register accordingly. A return code of 0 signals success.
- ACQUIRE is followed by instructions that check the return code and jump to an error handler if it is not zero (typically just a JNZ).
- The atomic-phase instructions (standard x86 instructions, including standard load and store instructions) are executed.
- The COMMIT instruction denotes the end of the atomic phase.

Figure 1 shows example code that uses ASF to implement compare-and-exchange for two independent memory locations, dubbed *DCAS* for "double compare-and-swap." (This code uses immediate retry as the recovery strategy. A real implementation might have a more elaborate recovery strategy, for example exponential backoff.)

[1]In this paper, the term "CPU" refers to one logical CPU (one hardware thread executing x86 instructions), irrespective of how these logical CPUs are packaged. (Its use is synonymous to terms like "CPU core" and "x86 thread," which are not used in this paper.)

```
; DCAS Operation:
; IF ((mem1 = RAX) && (mem2 = RBX)) {
;   swap (mem1, RDI)
;   swap (mem2, RSI)
;   RCX = 1
; } ELSE {
;   RCX = 0
; }

DCAS:
retry:
    LOCK MOV R8, [mem1] ; Specification phase begins
    LOCK MOV R9, [mem2]
    ACQUIRE  RCX, 2     ; Try to enter atomic phase
    JNZ      retry      ; Retry if unsuccessful
    CMP      R8, RAX    ; Atomic-phase code
    JNZ      out
    CMP      R9, RBX
    JNZ      out
    MOV      [mem1], RDI
    MOV      RDI, RAX
    MOV      [mem2], RSI
    MOV      RSI, RBX
    MOV      RCX, 1
out:
    COMMIT              ; End of atomic phase
```

Figure 1: DCAS implemented using ASF

3.3 Critical section aborts

Critical sections can be aborted at any point because of contention, far control transfers (including those caused by interrupts and faults), or software aborts.

Specification phase aborts are signaled by an ACQUIRE return code. ACQUIRE has a count argument that must match the number of declarators, allowing it to detect whether an interrupt occurred in the specification phase.

ASF is an unusual x86 architecture feature in that ACQUIRE has setjmp-like semantics: Atomic-phase aborts not only discard all modifications to all modified protected memory objects, but also reset the instruction and stack pointers to the values they had when ACQUIRE was executed. This results in a reexecution of ACQUIRE, which now returns an error code and directs the control flow (via the following conditional jump) to the error handler.

When an interrupt occurs during a critical section, that critical section will be aborted. Note that before an interrupt or exception handler returns, operating-system code or other processes may have executed in the interim. This is of no consequence as no ASF-related state is maintained across context switches. Other processes may even have executed ASF critical sections that inspected or modified any of the locations targeted by the interrupted critical section. The interrupted software will simply reinspect the state of the shared data structure and attempt its critical section again.

3.4 Implementation and performance

There are several conceivable ways in which processors can implement ASF. Our architecture simulator currently implements ASF as follows:

- We implemented ASF on top of the cache-coherency protocol. Any contention for a protected cache line will abort the critical section in question.
- The back-up copies of protected memory locations (to be written back in case of an abort) are held in a separate per-CPU buffer, called the *locked-line buffer* (LLB).
- Our pipeline allows only one ASF critical section in flight, thereby serializing all of a CPU's critical sections. Declarator instructions starting a new critical section are prevented from issuing until the previous critical section's COMMIT instruction has been retired. In consequence, the processor does not have to track independent lock sets.

The LLB allows CPUs to evict protected, speculatively modified memory out of the caches if necessary. Despite not being part of the memory hierarchy—the LLB only holds backup data—it participates in the cache-coherency protocol and monitors for contention for protected memory regions. If the LLB detects a contending probe, it holds off the probe response until the backup copies have been written back to the memory hierarchy.

Our design is easy to implement in an existing architecture, but limits the instruction-level parallelism (ILP) that can be exploited because the CPU cannot speculate across multiple critical sections. Other than that, all ASF instructions can be fully pipelined and have little latency (about one clock cycle).

The performance can be enhanced further in several ways:

- An implementation can track multiple ASF critical-section instances in parallel to come close to the ILP exposed by an unprotected version of the code.
- An implementation can prevent instructions in a critical section from committing before COMMIT, keeping all modifications in the internal store buffer. While such a design would limit the number of instructions in a critical section to the size of the reorder buffer, it works without an LLB (thereby removing the LLB as a potential bottleneck) because all outstanding writes remain in the store buffer until COMMIT.

4. APPLYING ASF

4.1 An ASF-based linked list

In this section, we introduce an implementation of a singly linked list based on ASF. It serves as an example of how to integrate ASF into a programming language and will also be the target of our evaluation in Section 5.

Before we present the ASF-based implementation, we show a lock-based version for comparison (Figure 2). We will use this version in our performance comparison in Section 5, along with the CAS-based lock-free list implementation proposed by Harris [5].

Figure 3 shows part of the ASF-based implementation. It is essentially similar to Harris' CAS-based one, but can remove elements directly from the list instead of marking them first. The code sample highlights this difference. It also illustrates a programming-language interface to ASF, which we implemented using C macros.

```
void acquire(lock_t* lock) {
    do {
        while (lock->locked);
    } while ( CAS(&lock->locked,0,1) );
}
void release(lock_t* lock) {
    lock->locked = 0;
}

int set_remove(set_t *set, int val) {
    ...
    acquire(&set->lock);
    find(set, val, &prev, &next);

    if (next->val != val) {
        release(&set->lock);
        return 0;
    }
    prev->next = next->next;
    release(&set->lock);
    ...
    return 1;
}
```

Figure 2: Removal from a singly linked list protected by a single lock

```
int set_remove(set_t *set, int val) {
    ...
retry:
    /* Traverse the list to the element,
       without any locks / ASF protection. */
    find(set, val, &prev, &next);
    ...
    contained = 0;
    prev_next = asf_lock_load(&prev->next);
    next_next = asf_lock_load(&next->next);
    next_val  = asf_lock_load(&next->val);

    if (!asf_acquire(3)) {       /* atomic start */
         /* Could not acquire locations -> Retry */
         goto retry;
    }
    /* check for chaining errors */
    if (prev_next != next) {
        commit();
        goto retry;
    }
    if (next_val == val) {
        contained  = 1;
        prev->next = next_next;
        next->next = (node_t*)NULL;
    }
    asf_commit();                /* atomic end */
    ...
    return contained;
}
```

Figure 3: Lock-free removal from a singly linked list using ASF

Like Harris' CAS-based implementation, our ASF-based one does not support concurrent memory reclamation and requires that elements removed from the list do not change in type.[2]

4.2 STM acceleration

We now describe how we applied ASF to accelerate an STM system. We started from the idea that ASF could relieve the STM's metadata-bookkeeping tasks by monitoring memory locations for conflicting modifications in hardware instead of in software.

We used TinySTM as the baseline for our experiments [4]. TinySTM is a state-of-the-art lock-based STM system. It has comparatively low overhead and scales well to the number of CPUs found in today's shared-memory systems. TinySTM avoids some of the overheads of lock-free STMs, such as additional indirections.

TinySTM works by tracking the time interval in which the currently running transaction is valid. As long as no value newer than the end of the current interval is read, the overhead of revalidating the set of previously read memory locations can be skipped and deferred to one validation at commit time.

Read-set validation ensures that all previously read values are consistent for a given time interval. To this end, TinySTM keeps track of the read set (and the version of the previously read values) in an internal data structure that it updates on every read operation, implemented by TinySTM's `stm_load` routine. We applied ASF by monitoring and validating a part of the read set in hardware, saving some of the bookkeeping and validation overhead.

The `stm_load` routine is already quite small and well optimized, which is one of the reasons why TinySTM performs so well. The original version (Figure 4) executes the following steps:

1. Locate the metadata for the memory location.
2. Read the version number of the memory location.
3. Check whether the write lock is set. If so, abort the transaction.
4. Read the desired memory location.
5. Check again whether the write lock is set or whether the object's version number has changed. If so, abort the transaction.
6. Check whether the version of the memory location is still within or lower than the transaction's validity interval. If not, try to extend this interval—this requires revalidating the read set. If this fails, abort.
7. Append the memory location to the list of addresses to verify at the end of the transaction.

With ASF, it is possible to read the memory location with a LOCK MOV instruction to let the hardware monitor for concurrent alterations. This allows us to omit the second lock check and the

[2]This restriction can be lifted by using a doubly linked list and checking the back references during list traversal. With ASF, the changes to the presented singly linked list algorithm are small. In our experiments, this safer list was still slightly faster than Harris' singly linked list.

```
stm_word_t stm_load(stm_tx_t *tx,
                    volatile stm_word_t *addr)
{
    ...
    lock = GET_LOCK(addr);
    /* Read lock, value, lock */
    l = ATOMIC_LOAD_MB(lock);

restart:
    if (LOCK_GET_OWNED(l)) {
        /* Locked: Check if by us, if not abort. */
        ...
    }
    value = ATOMIC_LOAD_MB(addr);
    l2    = ATOMIC_LOAD_MB(lock);
    if (l != l2) { l = l2; goto restart;}
    /* Check timestamp */
    version = LOCK_GET_TIMESTAMP(l);
    /* Valid version? */
    if (version > tx->end) {
        /* No: Revalidate read-set
               if that fails abort. */
        ...
        /* Recheck lock, perhaps
           locked during validation. */
        l = ATOMIC_LOAD_MB(lock);
        if (l != l2) goto restart;
    }
    /* Good version: Add to read set */
    if (tx->r_set.nb_entries == tx->r_set.size) {
        /* Enlarge read set */
    }
    r = &tx->r_set.entries[tx->r_set.nb_entries++];
    r->version = version;
    r->lock    = lock;
    return value;
}
```

Figure 4: Simplified version of the original `stm_load` operation (TinySTM, write-through version)

version comparisons (Steps 5, 6), as well as recording the location in the read log (final step).

The resulting new `stm_load_asf` routine (Fig. 5) uses ASF to monitor memory until ASF's capacity limit is reached (tracked with a thread-local counter variable), after which it transparently falls back to the original `stm_load` implementation when further extending the read set. Additionally, `stm_load_asf` prevents allocation of another protected memory location if the most recently allocated location and the current one share one cache line. This microoptimization is possible because ASF works on the granularity of the size of one cache line.

The `stm_load_asf` routine first protects the read value using ASF, then checks the lock. The subsequent VALIDATE ensures that the value has not been updated before reading the lock, thus allowing us to read the lock only once.

The ASF-based optimization works because it is easy and fast to check the validity of the ASF-protected memory locations along with those recorded in the read log: A simple VALIDATE instruction suffices. A final VALIDATE and COMMIT in the transaction-commit code completes the modification.

```
stm_word_t stm_load_asf(stm_tx_t *tx,
                        volatile stm_word_t *addr)
{
    stm_word_t res;
    ulong cache_addr = (ulong)addr & ASF_LINE_MASK;
    if (tx->asf_last & ASF_HINT_SOFTWARE)
        return stm_load(tx, addr);

    /* Aliasing on the last ASF line */
    if (tx->asf_last == cache_addr) {
      res = ATOMIC_LOAD_MB(addr);
      goto load_validate;
    }
    /* ASF capacity exceeded */
    if (tx->asf_entries >= ASF_ENTRIES) {
      tx->asf_last = ASF_HINT_SOFTWARE;
      return stm_load(tx, addr);
    }
    /* Check that the location is unlocked */
    res          = asf_lock_load(addr);
    tx->asf_last = cache_addr;
    tx->asf_entries++;
    stm_word_t l = ATOMIC_LOAD_MB(GET_LOCK(addr));
    if ( LOCK_GET_OWNED(l) )
        return stm_load(tx, addr);
    /* Validate recent ASF read-set */
load_validate:
    long asf_inv;
    asf_validate(asf_inv, tx->asf_entries);
    if (asf_inv) {
        stm_abort_self(tx);
        return 0;
    }
    return res;
}
```

Figure 5: Transactional load using ASF

5. EVALUATION

5.1 Evaluation setup

Given the high cost for developing a new processor core, instruction-set extensions are initially evaluated with processor simulators. We have chosen PTLsim [18] and have implemented ASF as described in Section 3. Additional modifications have been made to the simulator, partially bug fixing and architectural enhancements to bring its architecture more in line with our native hardware. We build on the work Yourst introduced in [18] making PTLsim behave similar to an AMD K8 core.

To show the significance of our simulation results, we will compare results from multi-threaded benchmarks on native hardware to our tuned simulator, thereby laying the foundation for the fidelity of the evaluation of the ASF extensions.

For the native measurements we have used a dual-socket system, equipped with two AMD Opteron^{TM} processors (family 10h, Barcelona) running at 2.2 GHz. Each processor consists of four CPUs, each with private caches (L1D & L1I: 64 KByte, 2-way

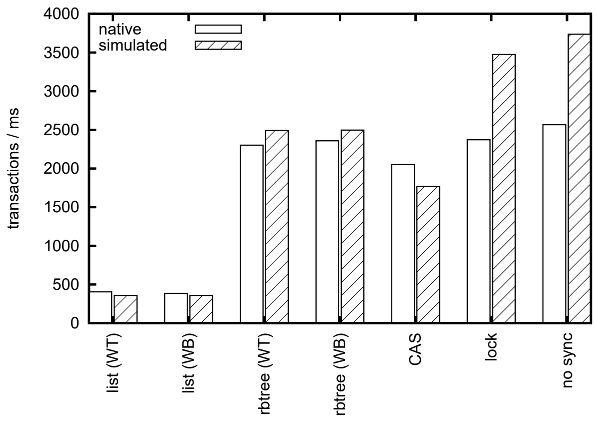


Figure 6: Single-thread performance for native and simulated execution

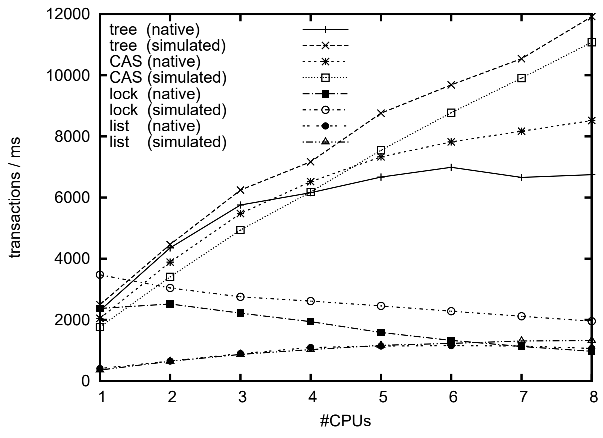


Figure 7: Multi-thread performance for native and simulated execution

set-associative; unified L2: 512 KByte, 16-way set-associative) in an exclusive hierarchy, and a shared (between the four CPUs) L3 cache (2 MB, 32-way) with a mostly exclusive (sharing-aware) configuration. Both sockets are connected with HyperTransport^{TM} links, making main memory at each socket available in a ccNUMA fashion.

Benchmarking is done using the well-known "intset" workload, a data structure that provides methods for insertion, removal, and query of a set of integers. We use the test harness included in Felber's TinySTM distribution[3] [4] and add the different implementations of the set interface. TinySTM itself is also extended to use ASF primitives, as sketched in Section 4.2.

Because simulation is rather slow (depending on the number of simulated CPUs about 100,000 times slower), we have limited the number of operations on the intset to 5000 per thread. Variance is reduced by pinning worker threads statically to CPUs (avoiding the OS's balancer) and maintaining a fixed seed for reproducibility. Other parameters of the benchmark from TinySTM remain at their default values (set with 256 entries, entries range from 0 to 65535, 20 % rate of updates).

In addition to the implementations mentioned previously (sorted singly and doubly linked lists using ASF) and those contained in TinySTM (sorted singly linked list and red–black tree using STM), we have added Harris' implementation of a lock-free singly linked list, as described in [5]. Another singly linked list simply protected with a single spin-lock and a single-threaded implementation without any locks mark the limits of (poor) scalability and excellent single-thread performance.

5.2 Simulator precision

Figure 6 compares throughput for different implementations of the intset interface on native hardware as well as inside the simulator. Simulator precision varies, dependent on the actual implementation, but is within about 20 % of native performance, except for the implementation using just a single big lock and the one without any synchronization. The larger gap for the latter originates from the tight loop that traverses the list. We have tuned the simulator to schedule instructions in the simulated CPU as efficiently as possible, a behavior that the native hardware has as well if we increase the total number of intset operations per test. The difference is likely an artifact of the testing environment, which is not as controlled as within PTLsim, or an effect of the incomplete knowledge of the AMD Opteron processor's Barcelona core in PTLsim. With the present tight traversal loop (about 3 cycles per iteration), every additional stall has a large effect on total performance.

Multi-thread results are shown in Figure 7, and although PTLsim captures the general trend, simulation results scale better, especially at working-thread numbers larger than four. This deviation is caused by our simulation's interconnect model, which does not differentiate between local (between CPUs in the same socket) and cross-socket communication. On native hardware, these links differ in both latency and available bandwidth. Therefore, our simulation can be viewed as modeling a single-socket eight-core processor instead of two four-core processors.

Additionally, PTLsim and native hardware differ in the way they treat atomic read-modify-write (RMW) instructions: PTLsim simply grabs a simulator-internal lock for the affected memory location (without any delays), whereas native hardware drains execution until the instruction is not speculative, waits for buffered stores to complete, and then executes the instruction. This leads to highly different behavior for atomic RMW instructions on contended memory locations, such as spin locks (as used in the "big-lock" implementation).

5.3 Lock-free data structures

Well designed lock-free data structures usually offer performance superior to those implemented with coarse-grained locks and STM, because of increased parallelism and reduced overhead. Figure 8 shows the results for the various implementations of the intset interface: singly linked lock-free list using ASF (labeled ASF), Harris' CAS-based lock-free implementation (CAS), and the version that uses a single lock (lock).

It can clearly be seen that the ASF implementation outperforms both the CAS-based and the lock-based implementation.

The performance advantage over the CAS-based implementation comes from three facts: First, the ASF lists do not keep deleted el-

[3]Available from http://www.tinystm.org

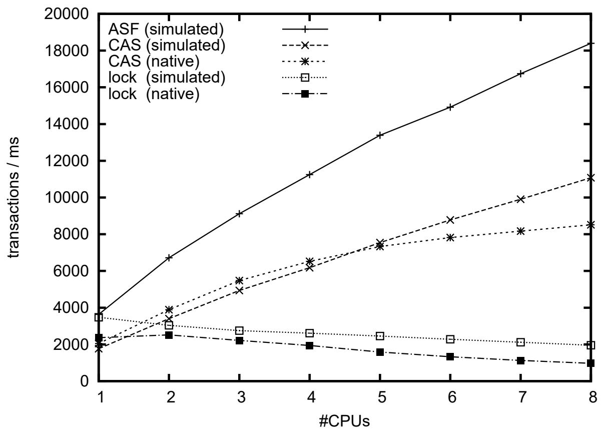


Figure 8: Lock-free data structures and single lock implementation

ements in the list for later clean-up. This keeps the list short and saves the overhead of marking and cleaning up later. Second, ASF does not guarantee progress and thus does not need any synchronization when contention on the same memory location occurs. Finally, ASF does not serialize the other memory accesses in the CPU, leaving more potential for parallel and out-of-order execution.

5.4 Acceleration of STM

In Figures 9, 10, and 11 we compare a standard TinySTM against the ASF-accelerated version from Section 4.2. The large tree and the linked list both benefit from the reduced revalidation overhead. The number of revalidations grows with the level of concurrency (frequent validity-interval extensions) and thus the accelerated version benefits more at higher CPU counts.

Surprisingly, the small tree (in Figure 9) does not profit from the acceleration although its entire read set should fit into ASF. We believe we observe this behavior because of the small read set, which makes the validation in the standard STM still reasonably fast. This reduces the advantage of ASF's fast VALIDATE and brings out some unknown overhead. We will investigate further into where this overhead of the accelerated STM comes from and how it can be avoided.

Figures 9, 10, and 11 also contain the performance of the native execution using the unmodified STM for reference. As we pointed out previously, the simulator does not yet model the limitations of the interconnect between the two sockets in the system, which obviously limits performance for the red–black tree on native hardware for CPU counts greater than four when cross-socket communication is necessary.

6. CONCLUSION AND FUTURE DIRECTIONS

In the lock-free programming and STM scenarios we have analyzed, ASF has provided substantial performance improvements—up to 15 %. Additionally, ASF significantly simplifies lock-free programming.

In the remainder of this section, we outline a few directions for future research.

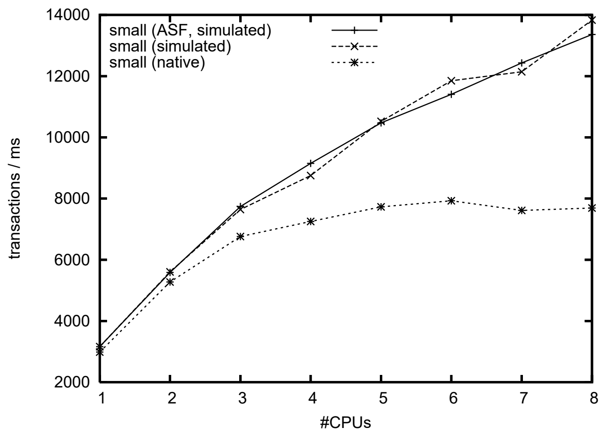


Figure 9: Comparison of ASF-accelerated and standard TinySTM with red–black tree containing 128 initial elements

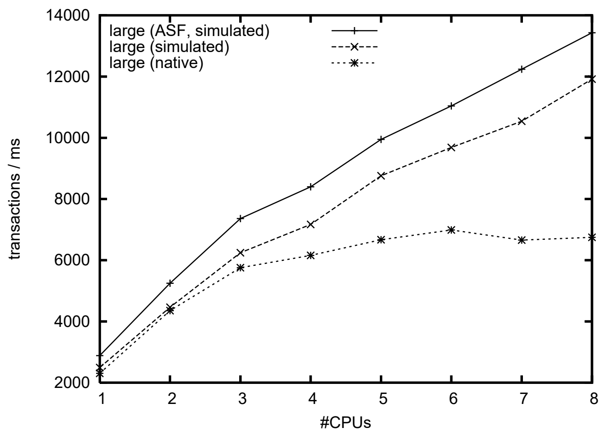


Figure 10: Comparison of ASF-accelerated and standard TinySTM with red–black tree containing 256 initial elements

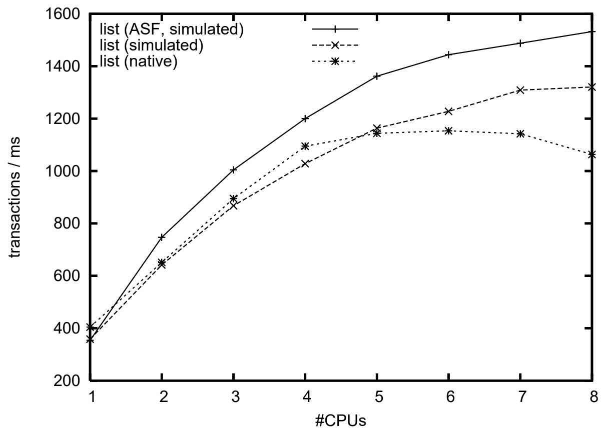


Figure 11: Comparison of ASF-accelerated and standard TinySTM with linked list

Accelerating lock-free STMs. The work presented in this paper attempted to accelerate TinySTM, one of the best performing STM systems available. TinySTM belongs to the class of STM systems that are based on locks. Lock-based STMs have largely replaced lock-free STM systems because they have less single-thread overhead while still scaling well to the number of CPUs found in today's shared-memory multiprocessor systems. However, lock-based STMs have two drawbacks to lock-free ones: susceptibility to lock-holder preemption, causing locks to be held longer than necessary; and lower scalability as the number of CPUs grows beyond what is found in today's systems. Therefore, one direction of future research is to use ASF-like hardware acceleration to reduce the overhead of lock-free STMs to the level of lock-based ones.

Compiler integration. In Section 4.1, we have sketched a C-preprocessor-macro-based interface to ASF. We acknowledge that a more robust and usable interface is needed to make use of ASF in programming languages. This requirement is reinforced by ASF being targeted not only to STM runtimes but also to lock-free application code.

The latter use case additionally raises the question of backward compatibility. The compiler interface should support application code that needs to work regardless of whether ASF is present or not.

Hardware changes. ASF can be used to protect both reads and writes against conflicting memory accesses, but the latter is bound to ASF's roll-back facility: It is currently not possible to discard memory modifications without ACQUIRE, which resets stack and instruction pointer to the values they had at the beginning of the atomic phase in case of contention. Accelerating STM-write operations would benefit from a more flexible mechanism.

Simulator precision. We outlined in Section 5.2 that our simulator lacks precision for tight loops and when modeling cross-socket communication. We plan to tackle especially the latter shortcoming to enable better prediction of highly parallel workloads.

Acknowledgments

ASF has been developed by an AMD team lead by Dave Christie and Mitch Alsup. We would like to thank Dave Christie (AMD), Andi Kleen (Novell), and Torvald Riegel (Technische Universität Dresden) for helpful discussions. We thank Matt Yourst for help with setting up PTLsim.

7. REFERENCES

[1] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[2] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (ASPLOS-XII)*, pages 336–346, New York, NY, USA, 2006. ACM.

[3] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, 2006.

[4] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2008.

[5] Tim Harris. A pragmatic implementation of non-blocking linked-lists. *Lecture Notes in Computer Science*, 2180:300, 2001.

[6] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402. October 2003.

[7] M. Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 197–206, New York, NY, USA, 1990. ACM.

[8] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing*, pages 92–101, July 2003.

[9] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.

[10] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of Symposium on Principles and Practice of Parallel Programming*, March 2006.

[11] Yosef Lev, Mark Moir, and Dan Nussbaum. PhTM: Phased transactional memory. In *Second ACM SIGPLAN Workshop on Transactional Computing*, 2007.

[12] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[13] Carl J. Mauer, Mark D. Hill, and David A. Wood. Full-system timing-first simulation. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems (SIGMETRICS)*, pages 108–116, New York, NY, USA, 2002. ACM.

[14] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. *SIGARCH Comput. Archit. News*, 35(2):69–80, 2007.

[15] Bratin Saha, Ali-Reza Adl-Tabatabai, and Quinn Jacobson. Architectural support for software transactional memory. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 185–196, Washington, DC, USA, 2006. IEEE Computer Society.

[16] John D. Valois. Lock-free linked lists using compare-and-swap. In *Symposium on Principles of Distributed Computing*, pages 214–222, 1995.

[17] Luke Yen, Jayaram Bobba, Michael M. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. LLogTM-SE: Decoupling hardware transactional memory from caches. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture(HPCA)*. February 2007.

[18] M.T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 23–34, April 2007.

Implementing AMD's Advanced Synchronization Facility in an out-of-order x86 core

Stephan Diestelhorst Martin Pohlack Michael Hohmuth Dave Christie Jae-Woong Chung Luke Yen
Advanced Micro Devices, Inc.
ASF_Feedback@amd.com

Abstract

AMD's Advanced Synchronization Facility (ASF) is an experimental architecture extension proposal aiming at making lock-free programming easier and at accelerating transactional memory systems.

We report our experiences implementing ASF in an out-of-order (OoO) CPU core simulator and our lessons learned for a future a real (silicon) implementation of ASF. Specifically, we describe how we integrated ASF into the pipeline of the simulated OoO core and how we handle the intricacies caused by the the inherently asynchronous multiprocessor memory-coherence protocol that can cause transaction aborts in any CPU state.

We present our ASF implementation's answers for four of ASF's key requirements: providing an architectural interface, rather than exposing microarchitecture directly; providing sequential memory access semantics; early abort semantics; and, capacity guarantees. We find relatively lightweight solutions for all of these requirements, but the OoO nature of the core necessitates many small changes to several CPU data structures to provide complete tracking of protected memory locations and timely reactions to conflicting memory access.

1. Introduction

Transactional programming is a promising paradigm for parallel programming that is based on a simplified programming model, but requires runtime support in the form of a software or hardware transactional memory (TM) system. Because software-only solutions come with a high overhead, hardware support for TM has been the subject of intense research in the past few years [5, 11].

Most hardware extensions (including the two implemented in real silicon [7, 10]) have been evaluated for (more or less) simple in-order processors. However, many modern commercial microprocessors employ out-of-order (OoO) cores. It is unclear whether results presented for in-order cores translate to OoO cores, for the following two reasons:

- First, OoO cores are significantly more complex than in-order cores, which also complicates the implementation of any hardware support for TM. For example, an OoO core could have multiple transactions in flight, competing for hardware resources and interfering with each other, or memory references could be reordered across the boundary of a transaction. Therefore, to assess the feasibility of TM support for an OoO core, it is important to evaluate it for an OoO model.
- Second, OoO cores exhibit a different performance profile than in-order cores. To accurately predict TM performance for modern microprocessors, it is crucial to use a simulator that closely tracks native performance for conventional workloads.

In this work we set out to implement a recent hardware-extension proposal, AMD's Advanced Synchronization Facility (ASF), in PTLsim, an instruction-driven near-cycle-accurate full-system OoO-core AMD64 simulator [24]. ASF is an experimental AMD64 architecture extension proposal developed by AMD [1]. It aims at making lock-free programming significantly easier and faster as well as accelerating TM systems [6].

We report our experiences integrating ASF with an existing OoO core simulator and our lessons learned for a future real (silicon) implementation of ASF. Specifically, we describe how we integrated ASF into the pipeline of the simulated OoO core and how we handle the intricacies caused by the the inherently asynchronous multiprocessor memory-coherence protocol that can cause transaction aborts in any CPU state.

We present our ASF implementation's answers for four of ASF's key requirements:

- Providing an architectural interface, rather than exposing microarchitecture directly
- Providing sequential memory access semantics
- Early abort semantics
- Capacity guarantees

We find that there are relatively lightweight solutions for all of these requirements, but that the OoO nature of the core necessitates many small changes to several CPU data structures to provide complete tracking of protected memory locations and timely reactions to conflicting memory access.

In this paper, we do not digress into the rationale that led to ASF's design, or any workloads we have used to validate our simulator or to evaluate ASF's performance. We refer interested readers to [6].

In the rest of this paper, Section 2 revisits the fundamentals of our work: it introduces ASF's programming interface and basic implementation options, provides background on OoO cores, and contrasts OoO speculation with ASF speculation. In Section 3 we present the implementation of ASF in the OoO core simulated by PTLsim. We discuss related work in Section 4. Section 5 summarizes our lessons learned and concludes the paper.

2. Fundamentals

2.1 ASF specification

ASF is purely experimental and has not been announced for any future product. However, it has been developed in the framework of constraints that apply to the development of a high-volume microprocessor.

```
; DCAS Operation:
; IF ((mem1 = RAX) && (mem2 = RBX)) {
;   mem1 = RDI;      mem2 = RSI;     RCX = 0;
; } ELSE {
;   RAX = mem1;      RBX = mem2;     RCX = 1;
; }  // (R8, R9, R10 modified)
DCAS:
  MOV      R8, RAX
  MOV      R9, RBX
retry:
  SPECULATE               ; Speculative region begins
  JNZ      retry          ; Page fault, interrupt, or contention
  MOV      RCX, 1         ; Default result, overwritten on success
  LOCK MOV R10, [mem1]    ; Specification begins
  LOCK MOV RBX, [mem2]
  CMP      R8, R10        ; DCAS semantics
  JNZ      out
  CMP      R9, RBX
  JNZ      out
  LOCK MOV [mem1], RDI    ; Update protected memory
  LOCK MOV [mem2], RSI
  XOR      RCX, RCX       ; Success indication
out:
  COMMIT
  MOV      RAX, R10
```

Figure 1: A DCAS primitive using ASF.

ASF provides seven new instructions for entering and leaving speculative code regions (or *speculative regions*, for short), and for accessing protected memory locations (i.e., memory locations that can be read and written speculatively and which abort the speculative region if accessed by another thread): `SPECULATE`, `COMMIT`, `ABORT`, `LOCK MOV`, `WATCHR`, `WATCHW`, and `RELEASE` (the last three are not further discussed in this paper). Figure 1 shows an example of a double compare-and-swap (DCAS) primitive implemented using ASF.

Speculative-region structure. Speculative regions have the following structure. The `SPECULATE` instruction signifies the start of such a region. It also defines the point to which control is passed if the speculative region aborts: in this case, execution continues at the instruction following the `SPECULATE` instruction (with an error code in the rAX register and the zero flag cleared, allowing subsequent code to branch to an abort handler).

The code in the speculative region indicates protected memory locations using the `LOCK MOV` instruction, which is also used to load and store protected data.

`COMMIT` and `ABORT` signify the end of a speculative region. `COMMIT` makes all speculative modifications instantly visible to all other CPUs, whereas `ABORT` discards these modifications.

ASF supports a limited form of nesting that allows simple composition of multiple speculative regions into an overarching speculative region. Nesting is implemented by flattening the hierarchy of speculative regions: memory locations protected by a nested speculative region remain protected until the outermost speculative region ends.

Aborts. Besides the `ABORT` instruction, there are several conditions that can lead to the abort of a speculative region: contention for protected memory; system calls, exceptions, and interrupts; the use of certain disallowed instructions; and, implementation-specific transient conditions. Unlike in Sun's hardware TM (HTM) design [10], TLB misses do not cause an abort.

In case of an abort, all modifications to protected memory locations are undone, and execution flow is rolled back to the beginning of the speculative region in a setjmp/longjmp-like fashion by resetting the instruction and stack pointers to the values they had directly after the `SPECULATE` instruction. No other register is rolled back; software is responsible for saving and restoring any context that is needed in the abort handler. Additionally, the reason for the abort is passed in the rAX register.

Page faults (as well as other exceptions and interrupts) abort the speculative region before they are reported to the OS. This allows the OS to resolve any faults before the speculative region is retried.

Strong isolation guarantees. ASF provides strong isolation: it protects speculative regions against conflicting memory accesses to protected memory locations from other speculative regions and regular code concurrently running on other CPUs.

In addition, all aborts caused by contention appear to be instantaneous: ASF never allows any spurious side effects caused by ASF misspeculation in a speculative region to become visible. These side effects include nonspeculative memory modifications and page faults caused by dependencies on stale data.

Eventual forward progress. ASF architecturally guarantees eventual forward progress in the absence of contention and exceptions when a speculative region protects not more than four memory lines.[1] This guarantee enables easy lock-free programming without requiring software to provide a second code path that does not use ASF. Because the guarantee only holds in the absence of contention, software still has to control contention to avoid livelock, but that can be accomplished easily (for example, by employing an exponential-backoff scheme).

2.2 Basic ASF implementation variants

We designed ASF such that a CPU design can implement ASF in various ways. The minimal capacity requirements for an ASF implementation (four transactional cache lines) are deliberately low so existing CPU designs can support simple ASF applications, such as lock-free algorithms or small transactions, with very low additional cost. On the other side of the implementation spectrum, an ASF implementation can support even large transactions efficiently.

In this section, we present two basic implementation variants. We implemented these two variants in the simulator described in later sections of this paper.

Cache-based implementation. A first variant is to keep the transactional data in each CPU core's L1 cache and use the regular cache-coherence protocol for monitoring the transactional data set.

Each cache line needs two additional bits, a speculative-read and a speculative-write bit, which are used to mark protected cache lines that have been read or written by a speculative region, respectively. These bits are cleared when the speculative region ends. In case the speculative region is aborted, the cache also invalidates all cache lines that have the speculative-write bit set.

This implementation has the advantage that potentially the complete L1 cache capacity is at disposal for transactional data. However, the capacity is limited by the cache's associativity. Additionally, an implementation that wants to provide the (associativity-independent) minimum capacity guarantee of four memory lines using the L1 needs to ensure that each cache index can hold at least four cache transactional lines that cannot be evicted by nontransactional data refills.

LLB-based implementation. An alternative ASF implementation variant is to introduce a new CPU data structure called the *locked line buffer* (LLB). The LLB holds the addresses of protected memory locations as well as backup copies of speculatively modified

[1] *Eventual* means there may be transient conditions that lead to spurious aborts, but eventually the speculative region will succeed when retried continuously. The expectation is that spurious aborts rarely occur and speculative regions succeed the first time in common cases.

memory lines. It snoops remote memory requests, and if an incompatible probe request is received, it aborts the speculative region and writes back the backup copies before the probe is answered.

The advantage of an LLB-based implementation is that the cache hierarchy does not have to be modified. Speculatively modified cache lines can even be evicted to another cache level or to main memory. (We assume the LLB can snoop probes independently from the caches and is not affected by cache-line evictions.)

Because the LLB is a fully associative structure, it is not bound by the L1 cache's associativity and can guarantee a larger number of protected memory locations. However, since fully associative structures are more costly, the total capacity typically would be much smaller than the L1 size.

2.3 Out-of-order core fundamentals

This section will briefly introduce some of the key concepts employed in current OoO microprocessors. A large number of publications exist on the matter; Hennessy and Patterson provide an extensive overview [14].

Many high-performance microprocessor cores do not process instructions in order (that is, not in the order demanded by the program executed), but rather reorder instructions to interleave long latency operations (such as complex computations and cache misses) with independent instructions, and to exploit instruction-level parallelism (ILP). With such OoO execution, book-keeping mechanisms need to be employed to maintain the sequential program semantics.

A central data structure called the reorder buffer (ROB) keeps track of in-flight instructions, their states, and required input operands. Dependencies among instructions are formed through producer–consumer relationships between instructions—operands required by one instruction are produced as results by an earlier one and are usually conveyed through registers. Because architecturally visible registers may be used by multiple independent in-flight instruction pairs, register renaming is used to separate these aliases.

Once an instruction has all input dependencies fulfilled, it is considered for execution and is eventually issued on one of the functional units of the core. Executing these instructions is not dependent on program order at this point anymore, but can proceed out of order: later instructions with fulfilled dependencies may execute before earlier instructions with unmet dependencies. Once the instructions complete execution, they forward the results to dependent in-flight instructions. The final pipeline step retires the instruction from the core. However, it processes completed in-flight instructions strictly in program order and thus maintains the sequential semantics of the code.

One source for long-latency operations is load instructions that miss in the cache. OoO execution helps here because the core can issue multiple cache-missing loads at once, thereby effectively overlapping the latencies for them. Several data structures keep track of in-flight memory operations: The load and store queue(s) of the core handle single load and store instructions, while a miss buffer keeps track of the pending cache-lines, which may be referenced by multiple in-flight memory operations.

Executing memory instructions out of order interferes with the global order of memory accesses in multiprocessor systems, impacting memory consistency guarantees [2, 21]. To free the application programmer from reasoning over the actual complex interactions, the core maintains stronger (simpler to reason about) guarantees by locally checking for consistency violations and selectively replaying memory instructions [13].

The core fetches instructions in the native AMD64 instruction-set architecture (ISA) from memory (the instruction cache) and decodes the instructions and operand information. A number of the instructions are not executed directly in the core, but are split up into multiple smaller instructions, so called microoperations (μops), instead. These flow through the pipeline independently and also retire in sequence.

2.4 Levels of speculation

Conditional branches make the code sequence dependent on data, which usually is produced only near the branch instruction and may be subject to long-latency operations, such as complex arithmetic and cache misses. To maintain a sufficiently large look-ahead instruction window, modern microprocessors employ branch prediction to forecast the instruction stream. If a conditional branch is predicted the wrong way (predicted taken vs. resolved not taken, and vice versa), instructions on the wrong branch have been executed. The instructions have to be removed from the core and their effects have to be undone, or annulled, and architectural state needs to be restored to a previous, known-good configuration.

Other predictions, such as predicting intrathread data dependencies (or their absence) for pairs of stores and loads with unresolved addresses (store-load aliases), or optimistic assumptions for scheduling conflicts and late resource shortages, may also cause re-execution of instructions.

In this paper, we will refer to this collection of speculation as employed by current OoO microprocessors as *OoO speculation* (OoO-spec). In contrast, we will refer to speculation caused by entering an ASF speculative region as *ASF speculation* (ASF-spec).

3. Pipeline and core integration

In this section we present how we integrated ASF with an OoO core simulated in an instruction-driven near-cycle-accurate AMD64 simulator, PTLsim [24]. We discuss the integration of both basic ASF implementation variants (introduced in Section 2.2) throughout this section, which we organize by high-level goals:

- In Section 3.1, we discuss the danger of using implementation artifacts as architecture, which motivates our choice of not reusing the existing OoO speculation mechanisms of the core for ASF speculation.
- Section 3.2 explains how we provide sequential ASF semantics on an OoO core.
- Section 3.3 describes our implementation of ASF's early-abort semantics.
- Section 3.4 discusses ASF's minimum capacity guarantee.

3.1 Avoiding implementation as architecture

Given that many cores already have mechanisms for keeping program state private, such as the store queue and an OoO-speculation mechanism, it is tempting to reuse these mechanisms for ASF speculation.

To illustrate, we consider the Rock processor [10], which relies on existing microarchitectural features to implement transactions: Rock uses the hardware's register checkpointing mechanism for keeping and restoring the register-file contents before starting speculation, and it keeps speculative memory updates in the core's store queue. Consequently, Rock needs to abort transactions when the capacity of either hardware resource is exhausted. Furthermore, Rock employs only a single level of speculation; the resolution of branch mispredictions, TLB misses, and other exceptional conditions abort an ongoing transaction. For these and other reasons, Rock does not give any guarantees on transaction success even in the absence of contention and interrupts.

In contrast, ASF does give an architectural forward-progress guarantee (in the absence of contention) and a minimum-capacity guarantee for speculative regions. Microarchitectural conditions such as a TLB miss or a store-queue overflow must not prevent a

```
retry1:
  SPECULATE
  ; First speculative region
  JNZ      retry1
  LOCK MOV [mem1], RBX
  ...
  COMMIT
retry2:
  SPECULATE
  ; Second speculative region
  JNZ      retry2
  LOCK MOV [mem2], RDX
  ...
  COMMIT
```

```
retry1:
  asf.spec1
  asf.mfence1
  br.nz    retry1
  asf.ld1  [mem1], RBX
  ...
  asf.commit1
retry2:
  asf.spec2
  asf.mfence2
  br.nz    retry2
  asf.ld2  [mem2], RDX
  ...
  asf.commit2
```

Figure 2: Two speculative regions close to each other, with original assembly (left) and decoded µops with added fences (right).

speculative region from ever succeeding. Because it would be impossible to provide these guarantees (including the weaker guarantee of eventual forward progress) based on the OoO microarchitecture, we chose to implement the ASF mechanisms separately from (and complementary to) the OoO mechanisms.

3.2 Sequential ASF semantics

ASF has sequential programming semantics, which a core must preserve whether it employs OoO-speculative execution or not. For example, a protected memory access occurring inside a speculative region must not be reordered to occur before the beginning of a speculative region. In this section, we discuss two aspects of executing ASF-speculative memory accesses on an OoO core. We begin with issues raised by executing protected memory accesses out of order in Section 3.2.1. Section 3.2.2 discusses how ASF resources reserved for ASF-speculative data can be managed when the instructions referencing this data are OoO-speculative and later annulled.

3.2.1 Speculative-region flow

The execution order of instructions in an OoO core is only determined through data dependencies. Instructions without dependencies can execute in arbitrary order.

For ASF speculative regions, we need to decide whether particular memory accesses (`LOCK MOV`s) are executed inside or outside a speculative region. Hence, these instructions have to be ordered with respect to the marker instructions that begin / end such a containing region (`SPECULATE` and `COMMIT`).

We add special ASF memory-fence µops during decode of the `SPECULATE` instruction to attain this goal, as shown in Figure 2. These fences operate mostly like normal fences in that they create an artificial dependency on any later memory instruction that is only resolved once the fence has retired from the core. Later memory instructions can therefore only issue after the fence has retired.

The fences are ASF-specific in that they only affect ASF-spec memory instructions and not regular memory references.

Memory instructions are then ordered by the following ordering rules (with $\rightarrow$ being similar to Lamport's *happened-before* relation [16] and $S(X)$ denoting the event of instruction X being in pipeline stage S):

$$\mathit{issue}(\texttt{asf.spec}) \rightarrow \mathit{retire}(\texttt{asf.spec}) \quad (1)$$
$$\mathit{retire}(\texttt{asf.spec}) \rightarrow \mathit{retire}(\texttt{asf.mfence}_1) \quad (2)$$
$$\mathit{retire}(\texttt{asf.mfence}_1) \rightarrow \mathit{issue}(\texttt{asf.memop}) \quad (3)$$
$$\mathit{issue}(\texttt{asf.memop}) \rightarrow \mathit{retire}(\texttt{asf.memop}) \quad (4)$$
$$\mathit{retire}(\texttt{asf.memop}) \rightarrow \mathit{retire}(\texttt{asf.commit}) \quad (5)$$

Ensuring that

$$\mathit{issue}(\texttt{asf.spec}) \rightarrow \mathit{issue}(\texttt{asf.memop}) \rightarrow \mathit{retire}(\texttt{asf.memop}) \rightarrow \mathit{retire}(\texttt{asf.commit})$$

Rules 1 and 4 trivially follow from the regular pipeline flow; Rule 5 is ensured by retiring all instructions in order. We implemented the other rules by introducing artificial dependencies in the instructions' ROB entries.

Speculative-region overlap. Short speculative regions that execute neck-to-neck, such as in Figure 2, can be in flight in the core simultaneously because of the reorder window.

Keeping track of the state of multiple simultaneous speculative regions is complicated. Whereas conventional state containers—registers—are renamed to track simultaneous usage of shared resources, making all of ASF's state renameable is complex, because it contains not only the information of the speculative region state, the abort instruction and stack pointer, but also the entirety of the bits used to track the read/write sets.

While there may be safe approximations to full renaming (such as merging read/write sets), we have chosen a more straightforward approach by serializing the execution of consecutive speculative regions in the core. The heavy pipeline-serialization mechanisms, such as flushes and stalls, have a large performance impact because of the time needed to drain and fill the pipeline, decreasing performance especially for frequent, small speculative regions that would execute in few cycles.

To avoid this performance decrease, we chose to implement the serialization through the existing dependency rules, ASF memory barriers, and by not changing the state of the speculative region until `SPECULATE` and `COMMIT` hit the pipeline's retire stage. The serialization of two consecutive speculative regions then is ensured through the following dependency chain:

$$\mathit{issue}(\texttt{asf.memop}_1) \rightarrow \mathit{retire}(\texttt{asf.memop}_1) \rightarrow \mathit{retire}(\texttt{asf.commit}_1) \rightarrow \mathit{retire}(\texttt{asf.spec}_2) \rightarrow \mathit{retire}(\texttt{asf.mfence}_2) \rightarrow \mathit{issue}(\texttt{asf.memop}_2)$$

3.2.2 Misspeculation

Section 2.3 introduced multiple instances for speculation in the OoO core and how they could fail. ASF-speculative load and store instructions are also subject to these mechanisms and this has caused several challenges for our implementation, because of the complex interactions imposed by release and redistribution of resources due to misspeculation.

Precise ASF working-set tracking. Because of OoO speculation, the core may overestimate ASF's working set: misspeculated memory instructions can add spurious ASF-spec entries to the LLB or cache before the misspeculation is detected and the corresponding memory instructions are annulled.

The overestimation does not impact correctness of the execution conceptually (all lines that need protection are protected), but has performance implications, since the additional lines artificially increase contention and also put additional pressure on the limited capacity.

It is thus desirable to detect and remove spurious entries in ASF's working sets. However, recomputing the actual ASF-spec state of a cache line when annulling an ASF-spec memory access is challenging. It depends not only on in-flight memory instructions, but also has to take into account retired ASF-spec memory instructions of the current speculative region that have referenced the cache line.

Our LLB-based ASF design supports reference counting for that particular purpose and thus can track read/write sets precisely. Adding reference-counting mechanisms to the existing L1 cache would be expensive; thus, the L1-based ASF implementation currently may overestimate the read set.

Orphan cache entries. Although not precisely tracking ASF's working set in an L1-based ASF implementation is safe in principle, under specific timing constraints it can lead to orphan ASF-spec entries in the cache even though the originating speculative region has already successfully committed or aborted.

To illustrate, consider the following sequence of events: an ASF-spec load misses in the cache and sets up an ASF-spec miss-buffer entry to track the cache miss. The load eventually is annulled because it is on a wrongly predicted branch. The cache-miss handling cannot be aborted at this time. Eventually, the speculative region commits by successfully retiring the `COMMIT` instruction (the original dependency on the cache-missing load is not present anymore, since that load has been annulled). The cache line is eventually filled into the cache and gets its spec-read bit enabled because the corresponding miss-buffer entry was tagged as ASF-spec, leading to an orphan spec-read cache line.

Note that simply resetting the cache line's spec-read bit on annulment of referencing ASF-spec loads would be incorrect, because multiple in-flight loads (ASF-spec and non-ASF-spec) may still reference the miss-buffer entry. Similarly, the miss-buffer entry's ASF-spec state cannot be simply reset because it may still be referenced by other in-flight ASF-spec loads.

A simplified version of the recomputation introduced previously solves this issue: we reuse the existing reference from a miss-buffer entry to its associated in-flight loads and count the ASF-spec-load references. We observe that no retired load can contribute to the ASF-spec state of the miss-buffer entry because loads can only retire once their cache misses have been resolved. Therefore, the number of ASF-spec loads referencing the miss-buffer entry can always be computed online by counting all nonretired (in-flight) loads with such a reference, allowing miss-buffer entries to precisely track their ASF-spec state and eliminating the need for dedicated reference counting in the L1 cache. In result, no modification to the L1 cache is necessary, and we readily implemented this mechanism to prevent orphan spec-read cache entries in our ASF prototype.

Flash clearing all ASF-spec bits (of miss-buffer and cache entries) at the end of a speculative region (retirement of the `COMMIT` instruction) would also work around the orphan-cache-entries issue. However, our recomputation approach tracks ASF's working set more closely and thus reduces the likelihood of contention.

3.3 Abort semantics

ASF has eager conflict detection and provides early-abort semantics: it defines that no side effects (e. g., memory modifications or page faults) ever become visible caused by ASF misspeculation (i. e., further execution of a speculative region after is has been aborted). The rationale is that no ASF-speculative state should be able to leak unintentionally from an aborted speculative region.

This section discusses how our ASF implementation realizes early abort semantics. Section 3.3.1 explains that, to receive timely abort information, cores need to track access conflicts with protected data in more CPU data structures than just the cache or LLB because of the asynchronous nature of memory accesses in OoO processors. In Section 3.3.2 we describe how a core recovers when it has received an abort signal.

3.3.1 Conflict detection handshake

The global linearizability of ASF speculative regions and consistency of the read and write sets is ensured through eager conflict detection. Conflict detection has to start when or before the value of the load is bound [12] or the load is performed [22]. Usually, some limited form of conflict detection and additional ordering is already employed in current multiprocessor systems to provide suitable memory-consistency semantics to the application. To keep changes to this very sensitive area of microarchitecture small, it is advisable to reuse the existing mechanisms and extend coverage of the conflict observation until the speculative region commits. However, extending the monitoring period of the legacy mechanisms is difficult, because it again involves touching sensitive hardware and furthermore may not be possible due to design decisions, such as reliance on bounded delay for certain operations, or serialization of monitoring requests.

Therefore, the responsibility for monitoring ASF-spec data eventually has to transition from the legacy mechanisms (such as the miss buffer) to ASF's monitoring facility (such as the LLB or the augmented L1 cache). During the transition, it has to be ensured that the data element is never without conflict observance, necessitating atomic transitions or overlapping intervals of conflict-detection responsibility.

For our prototype, we reuse the existing miss buffers and flag cache lines as soon as they are initially probed (for cache hits) or when they are delivered to L1 (for cache misses) with the according ASF-spec bits. Our LLB-based implementation similarly allocates entries as soon as possible, too. This design saves an additional cache lookup at a later point in time (to set the respective ASF-spec bits) and ensures overlapping contention monitoring.

The timing between the hand-over from one conflict detection mechanism to another (in particular to the enhanced L1 cache) has been a source of a lot of complexity. For example, one issue we encountered was caused by store-to-load forwarding, in which a load receives the data directly from an earlier store to the same address in the same thread. These loads effectively bypass the caches, circumventing any conflict-detection mechanism implemented in the cache. This issue was solved by creating additional entries in the L1 cache to ensure proper conflict monitoring.

3.3.2 Abort implications

Speculative regions abort whenever a conflicting concurrent data access is detected (requester-wins conflict resolution policy), which may happen asynchronously to other core timing. As outlined previously, we use the existing cache-coherence mechanisms to detect these conflicting memory accesses. Whenever an ASF-speculatively modified line is read by another core, it must be ensured that the requesting core receives the backup copy with the probe answer, and not the updated data.

Therefore, the timing between probes, replies, and the rollback operation is crucial for correct operation. To reduce the delay between the arrival of the conflicting probe and the final probe answer, we introduce partial rollbacks. These rollbacks undo modifications only for the requested line, deliver the probe answer, and then signal the core for further abort handling.

Aborting the core can then proceed independently of probe handling, at the core's discretion. The core checks for detected conflicts every cycle. If one is found, the core triggers the full rollback, encodes the abort reason into the rAX register, sets the flags register accordingly, and resets the instruction and stack pointer to the values right after `SPECULATE`. Finally, a pipeline flush and reset of the instruction fetcher (similar to the resolution of a mispredicted conditional branch) completes the abort.

Although checking for abort conditions every cycle seems sufficient on the surface, we had to address two subtleties of modern cores, which we describe in the remainder of this section.

Intra-cycle parallelism. It is possible for a specifically timed store operation to the line already rolled back to retire in the same cycle in which the abort condition was detected, but before the pipeline flush, essentially proceeding in parallel to the ongoing abort. Therefore, it is important to avoid disabling write-set tracking too early. Otherwise, the store would be able to make ASF-

```
  SPECULATE
  JNZ      abort_handler
traverse:
  LOCK MOV RDX, [RSI + val]      ; Load val
  CMP      RDX, RDI              ; Element found?
  JE       found
  LOCK MOV RSI, [RSI + next]     ; Load next pointer
  TEST     RSI, RSI              ; End of list?
  JNZ      traverse
  COMMIT                         ; Element not found
  ...
found:
  COMMIT                         ; Element found
```

Figure 3: A small linked-list traversal loop searching for a particular element, illustrating potential inflation of speculative working set because of mispredicted branches: The "Load next pointer" instruction may be mispredicted and use up ASF resources needed for maintaining ASF's capacity guarantee.

speculative modifications permanent (despite the abort of the enclosing speculative region).

µop splitting. As described in Section 2.3, native-ISA (AMD64) instructions do not have to proceed atomically through the core. Instead, they may be split up into smaller µops. These µops flow through the pipeline independently and also retire in sequence, which creates another subtlety with respect to the asynchronous nature of aborts: an abort may trigger when only a subset of the µops comprising an instruction have retired and updated the architectural state.

The most critical instructions regarding this behavior are CALL and RET, because they both access the stack pointer, the instruction pointer, and memory. Their partial retirement is, however, contained by ASF, because the abort resets both registers to a consistent value (and no guarantees for stack values below the stack pointer are given).

3.4 Capacity guarantees

The ASF specification mandates that implementations support a minimal number of read/write set entries (four cache lines), regardless of address layout and other aspects (such as TLB misses, branch misprediction, etc.).

Supporting such a guarantee under the OoO execution regime is complicated by several interactions. As described previously, ASF-speculative memory instructions may flag cache lines as speculative optimistically, artificially increasing the speculative region's working set and reducing the number of available entries that an application can really use. In particular, ASF loads behind unresolved and mispredicted branches, such as mispredicted pointer traversal loops (Figure 3), can cause this behavior.

Furthermore, loads may be issued out of order and may also fill missing cache lines in arbitrary order, depending on their residence in the underlying memory hierarchy (e. g., line present in L2 cache vs. line fetched from remote main memory). Determining precisely if and when the capacity limit is reached is therefore not clear-cut.

Non-ASF-spec memory instructions may also compete for space in the employed conflict detection device, in particular if an existing structure, such as the L1 cache, is reused for that purpose. It may be possible that non-ASF-spec entries displace ASF-spec entries, thwarting any possible capacity guarantee.

Finally, the organization of the speculative storage and tracking device heavily impacts the feasible minimal guarantee. Set-associative caches have a small worst-case minimal capacity—their associativity—because all requested addresses may alias into the same cache index. Other devices such as Bloom filters [4] may allow tracking of an arbitrary number of elements (with decreasing precision), but do not provide space for backup copies to support ASF-spec stores.

In summary, a naïve implementation does not even guarantee the worst-case capacity of the storage container (i. e., the associativity of the L1 cache for a cache-based implementation). Additional ordering and priority mechanisms are necessary to give such a guarantee, for example by carefully ordering accesses to capacity-critical parts of the storage device. However, strictly serializing all memory accesses would reduce overall performance and complicate core design.

For our LLB-based implementation, we have therefore crafted a staged buffer that has a (small) first stage where cache lines are held as long as they are only referenced by OoO-speculative inflight memory instructions. Whenever one of these instructions retires, the line in the LLB transitions to the non-OoO-spec second buffer stage. The minimal guarantee is then enforced by the non-OoO-spec second buffer stage, while the first OoO-spec stage basically controls how much (OoO-)speculation can go on. This design allows us to carefully trade performance (through higher ILP) for additional buffer space (for the additional first stage buffer).

Memory instructions have to wait until a free entry in the first stage is available before they can issue. To avoid deadlocks through OoO fill-up of the speculative buffer stage, we carefully replay later memory instructions (further down in the program flow) that have already been granted an entry to make room for the earlier ones waiting for a free entry.

Our cache-based implementation currently lacks these features, because it aims at reusing most of the existing cache implementation. Hence, it does not yet meet ASF's required minimal capacity guarantee under certain circumstances.

4. Related work

Using simulation is the most common approach for evaluating hardware-extension proposals for accelerating TMs because simulation can be realized with much less effort and lower costs than a hardware prototype. In related work, simplified simulation approaches are employed often. For example, trace generation and timing simulation are separated or simple in-order core models are applied.

This trading of simulation accuracy and speed for effort is, of course, a valid approach for research proposals in which creating a new simulation infrastructure or vastly extending existing simulators is not possible. In this paper, we present a prototype proposal targeted at implementation in current high-volume OoO microprocessors and want to achieve a very high simulation accuracy to accurately predict behavior and potential pitfalls for a silicon implementation.

Herlihy and Moss [15] employ the Proteus simulator for evaluating their TM proposal. The target programs to be simulated have to be written a superset of C, and calls into the simulator are created, for example, for calls to shared memory. Memory timing is only simulated for shared memory areas. Proteus is execution-driven, and cycle counting is embedded by a preprocessor into the simulation target programs.

Ananian et al. [3] use cycle-accurate simulation of a simplified architecture to evaluate their unbounded TM (UTM) proposal. The authors utilize UVSIM and simulate OoO MIPS 10K processors. The simulator supports cycle-accurate simulation and was extended to support a simplified HTM model (named LTM). Also, a trace-driven simulator is used that evaluates memory references and transactional operations. No detailed discussion regarding the implementability of UTM or LTM in an OoO architecture is provided.

Moore et al. [20] present log-based TM (LogTM) and employ Simics [17] for the processor model (single-issue, in-order). They use a multilevel memory model and integrated LogTM with Wisconsin GEMS [18]. The authors report that HTM instructions are implemented via Simics "magic," which leads us to believe that it would be hard to draw any conclusions for an implementation of LogTM in a real microarchitecture.

Damron et al. [8] introduce hybrid TM. Wisconsin GEMS with LogTM is used for simulation. Instead of letting hardware (LogTM here) do retries for transactions, the authors modified GEMS to hand over control for retry to software after the first failure.

Yen et al.'s LogTM-SE [23] was implemented using OoO processor cores supporting two-way SMT. Their implementation was done using a modified version of Wisconsin GEMS 2.0 for the SPARC ISA. LogTM-SE needs cache-coherence protocol changes (and NACKs probe requests on conflicts), benefits from OS adaptations, and does not provide minimal guarantees (e.g., conflict detection may produce false positives due to signature implementations). In contrast, our PTLsim implementation prevents changes to cache coherence protocols and HyperTransport, does not usually require OS interactions, and provides minimal guarantees. In this paper, we focus on the integration of HTM with OoO cores, and we believe our discussions are applicable to previously proposed HTM systems.

Moir et al. [19] discuss an adaptive TM test platform (ATMTP) and demonstrate its use in [9]. They provide a simulation environment targeted at Sun's Rock processor, especially its HTM aspects. The authors explicitly state that the model is not aimed at accuracy but for gaining early experience. ATMTP is based on Wisconsin GEMS 2.0. The detailed memory model Ruby is used, but not the OoO processor model (Opal). Instead, Simics with a simple model with one instruction per cycle covers the processor. LogTM (now integrated in Ruby) provides similar semantics to Rock's speculative cache bits. Rock's limitations (e. g., overflow of limited register window) are approximated in ATMTP. Up to now, only single-chip systems are supported. In [10], early Rock prototypes are compared to ATMTP.

Sun [10] and Azul Systems [7] have developed actual multicore processors with HTM mechanisms. Their implementations are based on in-order architectures. Both HTMs have a few notable differences to ASF. We have introduced Sun's Rock processor in Section 3.1. Azul Systems' HTM does not abort transactions in case of interrupts and exceptions and does not support selective annotation.

In contrast to the often-employed combination of GEMS and Simics with its single-issue in-order processor model, we have implemented and simulated ASF using PTLsim's OoO processor model to obtain detailed predictions and experience with implementing ASF in a modern OoO processor.

5. Lessons learned and conclusion

In this paper we outlined an implementation of ASF for the OoO core simulated by PTLsim. We reviewed four requirements imposed by ASF and how we addressed them in our ASF implementation for the OoO core:

- An architectural interface, rather than exposing microarchitecture directly
- Providing sequential memory access semantics in an OoO core
- Early abort semantics despite asynchronous memory requests
- Handling capacity guarantees in light of cache contents arriving out of order

We found relatively lightweight solutions for all of these requirements, but the OoO nature of the core necessitates many small changes to several CPU data structures to provide complete tracking of protected memory locations and timely reactions to conflicting memory access.

We found that, somewhat counterintuitively, the existing microarchitecture mechanisms for OoO speculation do not ease the implementation of ASF speculation. The reason is that ASF guarantees eventual forward progress (in the absence of contention), and an OoO core can annul speculative instructions for many more reasons than allowed for ASF aborts.

We stress that implementability is of major importance for any hardware-extension proposal, and argue based on our findings that ASF has passed this test. We believe that most of our findings directly relate to a real OoO core implementation. However, we did identify functional areas of ASF—in particular, the minimum-capacity guarantee—in which the benefits may not outweigh the additional implementation complexity. Further research is required to motivate inclusion or exclusion of the feature.

Acknowledgments

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/ 2007-2013) under grant agreement N^{o} 216852.

References

[1] *Advanced Synchronization Facility - Proposed Architectural Specification*. Advanced Micro Devices, Inc., 2.1 edition, Mar. 2009.

[2] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *Computer*, 29(12):66–76, 1996. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/2.546611.

[3] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Washington, D.C., USA, 2005. IEEE Computer Society. ISBN 0-7695-2275-0. doi: http://dx.doi.org/10.1109/HPCA.2005.41.

[4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/362686.362692.

[5] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: why is it only a research toy? *Commun. ACM*, 51(11):40–46, 2008.

[6] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Rivière. Evaluation of AMD's Advanced Synchronization Facility within a complete transactional memory stack. In *EuroSys '10: Proceedings of the 5th ACM European conference on Computer systems*. ACM, Apr. 2010.

[7] C. Click. Azul's experiences with hardware transactional memory. In *HP Labs - Bay Area Workshop on Transactional Memory*, 2009.

[8] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS*, 2006.

[9] D. Dice, M. Herlihy, D. Lea, Y. Lev, V. Luchangco, W. Mesard, M. Moir, K. Moore, and D. Nussbaum. Applications of the Adaptive Transactional Memory Test Platform. In *TRANSACT '08: 3rd Workshop on Transactional Computing*, Feb. 2008.

[10] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS*, 2009.

[11] U. Drepper. Parallel programming with transactional memory. *Communications of the ACM*, 52(2):38–43, Feb. 2009.

[12] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *ASPLOS-IV: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–257, New York, N.Y., USA, 1991. ACM. ISBN 0-89791-380-9. doi: http://doi.acm.org/10.1145/106972.106997.

[13] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. In *In Proceedings of the 1991 International Conference on Parallel Processing*, pages 355–364, 1991.

[14] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. ISBN 0-12-370490-1.

[15] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, 1993.

[16] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/359545.359563.

[17] P. S. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenström, and B. Werner. SimICS/sun4m: a virtual workstation. In *ATEC '98: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 10–10, Berkeley, Calif., USA, 1998. USENIX Association.

[18] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005. ISSN 0163-5964. doi: http://doi.acm.org/10.1145/1105734.1105747.

[19] M. Moir, K. Moore, and D. Nussbaum. The Adaptive Transactional Memory Test Platform: A Tool for Experimenting with Transactional Code for Rock. In *TRANSACT '08: 3rd Workshop on Transactional Computing*, Feb. 2008.

[20] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. LogTM: Log-based transactional memory. In *High-Performance Computer Architecture*, 2006.

[21] S. Owens, S. Sarkar, and P. Sewell. A Better x86 Memory Model: x86-TSO. In *TPHOLs '09: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, pages 391–407, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03358-2. doi: http://dx.doi.org/10.1007/978-3-642-03359-9_27.

[22] C. Scheurich and M. Dubois. Correct memory operation of cache-based multiprocessors. In *ISCA '87: Proceedings of the 14th annual international symposium on Computer architecture*, pages 234–243, New York, N.Y., USA, 1987. ACM. ISBN 0-8186-0776-9. doi: http://doi.acm.org/10.1145/30350.30377.

[23] L. Yen, J. Bobba, M. M. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LLogTM-SE: Decoupling hardware transactional memory from caches. In *HPCA*, 2007.

[24] M. T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *ISPASS*, 2007.

Compilation of Thoughts about AMD Advanced Synchronization Facility and First-Generation Hardware Transactional Memory Support

Jaewoong Chung, Luke Yen, Martin Pohlack, Stephan Diestelhorst, Michael Hohmuth, David Christie
Advanced Micro Devices, Inc.
jaewoong.chung,luke.yen,martin.pohlack,stephan.diestelhorst,michael.hohmuth,david.christie@amd.com

Abstract

AMD's Advanced Synchronization Facility (ASF) is an AMD64 extension for transactional programming and lock-free data structures. After we had released the ASF specification to the public, we contacted various transactional memory (TM) experts in academia and industry to get their opinions on ASF and suggestions for improvements. We found their feedback invaluable in understanding what the first-generation TM hardware support should look like and how to improve ASF. In this paper, we present the summary of their likes, dislikes, and concerns about ASF and explain our opinions on their suggestions. By sharing the reviews, we hope to encourage further involvement of TM experts in defining a desirable set of requirements for the first-generation TM hardware support. We believe that this will greatly help to bring out a better TM support sooner in commercial processors.

1. Introduction

Transactional memory (TM) is a promising solution to help programmers develop parallel programs [4, 7, 11, 13, 14]. With TM, programmers enclose a group of instructions within a transaction to execute them in an atomic and isolated way. The underlying TM system runs transactions in parallel as long as they have no inter-transaction data dependencies.

Advanced Synchronization Facility (ASF) is an AMD64 hardware extension for transactional programming and lock-free data structures [1, 8]. ASF consists of seven instructions. SPECULATE and COMMIT are for demarcating transaction boundaries. ABORT is for rolling back a transaction voluntarily. LOCK MOV is for selectively annotating the memory accesses to be processed transactionally. RELEASE is to semantically drop a transactional read access performed previously by LOCK MOV before COMMIT. Advanced programmers can use the ISA directly to implement lock-free data structures. They use LOCK MOV and RELEASE to control the number of words accessed transactionally between SPECULATE and COMMIT. By keeping the number of cache lines accessed in a transaction under what ASF guarantees to support in hardware, they do not need to have a software backup mechanism for transactional execution and can use ASF as a flexible extension of the existing single-word atomic primitives such as CMPXCHG [3]. Average programmers can rely on compilers that accept high-level language constructs such as atomic blocks [4] and that generate ASF-based code. WATCHR and WATCHW are used to set a system-wide access monitor on an address in a transaction and to detect memory accesses to the address originating from other cores in the system.

After the ASF specification [1] had been released, we contacted many experts ranging from professors to OS developers and game programmers to get various reviews on ASF. In this paper, we present the summary of the reviews that we find invaluable in understanding what the first-generation TM hardware support should look like and how to improve the current ASF specification. This summary paper presents what the reviewers liked and disliked about ASF. It also includes their concerns and our opinions on their suggestions. By sharing the reviews, we hope that readers formulate their own opinions on the issues discussed in the paper, make suggestions to the TM community, and identify more issues. All of this will greatly help to bring out a better first-generation TM hardware support in future commercial processors.

The paper is organized as follows. Section 2 provides an overview of ASF and programming examples. Section 3 presents the summary of the reviews and our opinions on them. Section 4 discusses related work and Section 5 concludes the paper.

2. ASF Overview

2.1 ISA

Table 1 shows the seven instructions ASF adds to the AMD64 architecture. SPECULATE starts a transaction. It takes a register checkpoint that consists of the program counter (rIP) and the stack pointer (rSP). The rest of the registers are selectively checkpointed by software in the interest of saving hardware cost. Nested transactions are supported through flat nesting — parent transactions subsume child transaction [13].

LOCK MOV moves data between registers and memory like MOV, but with two differences. First, it should only be used within transaction boundaries; otherwise, a general protection exception (#GP) is triggered. Second, the underlying ASF implementation processes the memory access by LOCK MOV transactionally (i.e., data versioning and conflict detection for the access). A conflict against the access is

Category	Instruction	Function
Transaction Boundary	SPECULATE	Start a transaction
	COMMIT	End a transaction
Transactional Memory Access	LOCK MOV [Reg], [Addr]	Load from [Addr] to [Reg] transactionally
	LOCK MOV [Addr], [Reg]	Store from [Reg] to [Addr] transactionally
ASF Context Control	ABORT	Abort the current transaction
	RELEASE [Addr]	Undo a transactional load to [Addr] done by a previous LOCK MOV
Access Monitor	WATCHR [Addr]	Detect a store from [Addr] by the other cores
	WATCHW [Addr]	Detect a load or store to [Addr] by the other cores

Table 1. ASF instruction set architecture.

Status Code	Aborted by
ASF_CONTENTION	Transaction conflict
ASF_ABORT	ABORT instruction
ASF_CAPACITY	Transaction overflow
ASF_DISALLOWED_OP	Prohibited instructions
ASF_FAR	Exception, Interrupt

Table 2. ASF abort status codes set in rAX.

detected when either a transactional access from another transaction or a non-transactional access also touches the same cache line, and at least one of the accesses is a write. This ensures strong isolation of the memory accesses by LOCK MOV [7]. Since the detection is done at cache-line granularity, there can be false conflicts due to false data sharing in a cache line. To reduce design complexity, LOCK MOV is allowed only for the WB (writeback) memory access type [3]. We provide the minimum capacity guarantee as part of ISA so that transactions that access up to four distinctive memory words with LOCK MOV are guaranteed not to suffer from capacity overflows.

Since ASF allows transactional accesses and non-transactional accesses to be mixed within transaction boundaries, it is possible that the same cache line is accessed by both access types. ASF disallows only one case where a cache line modified by a transactional access is modified by a non-transactional access later in the same transaction. This rule aims to separate the previous transactional data that will be committed at the end of the transaction from the current non-transactional data that must be committed immediately. If this rule is violated, a #GP exception is triggered.

All the other cases are allowed. A transactional access following a non-transactional access to the same address is allowed since the non-transactional access is committed when the instruction triggering the access retires. A non-transactional load following a transactional load is allowed since loads do not conflict. A non-transactional load following a transactional store is allowed since the load just reads the result of the store in program order. A non-transactional store following a transactional load is allowed simply because it does not break the memory consistency maintained by the underlying ASF system. There are two sub-cases here with regard to another thread accessing the cache line. If another thread reads from the cache line, the value written by the non-transactional store is returned since the previous transactional load does not conflict with it and the non-transactional store has been already committed. If another thread writes to the cache line, a conflict is detected against the previous transactional load regardless of the non-transactional store.

RELEASE drops isolation on a transactional load access performed to an address by LOCK MOV. The underlying ASF implementation may stop detecting conflicts to the address with the semantics that the load access never happened. It is ignored if used on an address previously modified by LOCK MOV to prohibit discarding transactional data before committing a transaction.

COMMIT concludes a transaction. The register checkpoint is discarded and the transactional data are committed. A nested COMMIT does not finish a transaction for flat nesting. The underlying ASF implementation checks if there is a matching SPECULATE. If not, a #GP exception is triggered.

ABORT is for rolling back a transaction voluntarily. Transactional data are discarded, and the register checkpoint is restored. This brings the execution flow back to the instruction following the outermost SPECULATE and terminates transactional operation. ASF supports jumping to an alternative rIP at a transaction abort by manipulating the zero flag (ZF). ZF is cleared by SPECULATE and set when a transaction is aborted. JNZ (jump when not zero) with an alternative rIP can be placed right below SPECULATE. JNZ falls through at first since ZF is cleared by SPECULATE but jumps to the alternative rIP at transaction abort since ZF is set for an aborted transaction. Since the execution flow is out of transactional context after the transaction abort, JNZ needs to jump back to SPECULATE if the transaction is to be retried. The combination of SPECULATE and JNZ is essentially identical to an alternative design in which SPECULATE takes an alternative rIP as an operand since AMD64 processors translate this kind of complex instructions into multiple micro-operations (e.g., the micro-operation versions of SPECULATE and JNZ in this case). On detecting a transaction conflict, ASF performs the same abort procedure to roll back the conflicted transaction.

There are multiple conditions for a transaction abort besides ABORT and a transaction conflict. Since it is important for software to understand why a transaction has failed and respond appropriately, ASF uses rAX to pass an abort status code to software, as shown in Table 2. Since rAX is updated with the status code at a transaction abort, compilers must not use rAX to retain a temporary variable over SPECULATE. A general purpose register is used for the status code rather than a new dedicated register that would require additional OS support to handle context switches.

There are five abort status codes. ASF_CONTENTION is set when a transaction is aborted by a transaction conflict. ASF_ABORT is set by ABORT. ASF_CAPACITY is set when a transaction is aborted due to transactional hardware-resource constraints. ASF_DISALLOWED_OP is set when a prohibited instruction is attempted within transaction boundaries. Prohibited instructions are categorized into three groups. The first group includes the instructions that may change the code segments and the privilege levels such as FAR CALL, FAR JUMP, and SYSCALL. The second group includes the instructions that trigger interrupts such as INT and INT3. The third group includes instructions that can be intercepted by the AMD-V (Virtualization) hypervisor [2].

ASF_FAR is set when a transaction is aborted due to an exception (e.g., page fault) or an interrupt (e.g., timer interrupt). Due to design simplicity, ASF rolls back transactions at exceptions and interrupts. To report which instruction triggered the exception, ASF adds a new MSR (Model Specific Register), ASF_Exception_IP, which contains the program counter (rIP) of the instruction triggering the exception. At a page fault, a transaction is aborted and as usual the page fault's linear address is stored in CR2 (Control Register 2) [3].

WATCHR and WATCHW set an access monitor to track memory

```
Push:
  SPECULATE
  JNZ <Push>
  LOCK MOV RAX, [RBX + head]
  MOV [RDX+ next], RAX
  LOCK MOV [RBX + head], RDX
  COMMIT

Pop:
  SPECULATE
  JNZ <Pop>
  LOCK MOV RAX, [RBX + head]
  CMP RAX, 0
  JE <Out>
  MOV RDX, [RAX + next]
  LOCK MOV [RBX + head], RDX
Out:
  COMMIT
```

(a) Lock-free LIFO

```
Insert:
  SPECULATE
  JNZ <Insert>
  LOCK MOV RAX, [table_lock]
  CMP RAX, 0
  JE <ActualInsert>
  ABORT
ActualInsert:
  // insert an element
  COMMIT


Resize:
  LOCK BTS [table_lock], 0
  JC <Out>
  // resize the table
  MOV [table_lock], 0
Out:
```

(b) Resizable Hashtable

Figure 1. Lock-free LIFO and resizable hashtable with ASF ISA.

accesses to an address originating from other cores. WATCHR detects a store to the address. WATCHW detects a load or a store to the address. If such accesses are detected, the transaction enclosing the instructions is aborted.

2.2 Programming with ASF

ASF supports three programming styles: transactional programming, lock-free programming, and collaboration with traditional lock-based programming.

Transactional Programming: It is straight-forward to write transactional programs with ASF. A transaction is enclosed by SPECULATE and COMMIT, and all memory accesses in the transaction are performed with LOCK MOV. ABORT is used for rolling back the transaction voluntarily.

Lock-free Programming: ASF makes it easy to construct lock-free data structures for which simple primitives such as CAS are either insufficient or inconvenient. For example, a lock-free LIFO list is a concurrent linked list that pushes and pops elements like a stack without locking. It can be implemented with a single-word CAS (Compare-And-Swap) instruction such as CMPXCHG. A new element B is pushed by first reading the top element A, setting B's next pointer to point to A, and then writing B to the link head with CAS that updates the link head only when the head still points to A. While providing better concurrency than the lock-based LIFO, the CAS-based implementation has the *ABA* problem [12] caused by the time window between reading A and executing CAS. If another thread pops A, pushes a new element C, and pushes A back during the time window, CAS will update the list header with B since the header still points to A. This breaks the list since C is lost. This issue has traditionally been addressed by appending a version number to the list head pointer which is atomically read and updated with the pointer. However, this requires a wider CAS operation and extra space consumed for the list head pointer. ASF avoids these requirements by detecting data races not based on data values but based on the accesses themselves. In the *Push* function in Figure 1(a), the current value of the head pointer (RBX + head) is loaded transactionally to a temporary register (RAX), which initiates conflict detection against

```
SPECULATE
LOCK MOV RAX, [mem1]
LOCK MOV RBX, [mem2]

/* a random op with RAX and RBX */

LOCK MOV [mem1], RAX
LOCK MOV [mem2], RBX
COMMIT
```

Figure 2. A flexible fetch-and-op pattern.

the head pointer. Then, the current head pointer value is assigned to the next pointer of a new element (RDX + next) being pushed. Finally, the head pointer is updated with the new element (RDX). In this way, the Push function is free of the ABA problem since the head pointer is protected by ASF throughout the function and a transaction conflict is detected when C is pushed. The *Pop* function works similarly except that it has an additional check (i.e., CMP RAX, 0) to see if the LIFO is empty. Moreover, ASF allows multiple elements to be popped in one atomic operation, by allowing one to safely walk the list to the desired extraction point, then updating the head pointer.

Collaboration with Lock-based Programming: It is beneficial for ASF-based code to work with traditional lock-based code in order to use it as a simple software backup mechanism covering uncommon cases. For example, consider a concurrent hashtable. It is easy to develop the ASF-based code that inserts/removes an element to/from the hashtable. Occasionally, the hashtable may need to be resized, which requires accessing all elements in the hashtable. If the hashtable is large, the limited hardware resource in a first-generation ASF implementation will cause a transaction capacity overflow.

Our recommendation is to implement a lock-based resizing code with a 1-bit hashtable lock, as shown in Figure 1(b). The insertion code starts a transaction and reads the lock bit (table_lock) with LOCK MOV. If the lock bit is not set, it jumps to *ActualInsert* and inserts a new element. If the lock bit is set, it busy-waits by aborting and retrying the transaction. The resizing code grabs the lock non-transactionally with BTS (bit test and set) [3]. The BTS instruction reads the lock bit, copies it to CF (Carry Flag), and sets the lock bit. If the lock bit is set, someone else is resizing the hashtable, in which case, it escapes the function (JC). If the lock bit was not set, it resizes the hashtable and finishes the function by resetting the lock bit. By setting the lock bit with BTS, it aborts all active insert transactions through transaction conflicts and blocks future insert transactions until the resizing code resets the lock bit. This ensures that the resizing code accesses the hashtable exclusively, and the hashtable is race-free during resizing. While the resizing code is not running, the transactions inserting elements execute in parallel since they read-share the lock bit.

3. Likes, Dislikes, Concerns, and Our Opinions

In this section, we present the summary of the reviews on our ASF specification. We also present our opinions.

3.1 Overall Rating and Usage

As the first x86 TM hardware-support specification, ASF was highly welcome with very positive expressions such as "really cool", "drooling on it", and "I want it now". Some reviewers perceived it as more of a flexible x86 atomic primitive beyond CAS due to the limited TM

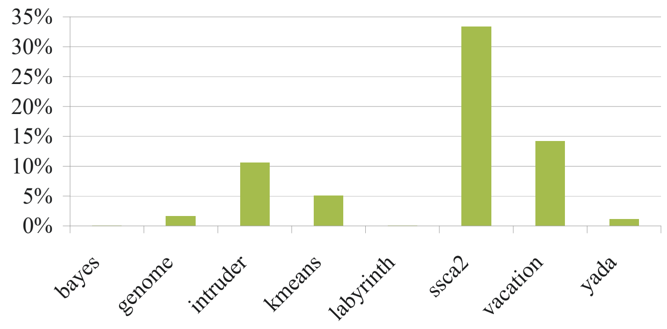


Figure 3. The ratios of the memory accesses instrumented with software barriers to all memory accesses within transaction boundaries in the STAMP benchmark suite [6]. The ratios of bayes and labyrinth are almost negligible.

support for general transactional programming with ASF. An interesting stereotype usage suggested for atomic operations was a flexible multi-word fetch-and-op as shown in Figure 2. Multiple data items are loaded transactionally (e.g., two data items in the figure), manipulated for a random op in private storage (e.g., registers or stack memory) non-transactionally, and stored back transactionally. If used, stack variables have to be alive only between SPECULATE and COMMIT. This usage encompasses many interesting cases such as multi-word compare-and-swap and multi-word fused-multiply-add (i.e., A = A + B x C). Some reviewers mentioned using ASF for speculative lock elision of small critical sections that works similar to the code in Figure 1(b).

3.2 Selective Annotation

Selective annotation of transactional memory accesses in a transaction enables *(a)* TM hardware resource saving for transactional programming and *(b)* flexible mixture of speculative accesses and non-speculative memory accesses for lock-free programming. Our observation from the existing transactional programs for STM systems is that only a small portion of memory accesses in a transaction has to be annotated with software barriers for transactional execution. Figure 3 shows the ratio of the memory accesses instrumented with software barriers to all memory accesses in transactions in the STAMP benchmark suite [6]. On average, the ratio is only 8%. There are various memory access patterns that contribute to the 92% of memory accesses that do not require software barriers. For example, as a CISC architecture, AMD64 has a small number of architectural registers and induces stack accesses to spill the registers. These stack accesses do not require conflict detection since they are to private data if the data do not escape the stack. If the stack variables are created after a transaction begins, the accesses to the variables do not need data versioning as well since the variables are effectively discarded by restoring the stack pointer when the transaction is aborted. As a result, these stack accesses do not require software barriers. Overall, the low ratio indicates that the majority of memory accesses in a transaction can be executed non-transactionally without compromising program correctness. Reviewers seem to easily acknowledge this opportunity of saving TM hardware resources.

On the other hand, as for the flexible mixture of transactional accesses and non-transactional accesses, some reviewers disliked allowing non-transactional accesses in a transaction since it could potentially

```
SPECULATE
LOCK MOV [mem1], RBX
MOV [mem2], RCX
COMMIT
```

Figure 4. A simple example that breaks the x86's memory consistency model.

weaken isolation among transactions. Other reviewers liked it since it enables TM software tools to "punch through" a transaction. This feature can, for example, be useful for debuggers to log information about outstanding transactions [10]. We advocate selective annotation in favor of giving more programming freedom to software developers. Programmers can always use transactional accesses to be on the safe side whenever they are concerned with weakening isolation.

Another concern with mixing transactional accesses and non-transactional accesses was about the exception triggered when a transactional access and a non-transactional access modify the same cache line. This can make ASF-based code less portable. For example, assume an object with two fields one of which is accessed transactionally and the other accessed non-transactionally. Depending on memory allocation schemes and runtime systems, the two fields may or may not be allocated in the same cache line, which means that the exception could be avoided in some systems but will be triggered in the other systems. This is troublesome and calls for open discussion.

3.3 Memory Access Ordering

There were questions about the cases where ASF breaks the x86's memory consistency model [3]. Figure 4 shows a very simple transaction with a transactional store and a following non-transactional store to two different cache lines. According to the x86 memory model, memory accesses should be observed in program order, which means that the transactional store should be exposed to the rest of the system first. However, in ASF, the transactional store is exposed after COMMIT is executed. The non-transactional store is exposed ahead in the reversed program order. We think that this deferred commit of the transactional store is essential for ASF to support atomicity. In our opinion, programmers should either use only transactional accesses for the code sensitive to the memory consistency model or be aware of this behavior and write the code accordingly. A COMMIT works as a memory barrier so that the memory accesses before the COMMIT are always exposed to the rest of the system ahead of the memory accesses after the COMMIT.

3.4 Minimum Capacity Guarantee

The issue of minimum capacity guarantee (i.e., the largest transaction memory footprint guaranteed not to cause capacity overflows) is one of the hottest topics not only for ASF but also for any TM hardware support in general. Should processor vendors provide any guarantee about TM or can TM support be purely best-effort (i.e., no guarantee at all)? Obviously, no guarantee is an easier choice for processor vendors and is advocated by some reviewers. However, other reviewers also pointed out that best-effort hardware transactions lack a good property of the existing atomic primitives (e.g., compare-and-swap) — that the primitives always commit in a certain way and make progress. They liked the minimum capacity guarantee supported by ASF in two ways. First, it makes best-effort hardware transactions look less like "black

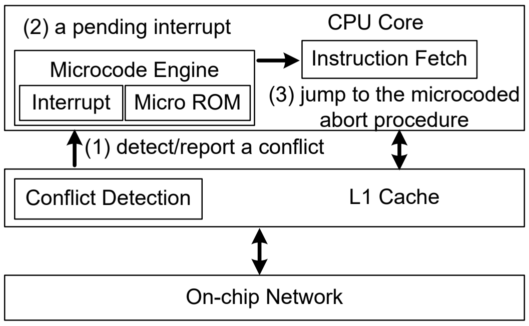


Figure 5. A time window for orphan transactions is from (1) to (3).

magic" for successful transactional execution. Second, programmers will know when they do not need to write software fallback code to deal with capacity overflows.

An obvious follow-up question was how we knew that the minimum capacity guarantee in the current ASF specification (i.e., four distinctive memory lines) was sufficient. The answer is that we did not know. As most readers can easily guess, the number four came from the likely set-associativity of four in the L1 cache. Since all AMD processors support out-of-order execution with the load/store queues, it should not be hard to increase the minimum capacity guarantee by leveraging the queues as transactional buffer as SUN Rock did [9]. However it is not easy to make a company-wide commitment on the minimum capacity guarantee for any future AMD processor with ASF support, as it may restrict the design freedom of future AMD micro-architectures.

3.5 Best-effort Maximum Capacity

Our discussion with AMD engineers brought up an interesting issue. The best-effort maximum capacity supported by ASF with no guarantee (i.e., the largest possible transaction memory footprint that may not cause capacity overflows) can also be problematic from the perspective of practical business. Assume a software product that has transactions bigger than the guaranteed minimum capacity but runs fine without capacity overflows due to additional best-effort transactional buffer provided by an AMD processor. The product does not have software fallback code to deal with capacity overflows simply because it just runs fine without the code. The problem happens with a potential next-generation AMD processor which provides a lower degree of best-effort support (e.g., a smaller best-effort transactional buffer). The software product would suffer from capacity overflows on this processor. According to the ASF specification that guarantees nothing for transactions bigger than the minimum capacity guarantee, it is clear that the software company has to add proper software fallback mechanisms. But what could happen in practice is that the company blames AMD for not being able to execute the code that used to run fine with older processors and demands AMD to fix it. One solution is to make the minimum capacity guarantee equal to the best-effort maximum capacity (i.e., no more best-effort approach). We present it as another open question for TM experts.

3.6 Abort

There was a question about the possibility of "orphan transactions" [11] in ASF. Orphan transactions are those that are marked to be aborted due to a transaction conflict by the underlying ASF system but not yet aborted. We noticed that there could be a time window for orphan transactions depending on ASF implementations. For example, one of the cost-effective ways we consider to implement the abort procedure is to deal with it as if it was a special interrupt. We refer to Figure 5 in our discussions. In step (1), the interrupt is triggered by the cache when a transaction conflict is detected with cache coherence protocol. In step (2), the interrupt is delivered to the CPU core by setting an interrupt bit. Finally, in step (3) the microcode engine checks the bit (typically at the end of issuing micro-ops of an x86 instruction) and starts the abort procedure. In this case, the window for orphan transactions starts at the time when the conflict is detected (i.e., (1)) and ends at the time when the microcode engine starts the abort procedure (i.e., (3)). With the out-of-order execution pipeline, many things can happen in this window. The potential problems with the window will have to be worked out with the designers of a specific baseline AMD processor. We intend not to expose any side effects of potential orphan transactions.

There were suggestions to clarify what happens with the registers other than rIP (program counter) and rSP (stack pointer) when a transaction aborts. The current specification guarantees the restoration of only rIP and rSP, leaving the other registers to be restored by software. The question was if those registers that were not modified in the aborted transaction are guaranteed to remain unchanged after the abort. The current specification does not guarantee it. However, we agree that this guarantee can enable interesting compiler optimizations to reduce the software–register-checkpoint overhead. We consider adding this guarantee to the next version of ASF.

3.7 Software Fallback

In comparison to traditional lock-based code, some reviewers did not like the programming pattern of combining hardware transactions and software-fallback code since it makes the best-case faster but the worst-case slower. This is not a clear win from the performance perspective unless there is a good proof showing that the best-case is the common case. We agree that it depends on application characteristics if the hardware TM support helps improve performance.

3.8 Nesting

Multiple reviewers suggested not to bother supporting nested transactions. They agreed that transaction composability with nesting is important but argued that this may have to be supported by software for first-generation TM hardware support. While it is quite easy for us to support flat nesting with a simple nesting depth counter [13], we agree that nested transactions will be rare at least with the limited TM hardware support of first-generation ASF implementations.

3.9 Contention Management

The baseline ASF contention-management policy is *attacker wins* where a transaction issuing a conflicting memory access wins a transaction conflict [5]. This can cause live-locks, and some reviewers expressed that "dead-lock is hard to deal with, but live-lock is harder". We chose the attacker wins policy for two reasons: 1) it is cheap to implement and 2) the complexity of modern processor designs tends to introduce random back-off latencies when transactions are re-executed, which can eliminate live-locks naturally in some cases. However, we agree that there still is a danger to suffer from live-locks and are developing cost-effective hardware schemes to eliminate live-locks. For now, we expect that software backup code either takes an alternative execution path or retries an aborted transaction after random backoff time.

3.10 RELEASE

In addition to the general difficulty of using early release [5], there was a concern about the case where RELEASE can unintentionally release transactionally accessed data. For example, assume two memory words are accessed transactionally but only one word is intended to be released. Depending on memory allocation mechanisms, the two memory words may or may not be located in the same cache line. The RELEASE instruction can only release whole cache lines. Consequently, if both words are located in the same cache line, both will be released together. Though we think that this problem has to be essentially dealt with by software developers, we also consider a hardware mechanism that detects this case with a set of counters incremented whenever distinctive locations of a cache line are transactionally accessed. An exception could be triggered when a RELEASE instruction is executed on a cache line with the counter value bigger than one.

3.11 Imprecise Exception

Since ASF aborts transactions at exceptions, the processor state observed by the OS exception handler is different from the processor state of the moment exceptions are triggered. In other words, ASF makes exceptions imprecise from the perspective of software. There were questions about what kinds of information the OS can get when a page fault happens. We thought that it to be enough for ASF to provide the accurate rIP and the faulting memory address. However, there were questions about other information such as the rSP value at a page fault. We were not told what the information is exactly for but certainly can consider providing more information if needed. Another question was about stepping an outstanding transaction through for debugging. We have an idea to allow for the stepping by suspending a transaction at a debug trap, running the debugger non-transactionally, and resuming the transaction when returning from the trap. However, it incurs additional cost to do that and will be considered only when there is a clear demand from software companies.

3.12 Cache-line Awareness

A reviewer disliked the fact that the current specification defines a transaction conflict in connection with cache lines instead of describing it more abstractly. We partly agree with him since this definition style may reduce the design freedom in choosing implementation schemes. However, it is highly likely that ASF implementations will leverage the existing cache coherence protocols for conflict detection. We think the concrete descriptions of the conditions for conflict detection with cache lines is better for programmers and compiler developers to help understand exactly when a transaction conflict happens.

3.13 Far Call and Ring 0

There were questions about the reason to prevent far calls (i.e., function calls that change segment registers) in a transaction. The answer is that we want to avoid implementing additional hardware schemes to eliminate security issues with program control transfer. The program control is transferred to the OS at system calls, exceptions, and interrupts [3]. If a control transfer happens in an application transaction and the underlying ASF implementation is not equipped with additional hardware to deal with program control transfer, the OS code is executed as part of the transaction. The problem is that if the transaction fails to complete, there can be security problems. For example, like most modern processors supporting security features to separate the OS and applications, the x86 architecture allows the OS and applications to use different code segments and privilege levels by changing the code segment selectors at the boundary of system calls [3]. If an ASF implementation does not have additional hardware to manage the segment registers that contain the segment selectors, an application transaction aborted in the middle of executing a system call will be restarted with the OS privilege level since the segment registers will still hold the segment selectors for the OS. This results in a security breach. Malicious programs can take advantage of this security hole to get the OS privilege level with a contrived multi-threaded TM code that forces a transaction in the middle of a system call to conflict with another transaction intentionally. We have a general hardware design to prevent security problems like this but have not reflected it yet in the specification due to its additional hardware cost.

On the other hand, the current ASF specification supports transactions in Ring 0 [3] (i.e., transactions that stay in the kernel mode throughout their lifetime).

4. Related Work

SUN developed their TM hardware support in the Rock processor [9]. There are several differences between ASF and SUN's TM support. First, ASF supports selective annotation of transactional memory accesses for efficient TM resource use. Second, it offers a minimum capacity guarantee to help programmers develop sophisticated lock-free data structures without complex backup software. Third, near function calls (i.e., function calls that do not change segment registers) and TLB misses do not abort transactions with ASF. Fourth, ASF takes a register checkpoint of rIP and rSP at the beginning of a transaction, leaving the rest of the registers to be managed by software.

5. Conclusions

In order to stimulate discussions on what the first-generation TM hardware support in commercial processors should look like, we present the summary of the various reviews on ASF and our opinions on them. We believe that this will enable a better TM hardware support to come out earlier.

6. Acknowledgments

We appreciate all TM experts who provided invaluable reviews for ASF. Their suggestions are duly considered for improvement of the current ASF specification.

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement N 216852.

7. References

[1] Advanced Synchronization Facility. http://developer.amd.com/CPU/ASF/Pages/default.aspx.

[2] AMD Virtualization. http://www.amd.com/us/products/technologies/virtualization/Pages/virtualization.aspx.

[3] AMD64 Architecture Programmer's Manual. http://developer.amd.com/documentation/guides/Pages/default.aspx.

[4] Intel c++ stm compiler. http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition-20/.

[5] J. Bobba, K. E. Moore, et al. Performance pathologies in hardware transactional memory. In *ISCA '07: Proc. of the 34th Intl. Symp. on Computer architecture*, pages 81–91, 2007.

[6] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, 2008.

[7] C. Cao Minh, M. Trautmann, J. Chung, et al. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *the Proc. of the 34th Intl. Symp. on Computer Architecture*. June 2007.

[8] D. Christie, J. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Riviere. Evaluation of amd's advanced synchronization facility within a complete transactional memory stack. In *Proceedings of Eurosys 2010 Conference*, Paris, France.

[9] D. Dice, Y. Lev, et al. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS'09: 14th Intl. Conf. on Architectural Support for Programming Languages and Operating System*, 2009.

[10] V. Gajinov, F. Zyulkyarov, et al. Quaketm: parallelizing a complex sequential application using transactional memory. In *ICS '09: Proc. of the 23rd intl. conf. on Supercomputing*, 2009.

[11] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *the Proc. of the 20th Intl. Symp. on Computer Architecture*, May 1993.

[12] IBM Corporation. *IBM System/370 Extended Architecture, Principles of Operation*. IBM Publication No. SA22-7085, 1983.

[13] A. McDonald, J. Chung, et al. Architectural Semantics for Practical Transactional Memory. In *the Proc. of the 33rd Intl. Symp. on Computer Architecture*, June 2006.

[14] K. E. Moore, J. Bobba, et al. LogTM: Log-Based Transactional Memory. In *the Proc. of the 12th Intl. Conf. on High-Performance Computer Architecture*, Feb. 2006.

From Lightweight Hardware Transactional Memory to Lightweight Lock Elision

Martin Pohlack Stephan Diestelhorst
Advanced Micro Devices, Inc.
ASF_Feedback@amd.com

Abstract

AMD's Advanced Synchronization Facility (ASF) has been evaluated in earlier work in the context of hardware and hybrid transactional memory, software transactional memory, and lock-free programming. In this work, we describe an extension to ASF for applying it in the area of lock elision (LE), which is now a well established concept in academia, but has not found its way into mainstream micro-processors.

We extended ASF to allow transactional execution of unmodified binary code, minimizing toolchain requirements and employing this extension to run existing lock-base multithreaded programs using a combined software-hardware approach. Software is responsible for demarcating transaction boundaries, for register snapshotting, for providing an elision policy, and a software backup path. Hardware in the form of an extended ASF is used for data conflict detection at runtime and rolling back modified memory in an abort case.

Early measurement results for a memcached-based setup show great potential for concurrent execution.

1. Introduction

The ubiquity of commodity multi-core processors and diminishing performance gains for single-core processors have spurred interest in making parallel programming more applicable to a wider variety of applications and tangible for a larger number of programmers. Transactional memory [19] proposes simple semantics of executing blocks of code atomically, but with enabled fine-grained parallelism. As such, transactional memory requires changes to the source code of applications and elaborate adaptations to the compiler and hardware. Although the original goal of transactional memory has been simple semantics, it turns out that software transactional memory (STM) implementations provide a wealth of weakened atomicity semantics [32] and pay a performance premium to support stronger semantics. The weakened semantics in turn compromise the simplicity goal by requiring knowledge of the underlying STM algorithm.

Although hardware solutions can provide stronger semantics, they suffer from limitations induced by the underlying microarchitecture. Existing attempts to lift these limitations complicate the microarchitecture [5, 7, 9, 22, 29], and consequently have not been adopted by any of the industry-proposed transactional memory proposals.

Speculative lock elision (SLE) [27] re-uses the existing locking infrastructure and critical-section annotations and executes non-conflicting critical sections in parallel. To be transparent on the instruction set architecture (ISA) level, SLE requires prediction and tracking logic in addition to the HTM-like speculation logic. In the AMD64 ISA, the efforts of the prediction and detection logic are complicated by the many different ways to write locks and also by several idioms using the same instructions but not demarcating critical sections (statistics updates, lock-free data structures). SLE misprediction may cause significant numbers of wasted cycles due to aborts and subsequent re-tries of code not belonging to critical sections.

In this paper, we propose to lift the simplicity requirement of transactional memory, and instead offer incremental performance improvements for incremental programmer effort, akin to earlier work on transactional lock elision (TLE) [13]. We also propose to reduce the burden on the hardware prediction and tracking logic and reduce erroneous speculation attempts by keeping entry into and exit from speculation as special instructions. For this we extend AMD's Advanced Synchronization Facility (ASF, [4]) with a *speculation-by-default* mode, allowing execution of unmodified code inside the critical sections. We effectively side-step the need for re-compilation by wrapping the pthread-library mutex functionality with support for lock elision using our new instructions. This wrapper library can be linked dynamically (at load time) to unmodified, binary applications and makes the benefits of LE available to them. Using this infrastructure, we will report on early results with a memcached-based setup in this paper.

2. AMD's Advanced Synchronization Facility

AMD's ASF [4] is a flexible, multi-word atomic primitive, much like transactional memory. To keep microarchitectural complexity small, we have opted for an almost entirely best-effort design: The processor is free to abort any ongoing transaction at any time. In particular, data conflicts, capacity overflows, exceptions, interrupts, and specific unsupported instructions can lead to an abort. However, in the absence of conflicts and when keeping the number of accessed locations within the *worst-case capacity bound*[1], ASF will not indefinitely abort transactions; the intention is to have transactions succeed on the first try most of the time.

ASF tracks data at cache-line granularity (naturally aligned blocks of 64 bytes) and aborts in-flight transactions with conflicts in the tracked working sets, employing a simple requester-wins abort policy. Conflicts are also detected between transactions and normal code, making ASF *strongly isolating*.

ASF extends the AMD64 ISA with seven new instructions: `SPECULATE` and `COMMIT` begin and end transactions; `LOCK MOV` (both loads and stores) is used for speculative data access and conflict detection, while `WATCHR` and `WATCHW` arm the conflict-detection mechanism without loading the data. The `RELEASE` instruction will remove an unmodified cache line from the set of conflict-checked locations. Finally, `ABORT` allows code within a transaction to voluntarily abort it.

An abort in ASF is performed by rolling back modifications to cache lines accessed with `LOCK MOV` or protected with `LOCK WATCHR/W`, reverting the stack pointer to the value it had when passing `SPECULATE`, returning an abort status code in the rAX and rFLAGS registers, and finally resetting the instruction pointer to the instruction following `SPECULATE`. Note that ASF does not keep a full register snapshot, nor does it explicitly have the notion of an abort handler. The former can easily be achieved by register clobbering code from the compiler, and the latter can be emulated by checking the abort code after `SPECULATE` (which is zero on success and non-zero on an abort) and branching to an appropriate handler.

Aborts caused by transient conditions, such as interrupts or page faults due to lazy paging, and conflicts, usually warrant a re-try of the transaction, assuming that the transient condition has vanished in the meantime. To this end, ASF conveys status information in the abort code, allowing streamlined re-try logic in the application. In addition, ASF makes exceptions inside transactions visible after the abort so that page faults can be handled by the operating system, *transparently* to user code.

The ASF specification does not mandate a particular implementation; we have experimented with a number of them in earlier publications [6, 8, 10, 14, 16].

Selective annotation Transactions in ASF do not mandate each memory access to be annotated with the `LOCK` prefix, but instead also allows normal `MOV` and other memory accessing instructions. These instructions will perform standard, non-speculative memory operations, but will not add the accessed memory location (the accessed cache line(s)) to the set of tracked locations causing abort on conflict.[2] This *selective annotation* allows significant reductions in the footprint of applications [10], for example, by removing thread-local or stack accesses from the limited working set. Selective annotation can be employed by compilers automatically, proving that specific locations are thread local, or by expert programmers tweaking performance through working-set size and conflict probability reductions [12, 14, 30].

3. Speculation by Default

ASF in its existing shape is targeted to be used with strong toolchain support for making the most use of its limited hardware capacity. The selective annotation feature is a key component here and can be used to great benefit given a suitable toolchain [6].

However, due to the need for affixing `LOCK` prefixes to `MOV` instructions for speculative memory accesses, it does not support execution of unmodified binary code within a transaction. Code executed that way would issue non-speculative memory instructions, which would not participate in conflict detection and would not be rolled back on abort.

To fulfill our transparency requirement for LE, we decided to introduce a new ASF mode, *speculation by default*, which changes the “polarity” of the `LOCK` prefix annotation: accesses with the prefix become non-speculative, while all other accesses are treated as speculative within transactions. For starting speculation-by-default transactions, we provide a new instruction, `SPECULATE_INV`.

The high-level effect of this inversion is that high-level language code and third-party library code can be called directly inside transactions. No additional compilation or instrumentation step is required as *all standard memory accesses* become part of the enclosing transaction and are subject to conflict detection and rollback. Toolchain support required to work with this mode is minimal; usually, only some assembly-language bindings with register clobbering are required for forcing the compiler to do the snapshotting. Of course, some of ASF’s limitations are still active: hardware capacity is limited, certain instructions are still illegal inside transactions, and far calls (e. g., kernel entries due to interrupts or system calls) will still lead to aborts.

This new mode lends itself to a software-supported implementation of lock elision.

4. Application Example

For evaluating ASF-based lock elision, we looked at lock-based multi-threaded workloads that have potential for speculative execution of concurrent critical sections. In this section, we report on our experience with memcached, which

[1] We consider four cache lines of 64 bytes each.

[2] The specifics of mixing speculative and non-speculative accesses to the same data are discussed in more detail in [4, 15].

has already received some attention in the area of scalability [2, 24, 26, 33].

4.1 memcached

Memcached is a distributed key-value in-memory database. In a large setup, several server instances run in a cluster-like environment serving from in-memory hash tables. Clients distribute load across the server instances by hashing the key part of requests [18]. Memcached is used, for example, by Facebook, Flickr, Twitter, and YouTube.

For our experiments we used the most recent public version from memcached.org (version 1.4.5). This version also supports running with multiple threads and supports the new, binary version of the memcached protocol [1].

The two most simple commands in the memcached protocol are `GET` and `PUT` for requesting and storing and string under a given key, respectively. `GET`-type requests are usually assumed to be the most common ones for memcached setups.

Memcached internally uses several locks for protecting critical data structures against races by concurrently running threads. We extended `mutrace` [25] to also measure blocking time per mutex to help identifying potentially contending locks. In our experiments, the central `cache_lock` was by far the most contended one, suggesting around 20% blocking time in the server for a high-load scenario. Other locks are used for protecting access to statistics and memory allocator data structures. All locks seen in memcached are standard pthread mutexes.

After considering the typical situation that `GET` requests usually dominate the workload, one is tempted to replace the standard mutexes with reader-writer locks. From looking into the code, it turns out that even typical read requests (like `GET`) do not follow pure read paths in the server. Occasionally, statistics have to be written, living timeouts for entries trigger, or the hashtable has to be resized. Typical `GET` paths are not read-only, but are read-mostly.

A second observation is that locks are usually only held for very brief moments. Typically, only some pointers are exchanged with brief meta-data updates. There is no `memcpy()` or similar code in the critical sections that would depend on the actual data targeted.

To summarize: *(a)* critical sections protected by the `cache_lock` are pessimistic in the sense that often only read-access is required and concurrent readers would typically be possible, and *(b)* those critical sections are usually very short, with few memory locations touched, lending themselves to a hardware-based solution with limited capacity.

4.2 Setup

Our complete measurement setup is located in one instance of the full-system simulator PTLsim, which we enhanced with ASF implementations [6] and the extension *speculation by default*. The implementation of the extension in PTLsim is relatively straightforward: The speculative bit for memory micro-ops is inverted in an early pipeline stage. The most tricky part here was not to invert several times upon potential replay of instructions. We configured PTLsim to simulate a machine with eight cores and adapted the memory latencies and the core model to be similar to those obtained on native AMD Opteron^{TM} processors of families 0Fh (K8 core) and 10h (formerly code-named “Barcelona”).

We run both the client workload generator and the memcached server inside this single machine and use four threads for each. Although a multi-machine setup would be more realistic, this approach is the only one feasible when using the PTLsim full-system simulator. PTLsim can only simulate a single machine at once and does not have support for simulation of a full networking infrastructure. Also, coupling a simulated server with a non-simulated client machine via network is infeasible, due to the huge slow-down factor in the simulator that would lead to network timeouts. In a way, this setup behaves like a worst-case scenario for the memcached server, because network latency is very small and bandwidth is not limited by a physical network.

For generating workloads we use memslap, which is very suitable for finding sustained maximum throughput for a given setup. It contains a custom modern implementation of the memcached protocol and supports the modern binary version of the protocol and many concurrent requests. Memslap is part of the standard client library libmemcached. In its default configuration, memslap uses 90% `GET` requests and 10% `PUT` requests. The default key size is 64 bytes and the default value size is 1 024 bytes. We use this default configuration unless otherwise noted.

We experimented with each memslap parameter in a native, gigabit-network setup to determine values that would maximize system throughput to put high contention on the mutexes inside memcached. As a result of these experiments, we use a window size of 10 000 for each concurrency, with 256 concurrencies simulated. We use four threads in the workload generator and let the setup execute 500 000 requests — which easily takes more than a day to complete inside a full-system simulator. Finally, we target the server at our local test machine and use the binary protocol for communication. A typical command line in our experiments looks like this: `memslap -w10k -c256 -T4 -t500000 -s localhost -B`.

4.3 Manual instrumentation

As a first approach, we manually replaced the locking code in the very common GET path of memcached (`item_get()` in thread.c) with ASF-based lock elision code (cf. to Figure 1). `item_get()` is a wrapper around `do_item_get()` that grabs the main `cache_lock` for the hash table. From a high-level point of view, `pthread_mutex_lock` is the point to start the ASF speculation and `pthread_mutex_unlock` is replaced with ASF's commit instruction. Starting the speculation comprises enforcing a register snapshot by the compiler by clobbering all relevant registers inside an inline assem-

```
item *item_get(const char *key, const size_t nkey) {
    item *it;  int retries = 5;  ulong asf_fail;

    while (retries > 0)                     // ASF lock-elision path
    {
        asf_speculate_inv(asf_fail);        //   start speculation with inverted semantics
        if (unlikely (asf_fail)) {          //   rollback point
            if (asf_hard_error(asf_fail)) {
                retries = 0;                //   hard error, don't try again
            } else {
                retries--;                  //   soft error, maybe try again
            }
            continue;
        }
        if (cache_lock.__data.__lock) {     //   pull lock into ASF's readset and check for race
            asf_abort(1);                   //   lock was already held, bail out
        }
        it = do_item_get(key, nkey);        //   actual memcached GET path
        asf_commit_();                      //   commit ASF transaction
        return it;
    }

    pthread_mutex_lock(&cache_lock);        // software fall-back path: really grab lock
    it = do_item_get(key, nkey);            //   ...
    pthread_mutex_unlock(&cache_lock);      //   ...
    return it;                              //   ...
}
```

Figure 1. Simplified manual instrumentation in `GET` path.

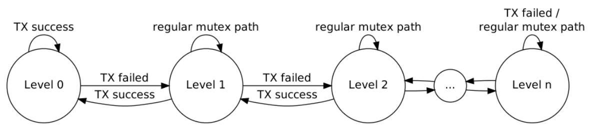


Figure 2. Software predictor state machine with the hardware lock-elision level stored per mutex and thread. Low levels incur a high chance to try hardware elision; high-levels imply a very low probability. Failed hardware attempts increase the level; successful ones decrease the level

bler snippet and adding the actual lock part of the pthread mutex struct to the transaction's read set by reading it. After the transaction is started, one has to look into the lock variable to verify that is was actually free at the time of starting the transaction. A side-effect of looking into the variable is its addition into the read set.

In case of contention or aborts, we currently employ a very simplistic approach of retrying a small number of times with ASF before falling back to the traditional approach of actually taking the lock without ASF. The write request to the actual lock done inside `pthread_mutex_lock` in the fall-back path is the point that aborts potentially running ASF transactions on other cores.

4.4 Dynamic instrumentation

Obviously, manual instrumentation has some disadvantages: access to source code is required and potentially a lot of locations have to be patched. To eliminate these problems, we use a technique of wrapping access to functions in shared libraries. We designed a library that wraps calls to `pthread_mutex_lock` and `-unlock` using Linux's LD_PRELOAD mechanisms. The core approach to elide locks is very similar to the manual instrumentation approach discussed in Section 4.3, but we now generate statistics about each mutex that we use to better guide the decision whether lock elision is feasible for a given mutex at a given time. We count the number of soft and hard aborts, the number of successful ASF transactions, and the number of normal lock operations required. Our software predictor state machine represents a notion of recent elision successes per mutex and thread and is comprised of a level and a chance for hardware re-try (cf. to Figure 2). The level is increased on ASF transaction failures, and decreased on successful ASF transactions. The current level for a given mutex determines the chance to actually try to elide the lock using ASF. For low levels, the chance is high and for high levels, the chance is very low (for level 0 mutexes, ASF will be always attempted; level 10 mutexs very rarely attempt ASF elisions but often directly fall back to normal mutex operations). We currently implement the chance with a variable, counting normal locking approaches. If this counter reaches a level-specific threshold, another ASF attempt is made.

That way, mutexes quickly adapt to the actual code protected by them at runtime and are still able to adjust to changes in workload.

In Section 4.5 we provide results for two slightly different strategies for reducing the level part of the statistics. The more aggressive strategy starts a hardware transaction directly after a level reduction, while the more conservative strategy first runs a level-specific amount of software locking rounds. The aggressive strategy is able to adapt more quickly to hardware-lock-elision-friendly changes but might overshoot doing so.

The dynamic approach that we describe here also incurs additional overhead of two forms: *(a)* call indirection and *(b)* statistics gathering. We user another level of indirection for acquiring locks. Where the manual instrumentation approach could directly inline the ASF code into the memcached code, the dynamic instrumentation puts this code into a function in a shared library, which itself does another indirect call to the actual pthread_mutex functions in the fallback case.

The dynamic part also gathers and uses per-mutex thead-local statistics, which are stored in a hash-table indexed by mutexes' addresses. This information is updated after transactions and guides the lock elision process before transactions. For the manual instrumentation approach, we simply use one static policy for the single case that we support.

The results show that the additional effort done with the dynamic instrumentation pays off in the form of increased throughput.

4.5 Results

Although we would like to report on a huge number of experiments, running everything inside a full-system simulator severely limits our resources to do so. Simulations inside PTLsim typically run six to seven orders of magnitude slower than on bare metal. We therefore only present here throughput results for the different approaches already described, but do not alter all other possible parameters.

Table 1 shows that the manual instrumentation yields a speedup of around 20%. The dynamic approaches are even more successful, with a speedup of around 30% and 35% to the baseline, respectively.

We identified the following factors contributing to the additional performance gained by the dynamic approach over the static one, despite the additional overhead for call indirection and statistics collection:

- With the dynamic approach, we instrument *all locks* in the program, not only the single, common `cache_lock`. This includes locks in linked shared libraries.
- We instrument *all paths* for those locks (e. g., also the store path in memcached).

  Additionally eliding rare paths might contribute overproportionally to reduce abort rates, as transactions in rare paths might abort serveral transactions in common paths (especially with high thread counts).
- The additional statistics collected contribute only indirectly to the additional speedup by restricting elision to effective locks. Statistics collection replaces the manual work of identifying relevant points for elision and will keep overhead down caused by useless elision.

  Statistics might also prevent some overhead on locks whose profiles are not temporally stable, for example, caused by workload changes over time. We don't expect to see this effect for the synthetic workload created by memslap though.

Table 1. Throughput results for four memcached setups.

Setup	**Throughput (transactions / s)**	**Improvement (to baseline)**
Baseline	430 470	0.0%
Manual instr.	524 430	21.8%
Dynamic instr.[a]	559 250	29.9%
Dynamic instr.[b]	576 675	34.0%

[a] Conservative back-in strategy
[b] Aggressive back-in strategy

The results reported in Table 1 are also roughly in line with the blocking times we saw with `mutrace` (cf. to Section 4.1) and indicate that lock elision may be a good technique to eliminate huge portions of overhead due to pessimistic locking in memcached.

5. Related work

AMD has proposed ASF [4] as a best-effort hardware transactional memory (HTM).

Converting critical sections requiring mutual exclusion into parallel code can be achieved in multiple ways. The original speculative lock elision proposal [27, 28] is a mechanism that does not change the CPU architecture but relies on complex prediction to transparently detect locks and critical sections. We suggest to offload the adaptation logic to a transparent software layer and thus do not need complex and inflexible hardware predictors.

Azul's Java-specific hardware transactional memory component allows parallel execution of Java's `synchronized` methods, but relies on a modified JIT compiler and custom hardware design [11].

Dice et al. use the transactional memory implementation of the Rock processor to implement transactional lock elision [13]. They modify the C++ standard vector class to use Rock's transactional memory primitives and suggest an implementation inside a Java virtual machine. Our approach does not require recompilation, but makes the benefits of lock elision available to all applications using the standard pthread mutexes. According to [13], Rock would have difficulties with such an approach due to hardware limitations in Rock's TM to execute arbitrary binary code (divisions, function calls, TLB misses, ...).

Transactional memory (TM) in general offers an alternative to lock-protected critical sections; however, several problems complicate the conversion process and hinder adoption.

Traditionally, software transactional memory (STM) provided a simple library-based interface that required programmers to manually annotate transaction begin and end, and wrap all memory accesses within the transaction. To reduce the tedious and error-prone manual effort, compilers for transactional memory have been proposed by industry [3, 23] and academia [6, 17], but language semantics are still in draft state [20, 21, 31].

The enhanced compilers provide `atomic` blocks and insert appropriate calls and new instructions automatically. Zyulkyarov et al. convert a lock-based Quake server to transactional memory [35]. Despite the use of a TM-enabled compiler, they require significant amounts of manual inspection of the source code.

We do not need compiler support and laborious software conversion, but provide instantaneous performance gains for existing critical section annotations and offer further performance improvements for programmer tuning effort.

Felber et al. discuss transactional annotation of binary code through binary translation [17], but still rely on manually annotated transaction boundaries and cause a 3x performance degradation.

Usui et al. proposed the concept of adaptive locks in [34], which combine traditional mutexes with STM-backed code paths. From a user's point of view adaptive locks look similar to our dynamic instrumentation. Behind the scenes, however, different techniques are employed. For adaptive locks, two code paths are created for all critical sections, one for either mode of operation. The authors use a full compiler tool-chain for recompiling programs and for instrumenting all memory accesses for the STM paths. We only need to provide one additional library to the system that is integrated by the dynamic linker at start time as our extension to ASF can directly execute legacy code in transactions and, unlike STM, provides strong isolation. Usui et al. also use a detailed cost-benefit model to limit the overhead for their STM mode. In our hardware-based implementation, overheads are small for execution in transactional mode. Our policy therefore only uses a simple and low-overhead success predictor but not expensive-to-determine, anticipated costs. In the future, more complex workloads or less capable hardware implementations may benefit from more elaborate statistics.

6. Conclusion

For this work we applied AMD's Advanced Synchronization Facility (ASF) proposal (a hardware transactional memory) to the domain of lock elision. We extended ASF with the speculation-by-default mode to allow better re-using of existing code and to work with a smaller and simpler toolchain.

We used a combination of software and hardware approaches for lock elisions. Software was used for detecting the actual locks (by manual instrumentation and dynamic instrumentation), for register snapshotting, for providing a software back-up path, and for implementing the policy of when to elide. Hardware, in the form of ASF, was used for detecting actual data conflicts at runtime and rolling back in case of conflict. We experimented with different software approaches and found that a more complex software scheme paid off in the form of higher transaction throughput for the memcached workload.

More investigations with more use cases are obviously needed (and underway) to get a broader understanding. This paper is meant as an initial report on our approach. These early results show interesting speedups in the area of 20–35%.

Acknowledgements

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement N^{o} 216852. We also thank our colleagues Michael Hohmuth and Dave Christie for fruitful discussions and feedback and the anonymous reviewers for their advice on the paper.

References

[1] BinaryProtocolRevamped. `http://code.google.com/p/memcached/wiki/BinaryProtocolRevamped`.

[2] Multithreading support in memcached. `http://code.sixapart.com/svn/memcached/trunk/server/doc/threads.txt`.

[3] Transactional memory in gcc. `http://gcc.gnu.org/wiki/TransactionalMemory`.

[4] Advanced Micro Devices, Inc. *Advanced Synchronization Facility - Proposed Architectural Specification*, 2.1 edition, March 2009.

[5] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture*, pages 316–327. Feb 2005.

[6] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne Riviere. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In Christine Morin and Gilles Muller, editors, *EuroSys*, pages 27–40. ACM, 2010.

[7] Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Brad Calder, and Osvaldo Colavin. Unbounded page-based trans-

actional memory. In John Paul Shen and Margaret Martonosi, editors, *ASPLOS*, pages 347–358. ACM, 2006.

[8] Jaewoong Chung, David Christie, Martin Pohlack, Stephan Diestelhorst, Michael Hohmuth, and Luke Yen. Compilation of Thoughts about AMD Advanced Synchronization Facility and First-Generation Hardware Transactional Memory Support. In *TRANSACT '10: 5th Workshop on Transactional Computing*, 2010.

[9] JaeWoong Chung, Chi Cao Minh, Austen McDonald, Travis Skare, Hassan Chafi, Brian D. Carlstrom, Christos Kozyrakis, and Kunle Olukotun. Tradeoffs in transactional memory virtualization. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 371–381, New York, N.Y., USA, 2006. ACM Press.

[10] Jaewoong Chung, Luke Yen, Stephan Diestelhorst, Martin Pohlack, Michael Hohmuth, David Christie, and Dan Grossman. ASF: AMD64 Extension for Lock-Free Data Structures and Transactional Memory. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 39–50, Washington, DC, USA, 2010. IEEE Computer Society.

[11] Cliff Click. Azul's experiences with hardware transactional memory. In *HP Labs - Bay Area Workshop on Transactional Memory*, 2009.

[12] Luke Dalessandro, Fraincois Carouge, Sean White, Yossi Lev, Mark Moir, Michael Scott, and Michael Spear. Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory (to appear). In *ASPLOS '11: Proceeding of the 16th international conference on Architectural support for programming languages and operating systems*, 2011.

[13] Dave Dice, Yossi Lev, Mark Moir, Dan Nussbaum, and Marek Olszewski. Early experience with a commercial hardware transactional memory implementation. Technical report, Mountain View, CA, USA, 2009.

[14] Stephan Diestelhorst and Michael Hohmuth. Hardware acceleration for lock-free data structures and software-transactional memory. In *EPHAM*, 2008.

[15] Stephan Diestelhorst, Michael Hohmuth, and Martin Pohlack. Sane Semantics of Best Effort Harware Transactional Memory. September 2010.

[16] Stephan Diestelhorst, Martin Pohlack, Michael Hohmuth, Dave Christie, Jae-Woong Chung, and Luke Yen. Implementing AMD's Advanced Synchronization Facility in an out-of-order x86 core. In *TRANSACT '10: 5th Workshop on Transactional Computing*, 2010.

[17] Pascal Felber, Christof Fetzer, Ulrich Müller, Torvald Riegel, Martin Süßkraut, and Heiko Sturzrehm. Transactifying applications using an open compiler framework. In *TRANSACT*, August 2007.

[18] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004:5–, August 2004.

[19] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[20] Intel. *Intel® Transactional Memory Compiler and Runtime Application Binary Interface*. Intel, 1.0.1 edition, November 2008.

[21] Intel. *Draft Specification of Transactional Language Constructs for C++*. Intel, IBM, Sun, 1.0 edition, August 2009.

[22] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. Logtm: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265. Feb 2006.

[23] Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. Design and implementation of transactional constructs for C/C++. In *OOPSLA '08: Proceedings of the 23rd annual ACM SIGPLAN conference on Object-oriented programming languages, systems, and applications*, 2008.

[24] Trond Norbye. Trond norbye's weblog: Scale beyond 8 cores?, July 2009. `http://blogs.sun.com/trond/entry/scale_beyond_8_cores`.

[25] Lennart Poettering. Measuring lock contention, September 2009. `http://0pointer.de/blog/projects/mutrace.html`.

[26] Zoran Radovic. Scaling memcached: 500,000+ operations/second with a single-socket ultrasparc t2, May 2009. `http://blogs.sun.com/zoran/entry/scaling_memcached_500_000_ops`.

[27] Ravi Rajwar and James R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *MICRO 34*, 2001.

[28] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *ASPLOS*, pages 5–17, 2002.

[29] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 494–505. IEEE Computer Society, Jun 2005.

[30] Torvald Riegel, Patrick Marlier, Martin Nowack, Pascal Felber, and Christof Fetzer. Optimizing hybrid transactional memory: The importance of nonspeculative operations. Technical Report TUD-FI10-06-Nov.2010, Technische Universität Dresden, November 2010. Full version of the DISC 2010 brief announcement.

[31] Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Robert Geva, Yang Ni, and Adam Welc. Towards transactional memory semantics for C++. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 49–58, New York, NY, USA, 2009. ACM.

[32] Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, pages 338–339, New York, NY, USA, 2007. ACM.

[33] Shanti Subramanyam. Multi-instance memcached performance, April 2009. `http://blogs.sun.com/shanti/entry/multi_instance_memcached_performance`.

[34] Takayuki Usui, Reimer Behrends, Jacob Evans, and Yannis Smaragdakis. Adaptive Locks: Combining Transactions and Locks for Efficient Concurrency. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, Washington, DC, USA, 2009. IEEE Computer Society.

[35] Ferad Zyulkyarov, Vladimir Gajinov, Osman S. Unsal, Adrián Cristal, Eduard Ayguadé, Tim Harris, and Mateo Valero. Atomic quake: using transactional memory in an interactive multiplayer game server. In Daniel A. Reed and Vivek Sarkar, editors, *PPOPP*, pages 25–34. ACM, 2009.

Sane Semantics of Best-effort Hardware Transactional Memory

Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack
Advanced Micro Devices, Inc.
stephan.diestelhorst@amd.com michael.hohmuth@amd.com
martin.pohlack@amd.com

August 18, 2010

1 Introduction

Transactional memory's (TM) biggest promise is that of making it easier to devise scalable multi-core programs. Arguably the biggest simplification of reasoning about parallel code with TM comes from *atomicity*: transactions either take effect instantaneously, or not at all. TM frees the programmer from reasoning how this atomicity is achieved and asks only where it should be employed.

Commercial proposals and implementations of hardware TM, such as Sun's Rock [2] and AMD's ASF [1], face a number of limitations and also propose extensions to the all-or-nothing semantics of TM, essentially permitting a set of visible side-effects on various levels.

In this abstract (and talk), we will outline several spots of weakened semantics, discuss implications for applications through some examples, and provide a solution within our ASF framework. Because this is work in progress, we would like to discuss whether our specification is useful and whether it is sufficiently detailed and clear.

2 Weakened Atomicity

There are several reasons for weakened atomic semantics. Some stem from practical requirements, such as software compatibility; others are imposed by conscious resource usage and the need to keep the underlying implementation simple.

2.1 Mixed-mode accesses

ASF [1] allows non-transactional accesses from within hardware transactions[1], and we believe Rock [2] has the same support. Conceptually, non-transactional accesses have several interesting properties:

- they do not cause aborts of the transaction on concurrent accesses;

[1]ASF calls these "speculative regions", we will use the term "transaction."

- they do not occupy resources needed for tracking the transactional working set; and,
- they allow intentional export of state from uncommitted transactions.

While the general idea of these accesses is rather clear, the exact usage has provoked interesting feedback and requests for clarification from users of ASF. For example, can non-transactional accesses be used to synchronise two transactions out of band? This depends on the *timeliness* of the global visibility of non-transactional stores relative to the commit of the hardware transaction.

Generally, memory semantics' specifications and their implementations guarantee that values stored will be visible eventually (and how they are ordered relative to other shared memory operations). An HTM implementation might buffer all stores (transactional and non-transactional) until successful commit of the transaction, and still adhere to the existing memory semantics.

However, the following code sequence will not commit with such an implementation:

```
Shared: a = b = 0;
Thread 1:                   Thread 2:
tx_start();                 tx_start();
...                         ...
  nontx_store(*a, 1);         while(!nontx_load(*a));
  while(!nontx_load(*b))      nontx_store(*b, 1);
tx_commit();                tx_commit();
```

A clear specification of the ordering of non-speculative memory accesses with respect to transaction commit is thus needed to reason about code wishing to employ this pattern.

2.2 Side-effects and aborts

For several reasons, it is not feasible to continue hardware transactions upon entry into the operating system and context switch: OSes ideally should not need to change for HTM support; the state associated with the transactional working set is large and difficult to grab from and inject to the hardware; and availability / security concerns mandate that kernel code should not get interrupted due to unknown transactions in user space.

If a transaction causes a kernel entry, what happens to the reason for the entry? Should the kernel still be invoked? Because the mechanics are similar to the disputed behaviour of exceptions in transactions in C++, can we learn something there?

Reasons for kernel entries that do not originate from the transaction itself, such as interrupts, should cause a kernel entry.

However, what if the reason for the kernel invocation is based inside the transaction? Strict atomic semantics mandates that these invocations do not happen. However, page faults (and other OS-handled transient conditions) are good candidates for circumstances in which resolving the fault outside the transaction makes sense; otherwise, a retry would likely hit the same condition again, making complex recovery methods necessary [2].

Nevertheless, there is a requirement that these faults are not caused by spurious, invalid data. We would certainly like to ensure that the state operated on is always consistent. However, a hardware implementation without permitted side-effects may be free to continue execution of transactions running on inconsistent state, just buffering all transactional stores and discarding them.

With visible side-effects of exceptions and non-speculative memory accesses, we need to specify the (visible) *timeliness of the aborts*.

For ASF, we have chosen to make page faults and other exceptions visible, fuelled by the assumption that code inside a transaction always runs on consistent state.

2.3 Imperfect Register Snapshots

Yet another side channel through which data can leak is through the partial register snapshots of ASF. While some of this state is beneficial (such as the page fault address in the preceeding example), some of it is disturbing, such as the unclear state of the general purpose registers.

The current ASF spec leaves it open, whether registers not touched by the transaction have to keep their value. While this can be fixed easily, the then following question remains: can we make further assumptions on the register values? Consider the following assembly example:

```
SPECULATE
...                 (RCX not modified here)
CMP %RBX, $0
JE over
MOV %1, %RCX
over:
...                 (RCX not modified here)
COMMIT
```

In case of an abort, can the programmer deduce that if $RBX = 0$, RCX will not have changed its value?

2.4 Spurious Contention

Relying solely on atomicity is not enough for a sane TM semantics, however, because an implementation may be free to abort all transactions executed. Ideally, transactions should abort only when absolutely necessary (i.e., on unresolvable conflict), but several aspects demand simplifications: estimating the minimal set of transactions to abort is complex, so Rock and ASF employ a simple static policy of requester-wins conflict resolution. Nevertheless, in earlier work [3] we have discussed that conflicts in a simple HTM implementation on top of an out-of-order microprocessor can abort due to conflicts on data on mis-speculated branches, both in the read and write sets.

Without a clear specification, resulting performance may be difficult to predict, particularly if optimisation with non-transactional memory is employed. Even more importantly, without a specification, implementations may even be free to treat non-transactional accesses as transactional. Apart from the obvious performance implications, how much does software rely on this?

For ASF, we have designed a minimal capacity for progress in the absence of actual contention, which ensures that the implementation must eventually execute sufficiently small transactions successfully.

3 Conclusion

Hardware transactional memory implementations deliberately weaken the atomicity property associated with transactions. We have illustrated several aspects of this weakening and why it may actually be beneficial to have in some cases, and provided small examples that depend on the actual way of weakening.

We have shown how we think these ambiguities should be solved; namely, in the ASF specification. However, we believe open areas of what we have specified remain, and open areas may remain in how we have specified certain aspects. We would therefore like to use the workshop as a platform for discussion and to learn about open issues and how other practitioners feel about out attempts.

We would also like to learn about work in deriving a more formal specification of TM.

References

[1] Advanced Micro Devices, Inc. *Advanced Synchronization Facility - Proposed Architectural Specification*, 2.1 edition, March 2009.

[2] David Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS*, 2009.

[3] Stephan Diestelhorst, Martin Pohlack, Michael Hohmuth, Dave Christie, Jae-Woong Chung, and Luke Yen. Implementing AMD's advanced synchronization facility in an out-of-order x86 core. In *TRANSACT 2010*, April 2010.

Safely Accessing Time Stamps in Transactions

Stephan Diestelhorst
stephan.diestelhorst@amd.com
Advanced Micro Devices, Inc.

Martin Pohlack
martin.pohlack@amd.com
Advanced Micro Devices, Inc.

Abstract

Time stamps are frequently used in multi-threaded applications, and provide a way for an application to determine order between events. We identify interactions between time stamps and transactional mechanisms that differ from the expected behaviour of using locks for mutual exclusion, and draft implementations that remove these differences.

Keywords: Transactional Memory, Lock Elision, Clocks, Synchronization

1 Roadmap

Section 2 provides the background of concurrency control mechanisms and time stamps, and Section 3 reviews their semantics. We then show critical interactions between time stamps and transactions in Section 4, and draft two implementations that properly handle these in Section 5. Sections 6 and 7 provide an outlook and conclude the paper, respectively.

2 Background

2.1 Concurrency Control

Critical sections traditionally operate on a strict mutual exclusion property: Instructions of two critical sections must not interleave if both critical sections are protected by the same lock variable. Recent literature calls this mode of operation Single Lock Atomicity (SLA) [6].

Database transactions provide *serialisability* [8], the strongest form of isolation. Briefly, transactions may overlap if there exists an equivalent execution in which no transactions overlap. *Strict* serialisability (and the similar *linearisability* [4]) is a stronger form of serialisability that restricts the serial order such that observed real-time order between non-overlapping transactions is maintained.

Transactional memory [3] brings the notion of transactions to general-purpose systems. The large variety of semantics [7] differs mainly in how they deal with interleavings of instructions that are not part of transactions and instructions that are part of a transaction.

2.2 Transactional Lock Elision

Transactional execution offers benefits over strict mutual exclusion imposed by critical sections, because transactions can be optimistically executed in parallel, as long as the specified isolation level is maintained. The idea to convert lock-protected critical sections into transactions and extract additional performance has been proposed in previous work on lock elision [10].

```
        Initially: c = 0
t1 = RDTSCP
c := 1
                  lc = c
                  t2 = RDTSCP
```

Figure 1: Dependent reads from the time stamp counter produce properly ordered time stamps: If $lc = 1$, then $t2 > t1$.

To *transparently* elide locks in existing applications, the semantics must not be weaker than that provided by mutual exclusion.

2.3 Access to Time Stamp Counters

Applications must measure time and its progression, to determine durations of, order and coordinate events. Computers provide time sources of varying quality. We will focus our analysis mainly on the CPU's time stamp counter (TSC), which has seen significant quality improvements during the last decade. We assume the TSC is a suitable real-time source,[1] and that applications (through libraries such as libc) rely on such behaviour. For brevity, and without loss of generality, we ignore the potential offset between different TSCs because it can be bounded by a small constant ϵ with initial calibration. Our results hold also in those cases, with some additional margins added to comparisons accounting for the difference.

On the AMD64 architecture, the TSC can be read with the `RDTSC` and `RDTSCP` instructions. We consider only `RDTSCP` because it is properly serialised with the instruction stream. We do not consider other clocks, due to space and because the TSC is the most frequently used fast, stable time source.

3 Semantics of Time Stamps

3.1 Traditional Code

Memory causality and synchronised time stamps allow us to reason about time stamp relations across multiple cores and application threads:

Causal TSCs: In non-transactional code, order imposed by memory accesses will be reflected in time stamps that are ordered accordingly.

In Figure 1, two `RDTSCP` instructions are ordered by a memory dependence. We assume for this and similar examples $t_2 > t_1$ always holds; generally, existing order such as memory dependence and program order also orders time stamps.

Critical sections implemented through locks serialise execution through memory dependencies. Therefore, for

[1]This has been true for most x86 microprocessors since 2007.

time stamps t_1 and t_2 read in inside critical section CS_1 with $t_1 < t_2$, and t_3, t_4 read in CS_2 also ordered $t_3 < t_4$, and CS_1 and CS_2 being protected by the same lock variable, we know from *Causal TSCs* that either $t_2 < t_3$, or $t_4 < t_1$. In other words:

Temporal mutual exclusion: For two critical sections CS_1, CS_2 protected by the same lock, the intervals spanned by the set of obtained time stamps $T(CS)$ do not overlap: $[min(T(CS_1)), max(T(CS_1))] \cap [min(T(CS_2)), max(T(CS_2))] = \emptyset$.

3.2 TSC-oblivious Transactions

Because transactions are a new programming construct, they need not maintain strict compatibility with legacy code and its assumptions. Therefore, access to the TSC is treated differently in the various implementations of transactional memory: software TMs (STMs) usually do not track `RDTSC(P)` instructions and so may allow an application to infer temporal placement and overlap of the transactions. AMD's proposed Advanced Synchronization Facility (ASF) [1] does not allow transactions to execute `RDTSC(P)`, making it impossible to detect temporal overlap for transactions at the cost of not being able to read the TSC in transactions.

In Intel's recent Transactional Synchronization Extensions (TSX) [5], access to the TSC is permitted in both transactions and elided critical sections. Because the elision does not work fully transparently,[2] software can be exposed to the changed semantics of (elided) critical sections [9].

3.3 The Need for Stronger Semantics

Usually, elided critical sections track memory accesses and ensure they correlate to a sequential execution by tracking conflicting accesses. However, as we will show in the next section, this is insufficient if applications make use of the TSC inside critical sections. Applications may infer concurrent execution of critical sections that were supposed to execute strictly sequentially. Such mismatch between expected behaviour and implementation breaks transparency of the elision and can lead to crashes, or other misbehaviour.

In addition to supporting stronger semantics for the lock elision case, we also consider a stronger semantics incorporating TSC accesses for (hardware) transactions. Providing the same semantics in both modes makes sense: (1) using hardware transactions as a drop in replacement for locking, and (2) providing a well-known semantics for TSC usage in transactions that is easy to understand and reason about.

4 Semantic Issues of Time Stamps in Transactions

This section illustrates problematic orderings of transactions and the use of time stamp accesses from within. We aim for transactions and elided critical sections to have a semantics that is equivalent to proper mutual exclusion.

[2]Applications need to use annotated instructions to acquire and release the lock variable, and can query whether they run in a transaction / elided critical section with the `XTEST` instruction

```
TX1.begin          TX2.begin
t1 = RDTSCP
                   t = RDTSCP
t2 = RDTSCP
TX1.end            ...
                   TX2.end
```

Figure 2: Overlapping time stamp values violate SLA.

```
        Initially: c = 0

TX1.begin
                   TX2.begin
                   t2 = RDTSCP
c := 1
t1 = RDTSCP
TX1.end
                   lc = c
                   TX2.end
```

Figure 3: Two transactions can exhibit contradicting ordering if $lc = 1$ and $t_2 < t_1$.

We will therefore use the term transactions also for elided critical sections.

4.1 Simple Overlap Case

In Figure 2, the case $t_1 < t < t_2$ directly violates *Temporal mutual exclusion* – the result for time stamps in traditional critical sections obtained in Section 3.1.

4.2 Order Mismatch: Memory vs. Time Stamps

In addition to detecting temporal overlap through time stamps, two transactions may observe mismatches in the respective memory or time stamp orders. Figure 3 shows an execution that orders transaction `TX1` before `TX2` through memory; however, because $t_2 < t_1$, the time stamps indicate the opposite. Proper SLA semantics would not allow such contradicting orders under *Causal TSCs*.

4.3 Weak and Strong Temporal Isolation

If code accesses the same data inside and outside critical sections, no new order between these accesses is created. In those cases, order can be created only through reasoning with the underlying memory semantics. If no such order can be established (usually called a data race), the involved accesses are subject to complex, sometimes even all-bets-are-off / catch-fire semantics.

With transactional memory, however, some systems provide *strong* isolation which properly orders transactions with respect to memory accesses outside of transactions, usually by assuming that the non-transactions are one-instruction mini-transactions. Systems that do not provide such isolation, but only order transactions, provide *weak* isolation.

We observe a similar interaction with TSC accesses, extending the weak / strong isolation property to weak / strong *temporal* isolation (WTI / STI).

Figure 4 highlights the interaction. The example clearly does not violate either SLA, nor does it produce overlapping time stamp intervals. It also does not violate purely memory-based strong isolation semantics, because

```
          Initially: c = 0

                    TX2.begin
                    t2 = RDTSCP
t1 = RDTSCP
c := 1
                    lc = c
                    TX2.end
```

Figure 4: Failure of strong temporal isolation if $lc = 1$ and $t_2 < t_1$. Instructions on the left do not execute from a transactional context.

```
          Initially: c = 0

                    TX2.begin
                    t2 = RDTSCP
t1 = RDTSCP
TX1.begin
TX1.end
c := 1
                    lc = c
                    TX2.end
```

Figure 5: Transparent lock elision requires strong temporal isolation: with SLA, if $t_2 < t_1$ then $lc = 0$; however, with weak temporal isolation $lc = 1$ is possible.

TX2 is ordered behind the non-transactional store from a memory perspective. With strong temporal isolation, the intuitive combination of strong memory isolation and *Causal TSCs* is enforced.

4.4 The Need for Strong Temporal Isolation

Strong temporal isolation restricts execution more than a system that provides only weak temporal isolation. However, SLA does not inherently order critical sections and code outside of critical sections, so it may seem sufficient to provide only weak temporal isolation for transparent lock elision (and transactions). However, if we modify the example in Figure 4 slightly to obtain Figure 5, we find otherwise. Using locks (SLA semantics), we can deduce the following: If $t_2 < t_1$, TX2 must also read the old value of c because it must have executed entirely before the empty transaction TX1 which in turn executed before the update to c. Strong temporal isolation will enforce the same implication to hold, but the depicted schedule is possible with weak temporal isolation. Therefore we conclude:

WTI insufficient: Transparent transactional lock elision with support for accesses to TSCs requires a stronger semantics than weak temporal isolation.[3]

5 Supporting Time Stamps in Transactions

In the previous section, we presented various problems that illustrated how existing systems with unrestricted access to time stamps can cause violations of the semantics outlined in Section 3.

Of course, forbidding all access to time stamp counters from within transactions and elided critical sections will provide the exact semantics. ASF works like that and aborts all executed transactions that try to use the RDTSC(P) instruction. However, because time stamping is employed in many applications, it is desirable to find less rigid solutions that provide temporal isolation.

Temporal Isolation Rules: Clearly, there are two rules needed for proper temporal isolation: (1) The time stamp order needs to agree with the order established by the memory accesses in transactions and non-transactional code; and, (2) time stamp intervals of transactions / elided critical sections must not overlap.

5.1 Single TSC Access

Restricting transactions to access the TSC at most once removes the problem of overlapping time stamp intervals. A simple implementation could therefore track locally that transactions do not read from the TSC twice, and abort otherwise.

The simplest solution to enforce consistent time stamp and memory order, *Single-AbortAll*, is to allow only a single transaction access to the TSC and abort all other overlapping transactions (e.g., by adding a dedicated memory location TSC_A to each transaction's read set and sending out conflicting write probes on an RDTSC(P)). For example, the execution of Figure 3 would abort TX1 immediately at the acquisition of t_2, regardless of TX1's content.

A more selective approach is *Single-AbortTSC*, which aborts only those live transactions that have or will eventually access the TSC. That can be achieved by recording TSC usage, handling the conflict only if a running transaction has already accessed the TSC, and letting future RDTSC instructions check whether their enclosing transactions have seen a remote TSC access. Compared to Single-AbortAll, the advantage is that transactions that do not access the TSC need not be aborted. The execution in Figure 3 would only need conflict handling when TX1 obtains t_1. If it would not have used RDTSCP, it could continue execution.

5.2 Multiple TSC Accesses

Allowing a transaction to read from the TSC multiple times allows transactions to observe the passage of time. Similar to Single-AbortTSC, transactions will not abort all other concurrently running transactions at the first read of the TSC, but rather ensure that no overlapping time stamp intervals can form.

In this case, each transaction checks at the second or later read from the TSC that it has not received any notifications from remote TSC accesses; otherwise, a conflict exists. This case needs to be handled by a conflict resolution policy, for example by self-aborting on the second local RDTSC(P), or by aborting all other time stamp-using transactions.

The advantage of this technique is that transactions can execute in parallel, and each can access the TSC multiple times, as long as the intervals formed by each transaction's first and last accesses to the TSC are disjunct. In the Figure 2 example, the conflict would be detected at the read of t_2 because of the concurrent TSC read to t . If the reads for t_2 and t would have been ordered the opposite way (and then also $t_2 < t$), no conflict would exist.

[3]Strong temporal isolation, for example. We do not know if a semantics weaker than STI would also work.

Enforcing order: Again, in addition to enforcing disjunct time stamp intervals, we need to make sure that these intervals are ordered consistently with all other orders – for example, imposed through memory accesses. Take the simple example in Figure 3: so far, the algorithm does permit the contradicting order. Consistent ordering betwen time stamps and memory accesses can be achieved by ensuring that `TX2` commits before `TX1`, for example by waiting at `TX1`'s commit point for a *can_commit* message from `TX2`. These messages do not need to broadcast, because `TX2` knows that `TX1` has acquired a later time stamp. `TX2` can therefore signal `TX1` (and all later) transactions that they must wait for a commit signal, and when they can commit.

Waiting alternatives: Instead of `TX1` stalling the CPU at commit, it may be possible for it to continue execution of instructions behind the transaction transactionally, essentially increasing the length of the transaction by adding the following, non-transactional instructions to its transactional tail. That way, useful work can be done, reducing the performance penalty incurred by waiting. Of course, this may lead to additional conflicts (due to additional memory accesses adding to the working set), and additional transactional TSC accesses that need special handling.

Another option is to put the core in a low-power state and wait for the special *can_commit* signal from `TX2` (similar to the low-power mode that can be entered with the `MONITOR` / `MWAIT` instruction combination [2]).

5.3 Handling Non-transactional TSC Accesses

STI requires that TSC accesses outside transactions need to participate in the conflict detection mechanisms. However, we may want to treat conflicts with non-transactional TSC accesses differently to ensure progress and minimal obstruction for non-transactional code. We therefore suggest biasing conflict resolution in favour of non-transactional TSC accesses, by always aborting the concurrent transactions with which the non-transactional `RDTSC(P)` conflicted (instead of retrying / delaying the non-transactional access).

6 Outlook

Although we believe our examples and implementations cover all critical interactions, and thus provide an identical semantics to fully sequential mutual exclusion, we have not yet proven our solutions to be correct or our restrictions to be minimal. We have started to develop a formalism, but have not discussed it here due to space constraints and prematurity.

We are particularly interested in the applicability of our proposed implementations. Clearly, our system is more permissive than AMD ASF's strict ban of TSC reads inside transactions (and more transparent than Intel's TSX lock elision), but we still need to abort and serialise a significant number of transactions. Although we believe that this is inherent to the limit to which one can make parallel execution behave similarly to sequential execution, some applications may just not care. Automatic inference of such properties seems very hard; therefore, we have restricted our analysis to provide as much performance as possible with STI. The straightforward way out (for us as hardware designers) is to leave the decision to software – for example, by offering an explicit choice between strong and weak temporal semantics with multiple types of transactions and / or time stamp accesses. This new freedom opens up a new space for looking into the interactions between different types and exactly how much each needs to be weakened to provide compelling performance, while remaining useful for applications.

7 Conclusion

In addition to communication through memory, time stamps provide another way for parallel code to coordinate. Therefore, time stamps need to be considered by systems that control parallel execution such as transactional memory and lock elision. Existing critical sections provide very strong semantics of strong temporal isolation, which we have not found to be provided by most STM and HTM solutions. In this paper, we have identified the issue of time stamp order and have shown with multiple examples how applications can observe time stamps inconsistent with the isolation level or with the order observed from memory accesses. We drafted various implementations that will provide strong temporal isolation without the need to fully serialise execution.

References

[1] Advanced Micro Devices, Inc. AMD Advanced Synchronization Facility Proposal. Available at developer.amd.com/CPU/ASF/Pages/default.aspx.

[2] Advanced Micro Devices, Inc. *AMD64 Architecture Programmers Manual Volume 3: General-Purpose and System Instructions*, 3.18 edition, March 2012.

[3] Maurice P. Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, Calif., May 1993.

[4] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[5] Intel Corp. *Intel(R) Architecture Instruction Set Extensions Programming Reference*, 319433-012a edition, February 2012.

[6] James R. Larus and Ravi Rajwar. *Transactional Memory*. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2007.

[7] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard Hudson, Bratin Saha, and Adam Welc. Practical Weak-Atomicity Semantics for Java STM. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[8] Christos H. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the JACM*, 26(4):631–653, 1979.

[9] Ravi Rajwar. Personal communication, 2012.

[10] Ravi Rajwar and James R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34th IEEE/ACM International Symposium on Microarchitecture*, Austin, Tex., December 2001.

Between All and Nothing–Versatile Aborts in Hardware Transactional Memory

Stephan Diestelhorst
ARM Ltd., Cambridge, UK
TU Dresden, Germany
stephan.diestelhorst@gmail.com

Martin Nowack
Christof Fetzer
TU Dresden, Germany
martin,christof@se.inf.tu-dresden.de

Michael Spear
Lehigh University, USA
spear@cse.lehigh.edu

Abstract

Hardware Transactional Memory (HTM) implementations are becoming available in commercial, off-the-shelf components. While generally comparable, some implementations deviate from the strict all-or-nothing property of pure Transactional Memory. Instead of trying to hide them, we lift these deviations to a simple *transactional resurrection* mechanism that can be used to accelerate and simplify both transactional and non-transactional programming constructs. We implement our modifications both architecturally and micro-architecturally in a detailed HTM proposal, without changes to system software and only light modifications to the existing HTM microarchitecture. We then show application of transactional resurrection in both transactional and non-transactional parallel programming: hybrid transactional memory; transactional escape actions; alert-on-update; and transactional suspend / resume.

Categories and Subject Descriptors C.1.2 [*Computer Systems Organization*]: Processor Architectures—Multiprocessors; D.1.3 [*Software*]: Programming Techniques—Parallel Programming

Keywords computer architecture, synchronisation, transactional memory, cross thread communication

1. Introduction

Originally proposed in 1993, Hardware Transactional Memory[11] (HTM) has at last gained traction with industry, and leading microprocessors have incorporated HTM support [13–15]. However, these products provide a much less exotic flavor of HTM than those proposed by researchers [26, 27]. They generally offer a comparable best-effort HTM with strong isolation, but very loose capacity specifications, as capacity is usually determined by size and organisation of the cache used to track the transactional working set.

Clearly, there is a gap between what the hardware provides now and in the near future,[1] and the compelling features suggested in academia. We show how to extend a basic HTM proposal to bridge this gap and bring features proposed in academia to product-grade HTM proposals.

[1] IBM POWER8 2014 [17], IBM zEC12 [2], Intel Haswell [39]

TRANSACT '15, June 15–16, 2015, Portland, OR, USA.
Copyright is held by the owner/author(s).
ACM X-XXXXX-XX-X/XX/XX.
http://dx.doi.org/10.1145/nnnnnn.nnnnnn

Even though the various HTM proposals and forthcoming products have many similarities in their core feature set, on the periphery the proposals differ, for example how they treat the register state, and in the availability and design of mechanisms that allow code to escape through the transactional layer. Comparing, for example, Intel's Transactional Synchronization Extension (TSX) and AMD's Advanced Synchronization Facility (ASF), both provide best-effort transactional memory (Restricted Transactional Memory (RTM) in Intel's proposal), but differ in (1) the way they treat the snapshot / rollback of a transaction's register state, (2) non-transactional accesses from within a transaction, and (3) the availability of a minimum capacity guarantee.

TSX snapshots all registers on transaction start, and restores them automatically on abort; it also does not provide instructions to bypass the transactional mechanisms (e.g., loads within a transaction that are not tracked, or stores within a transaction that do not roll back). ASF provides the opposite: registers are not automatically saved and restored, but instead software needs to manually save live registers on transaction start and restore them on abort. Additionally, ASF allows programs to bypass the transactional mechanisms through the application of an existing instruction prefix to mark memory operations as *non-transactional*; these operations will appear to take effect immediately, rather than at the end of the transaction. A similar feature was present in the cancelled Rock processor [4].

We explore the different policies for register snapshotting and propose *transactional resurrection* as a lightweight mechanism which we use to synthesise features such as alert-on-update[34], escape actions[22] and transactional suspend / resume; thus achieving a rich transactional programming environment.

We focus on extending the HTM interface, but are careful not to increase hardware verification costs or require changes to existing system software. In particular, we do not extend the architected state of applications, and thus the operating system and hypervisor can remain oblivious of the extensions, e.g., when performing context switches. Our hardware modifications are non-invasive in nature and do not require any additional associative tracking structures, or other deep changes to the processor pipeline or the cache coherence / memory subsystem.

Our contributions in this paper are: we propose the mechanism of transactional resurrection that allows aborted hardware transactions to resume; we implement these mechanisms as four new instructions in a detailed architectural and micro-architectural HTM prototype (ASF); on top of these, we build transaction suspend / resume, escape actions, and multi-location alert-on-update. Finally, we evaluate performance overheads and demonstrate the functionality of our implementation in a full-system, cycle-level simulator.

The paper is structured as follows: first, we give a brief introduction to ASF in (Section 2) and detail our proposed hardware extensions (Section 3). We present the different higher level use cases for the extensions (Section 4,5,6) and conclude with an evaluation (Section 7) and related work (Section 8).

2. Background: ASF

For our design we extend AMD's Advanced Synchronization Facility (ASF) [6]. In this section we briefly review how ASF exposes core HTM functionality, as well as the unique aspects of ASF that we use to build a more robust programming environment.

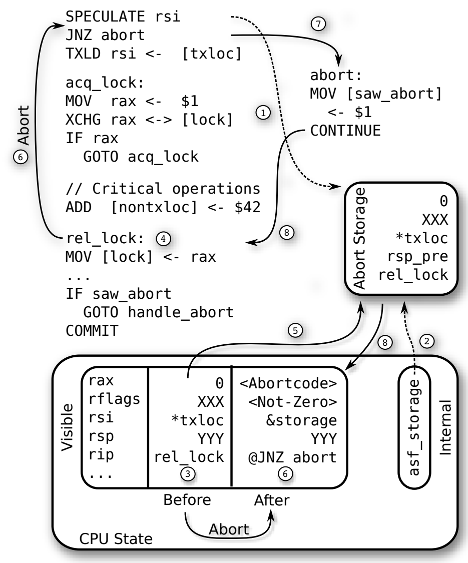
ASF transactions are started with the SPECULATE instruction which creates a partial checkpoint of the thread state. SPECULATE also serves as the entry to an abort handler if a transaction fails to commit. The COMMIT instruction ends a transaction, making all transactional updates immediately and atomically visible to memory. Within a transaction, regular x86 MOV instructions and prefixed LOCK MOV instructions (which can be either loads or stores) are used to distinguish between immediate, irrevocable accesses that escape the transaction and transactional accesses (i.e., stores are buffered until commit, and loads are tracked in the cache). The polarity of the LOCK prefix is determined by selecting either SPECULATE or SPECULATE_INV to start the transaction. The former executes undecorated accesses non-transactionally and uses prefixes to mark transactional accesses, while the latter inverts the scheme and is more similar to TSX, Rock and other HTM proposals. In the remainder of this paper we will explicitly state the transactional property of accesses.

ASF provides strong isolation: transactions will detect conflicts with concurrent accesses, even when those concurrent accesses occur outside of a transaction. Conflicts are resolved through a simple requester-wins abort policy which always aborts the transaction that added the conflicting item to its working set first. When percore private data caches are used to detect conflicts, this policy can be supported without any change to the underlying cache coherence protocol, thereby reducing the verification cost of transactional extensions to the ISA.

In case of an abort, ASF will undo any speculative memory writes, but will keep the processor registers and all other memory updates visible. The CPU redirects execution to the instruction following SPECULATE and provides an error code (in register rax) with information about the abort reason. The application should check the error code and branch to an abort handler that will take appropriate measures (e.g., back-off and restart the transaction). Aborts in ASF happen synchronously with the condition for the abort, and may occur between any two instructions in the transaction. As a "best effort" HTM implementation, additional causes of aborts include, but are not limited to, system calls, exceptions and interrupts (to include timer interrupts), and capacity/conflict cache evictions (i.e., due to the transaction's working set exceeding the size of the cache).

3. Resurrection–Aborts with Continuation

When an ASF transaction aborts, almost the entire register state is available to the abort handler. The only exceptions are the registers used to: convey the abort cause (rax, rflags); restore the stack pointer (rsp); and change the control flow to the instruction after the SPECULATE instruction (rip). If the values of these registers were made available, the abort handler could resume execution inside the transaction (ignoring for now that the abort would clear the transaction's working set).



```
SPECULATE rsi
JNZ abort
TXLD rsi <-  [txloc]

acq_lock:
MOV  rax <-  $1
XCHG rax <-> [lock]
IF rax
  GOTO acq_lock

// Critical operations
ADD  [nontxloc] <- $42

rel_lock:
MOV [lock] <- rax
...
IF saw_abort
  GOTO handle_abort
COMMIT
```

```
abort:
MOV [saw_abort]
  <- $1
CONTINUE
```

Figure 1. Basic Functionality of Abort with Continuation

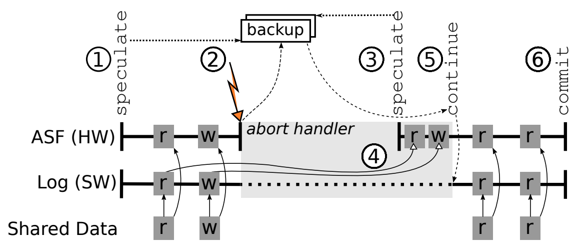


Figure 2. Suspend/Resume mechanism: (1) A hardware transaction is started (SPECULATE with "backup" as argument), but the transaction body is instrumented so that accesses will also be logged; (2) In case of an abort, ASF records the instruction pointer rax and executes the abort handler; (3) If the transaction can be recovered, the handler starts a new hardware transaction; (4) The working set is replayed; (5) The transaction resumes the normal execution (using CONTINUE); (6) The transaction commits its hardware transaction and resets its logs.

One option to place these register values is to use additional, new registers for either storing the old content of the overwritten registers, or conveying the necessary additional abort information and not use existing registers for that purpose. Both variants do increase the footprint of the architectural register state of applications. Therefore, operating systems and hypervisors would then have to be aware of these registers and save / restore them on context switches. To avoid affecting systems software, the register state must go elsewhere: we let the programmer allocate a buffer to hold the old values of the overwritten registers and provide the location of the buffer as a parameter to an extended SPECULATE instruction.

In Figure 1 we present the main interaction: SPECULATE is extended so that it accepts a memory buffer location parameter (1), looks up the virtual address and translates it to a physical address, and also checks write permissions to the location. Any page faults that could occur when accessing the buffer are thus already resolved before the transaction starts. In the event of nested transactions, the SPECULATE instruction ignores this parameter: Since ASF only supports flat/subsumption nesting, there is no meaning or benefit to saving multiple register checkpoints.

The CPU keeps the resulting physical address in an internal register (2) and starts the transaction. The transaction executes, mutating the CPU's register state (3). In case of an abort (4), the processor first copies `rax`, `rip`, `rsi`, and `rflags` into the application-provided buffer (5), and then updates these registers and control flow to reflect the abort condition (6), with `rsi` additionally holding the buffer address. Furthermore, `rsp` will no longer be restored:[2] this prevents stack smashing due to signals or interrupt handlers running within the abort handler. The application code checks for aborts and branches to an abort handler (7). The abort handler can simply restore `rsi` and `rsp` from the the buffer pointed to by `rsi` and reproduce the original ASF abort functionality. However, it can also resume the code in the transaction by restoring all overwritten registers from the buffer (8). Since existing assembly primitives cannot restore all registers without overwriting an additional temporary register, we provide a new `CONTINUE` instruction that performs a simple micro-code sequence to restore the registers.

To simplify the ASF interface, we also provide `RDINVMODE` and `WRINVMODE` instructions. These allow the programmer to detect and change the behaviour of `MOV` and `LOCK MOV` within transactions.

3.1 Handling synchronous aborts asynchronously

ASF transactions are aborted immediately in case there is a reason for it (e.g. contention). This provides strong isolation guarantees for transactions but puts the burden on a programmer to reason about correctness, as a transaction might be aborted at every machine level instruction. Especially, if non-transactional modifications are made inside a transaction (e.g. memory allocation), a correct state has to be preserved. With the proposed extensions, it is easy to translate HTM's synchronous aborts into asynchronous aborts. The abort handler will simply set a thread local variable (`saw_abort` in the example) signalling an abort, and will then execute `CONTINUE`. The code inside the transaction will query the variable at suitable intervals and then can handle the earlier abort asynchronously.

3.2 Cost

From a hardware perspective, the required changes atop ASF are minimal: memorising an additional pointer during the execution of the transaction is easily achieved in either an internal register or in scratchpad memory. The changes to `SPECULATE`, aborts and the new `CONTINUE` instruction can be effectively coded in microcode. Pre-checking the allocated buffer location for a proper virtual to physical address mapping when executing `SPECULATE` ensures that the processor will always be able to store the continuation information and no abort page fault deadlock can occur.[3]

Implementing our changes on top of other industry HTM proposals, such as Intel TSX, requires slightly more effort. Our additions to the abort handler and new instructions remain lightweight, but non-transactional accesses are usually absent from the currently specified proposals. Discussion with hardware designers showed, however, that supporting non-transactional accesses is neither overly complex nor requires a lot of silicon, but instead has been postponed due to lack of demand and semantic corner cases. We show how non-transactional accesses can be used in a beneficial, controlled, and semantically clear manner.

Best-effort TMs, such as ASF, usually do not virtualise the transactional resources on context switches; instead, they abort ongoing transactions. In our design we do the same, thereby avoiding resource-hungry virtualisation. Note that the full architectural CPU state can be reconstructed by the abort handler even after a context switch: saving the registers clobbered by the hardware abort in virtual memory lets them survive the context switch, and the OS's existing context save / restore mechanism naturally takes care of all other registers.

```
procedure SAHTM_START                    ▷ Start a software-assisted HTM transaction
    $buf_1, buf_2 \leftarrow$ malloc()
    $(buf_1.other, buf_2.other) \leftarrow (buf_2, buf_1)$    ▷ Cross-link $buf_1$ and $buf_2$
    $log \leftarrow ()$
    SPECULATE $buf_1$                    ▷ Start the hardware transaction
    $error \leftarrow$ rax
    if $error \neq 0$ then
        return SAHTM_ABORT(rsi, $error$)    ▷ Handle errors and resume if possible
    return $IN\_TX$

procedure SAHTM_ABORT($s, error$)    ▷ Handle aborts, $s$ holds the resume state
    push regs
    retry:
    if $error.type = CONTENTION$ then
        rsp $\leftarrow s.abort\_rsp$                ▷ Do a full abort
        return $ABORTED$
    else if $error.type = FAR$ then    ▷ Resurrect after interrupts, page-faults
        $(error, s) \leftarrow$ SAHTM_RESURRECT($s$)    ▷ Successful resurrection does not return
        goto retry
    else if $error.type = ILLEGAL$ then  ▷ Emulate syscalls etc.
        $s \leftarrow$ EMULATE($s$)
        $(error, s) \leftarrow$ SAHTM_RESURRECT($s$)    ▷ Resurrect after successful emulation
        goto retry
    else
        ...                                ▷ Handle other abort reasons

procedure SAHTM_COMMIT
    COMMIT                    ▷ Just commit the HTM transaction
    free($buf_1, buf_2$)
```

Figure 3. Handling the life-cycle of HTM transactions with suspend / resume extensions. Applicable to both software-assisted and hardware-extended suspend / resume HTM.

4. OS-transparent Transaction Suspend / Resume

Suspend / resume appears in the new IBM POWER8 HTM proposal [13], but relies on additional registers and special handling in the OS when dealing with suspended transactions. Suspending a transaction is useful for tolerating short execution of other code, for example dealing with hardware interrupts, syscalls or exceptions. We now show how to enable full transaction suspension and resume in ASF without changes to the OS, by building upon the simple extensions we proposed in Section 3.

Our general approach is to let the transaction abort instead of suspending it, and then offer a mechanism to *resurrect* the aborted transaction when resumption is necessary. From an architectural perspective, this means that suspended transactions are the same as aborted transactions, and thus do not require special treatment by the OS. Transaction-aware OS can use transactions without concerns for the application's usage of the transactional memory resources and legacy OS can handle applications using HTM. Our extended abort mechanism allows full access to all registers in the

[2] The old value of `rsp` from `SPECULATE` is stored in the buffer, instead.

[3] Since page faults cause aborts, we would otherwise risk deadlock from the following cycle: abort transaction $\rightarrow$ store to buffer to keep overwritten register values $\rightarrow$ page fault due to buffer access $\rightarrow$ abort transaction because of a page fault

```
28: procedure SAHTM_RESURRECT($s$)    ▷ $s$ holds the resume state
29:     if $s.resume\_ip \in$ lines(31 – 36, 41 – 48) then ▷ Aborted while resuming
30:         $s \leftarrow s.other$    ▷ Squash abort recursion
31:     SPECULATE $s.other$    ▷ Start HTM container transaction
32:     $error \leftarrow$ rax
33:     if $error = 0$ then    ▷ Successful start of HTM container
34:         SAHTM_REPLAY()  ▷ Replay transactional working set from SW log
35:         pop regs
36:         CONTINUE $s$    ▷ Restore full state and return to resurrected transaction.
37:     else    ▷ Abort in the resurrected transaction
38:         $s \leftarrow$ rsi    ▷ Update resumed state from new abort site
39:         $s.abort\_rsp \leftarrow s.other.abort\_rsp$
40:         return $(error, s)$ ▷ Outer logic handles abort condition and retries

41: procedure SAHTM_REPLAY    ▷ Replay and validate transactional accesses
42:     for $(addr, val, rw) \in log$ do    ▷ from the SW log
43:         if $rw = READ$ then
44:             txload $tmp \leftarrow [addr]$    ▷ Add to read set
45:             if $val \neq tmp$ then    ▷ and validate value
46:                 ABORT $CONTENTION$    ▷ Use HTM abort to unravel validation failure
47:         else
48:             txstore $[addr] \leftarrow val$    ▷ Redo stores

49: procedure SAHTM_TXLOAD($addr$) ▷ Software-assisted read barrier
50:     $log \leftarrow (log, (addr, \emptyset, READ))$    ▷ Append a sentinel protecting against an abort
51:                 ▷ between line 52 and 53 missing replay of $addr$
52:     txload $val \leftarrow [addr]$    ▷ Add to HTM read set
53:     $(log, (addr, \emptyset, READ)) \leftarrow (log, (addr, val, READ))$    ▷ Update with proper read value
54:     return $val$

55: procedure SAHTM_TXSTORE($addr, val$)  ▷ Software-assisted write barrier
56:     $log \leftarrow (log, (addr, val, WRITE))$    ▷ Append to log
57:     txstore $[addr] \leftarrow val$    ▷ Add to HTM write set
```

Figure 4. Resurrection and replay of aborted transactions, logging read / write barriers. For simplicity, we omit handling different sizes in SAHTM_TXSTORE and SAHTM_TXLOAD.

abort handler, so that it can restore the transaction's register state exactly as it was at the time of the suspension (abort).

In Figure 2, we depict the time line of a suspend / resume cycle. Resume / resurrection is initiated in the abort handler when the suspend condition has been handled, for example when control has passed back to the application after invocation of an hardware interrupt handler. A new transaction is started with SPECULATE with a new buffer for storing the abort state and CONTINUE then restores the suspended transaction's register state and resurrects it. Figure 3 shows the general behaviour in pseudo-code.

Clearly, making available all register state of the transaction to the abort handler is not enough to resurrect the aborted transaction, because the transactional working set in memory is rolled back in ASF upon any abort. We present two options to deal with resurrecting the transactional working set: (1) tentatively keeping transactional state across aborts in hardware, or (2) adding minimal logging instrumentation in software.

Inspecting the different abort reasons, we find that not all of them require immediate roll-back of the working set. In particular, for aborts not caused by violations of the integrity of the working set (i.e., aborts other than (certain types of) contention

```
1: procedure MESSYHTM_RESURRECT    ▷ Resurrect with additional HW support
2:     if $s.ip \in$ lines(4 – 8) then
3:         $s \leftarrow s.other$    ▷ Squash abort recursion.
4:     SPEC_RESURRECT $s.other$    ▷ Start HTM transaction using working set still in cache
5:     $error \leftarrow$ rax
6:     if $error = 0$ then    ▷ Successful resurrection
7:         pop regs    ▷ No need to restore the read / write set
8:         CONTINUE $s$    ▷ Restore full state and return to resurrected transaction.
9:     else    ▷ Abort in the resurrected transaction
10:        $s \leftarrow$ rsi    ▷ Update resumed state from new abort site
11:        $s.abort\_rsp \leftarrow s.other.abort\_rsp$
12:        return $(error, s)$ ▷ Outer logic handles abort condition and retries

13: procedure HTM_TXLOAD($addr$)    ▷ Read barrier for resurrection with HW support
14:     txload $val \leftarrow [addr]$
15:     return $val$

16: procedure HTM_TXSTORE($addr, val$)    ▷ Write barrier for resurrection with HW support
17:     txstore $[addr] \leftarrow val$
```

Figure 5. Hardware support in the caches to tentatively keep the aborted transaction's working set significantly simplifies the resurrection logic.

or capacity evictions), it may be possible to keep the transactional state tentatively in the cache and make it available to the resurrecting SPECULATE / CONTINUE pair. We propose *Messy-HTM* that extends the SPECULATE instruction so that the cache can distinguish between a request to start with a clean transactional state (SPECULATE) and attempts to reactivate the old transactional state and aborts when this fails (SPEC_RESURRECT). To keep the OS unmodified, the speculative state is cleared when the processor sees an event that causes a TLB flush, usually indicating a context switch. Nevertheless, the proposed suspension / resurrection mechanism can tolerate brief kernel invocations, for example due to interrupts or system calls from within the transaction.

If lazy clearing of transactional state proves too complex for an HTM implementation, or if support for surviving full context switches is desired, we can employ a lightweight hybrid TM approach of *Software-assisted HTM* (SAHTM): transactional accesses can be manually logged in a thread local buffer, which is used to validate and replay the hardware transaction upon resurrection.[4] We store in the buffer transactionally read and written values, the associated addresses, access sizes and access types (read / write). The buffer is updated during transactional execution by using *non-transactional* stores. Since the HTM is used to provide proper conflict detection and versioning, the log is append-only and never read during normal operation: buffered updates that are stored in the log are also performed as part of the transaction, and loads that are tracked in the log need not be validated, since these loads are also part of the transaction's working set. To reduce the overhead of logging, we combine the address and meta-data into a single 64 bit word (note that virtual addresses are only 48 bit wide in the current x86-64 specification). We contrast the details for resurrection and the respective read and write barriers in Figure 4 with SAHTM and in Figure 5 with Messy-HTM that keeps the cache content available. In combination with full hybrid TM systems em-

[4] Note that software and hybrid TM systems already require this instrumentation, and that compiler support for automatically adding this instrumentation is available in several production-grade compiler frameworks.

ploying STM for large transactions, SAHTM's logging mechanism allows seamless transition from SAHTM to STM execution without requiring an abort: the SAHTM's access log is replayed into the STM's tracking data structures. In total, we add points on the performance vs capacity / functionality spectrum. In Section 7 we will quantify the performance characteristics of each of these options.

4.1 Cost

There is no additional hardware cost associated with the logging-based SAHTM variant of resurrecting transactions, and even keeping the transactional working set in the caches with Messy-HTM requires only minimal changes. The cache remains unchanged from an existing cache-based HTM design, especially if that HTM already offers non-transactional accesses, as is the case with ASF. We change the handling of aborts: the core will not clear the transactional state in the caches when encountering an interrupt, exception, or instruction that calls into the OS. Furthermore, the core needs to memorise the fact that there is a suspended transaction and detect TLB flushes. The cost for these modifications in terms of silicon real-estate is small, but they incur a design and verification cost. We acknowledge the cost and offer an intermediate update step with our logging-based SAHTM approach.

The OS does not need to manage the transactional state of the application with our extensions, because all detection of context switches happens in hardware conservatively and is handled by fully aborting the transaction. Because the register state of each application reflects the aborted state already, no TM-specific update of the architectural state has to occur during / for a context switch. Accessing transactionally written data when the transaction is suspended is dangerous in all suspend / resume proposals, because hardware may need to drop the transactional updates to ensure the consistency of the working set. Our SAHTM approach side-steps the issue by hiding the transactional updates from the invoked OS routines. If data needs to be reliably transferred into the OS handler through memory, ASF's non-transactional stores provide a safe way to protect against spontaneously disappearing working sets. It is also possible to instrument library codes so that they perform a lookup in the log. This is reminiscent of techniques for achieving open nesting in BEHTM [19]. We will show a way to safely handle state transferal into the kernel in Section 5.

5. Escape Actions from Hardware Transactions

ASF already allows individual loads and stores to escape from a transaction. Composing longer code blocks escaping these mechanisms (as in [40]) is complicated due to the synchronous nature of aborts in ASF; whenever a condition for abort is detected (to include concurrent memory conflicts and timer interrupts), control flow can transfer from the middle of a basic block into the abort handler. This is usually not an issue with transactional code since all side-effects are tracked and rolled back. However, interrupting an escape action while it has not finished executing can leave escaped data in an inconsistent state.

Given our mechanism for suspend/abort from Section 4, it is straightforward to provide support for escape actions as well. For simplicity, we adhere to the principles set forth for delegated escape actions [19], namely that an escape action's accesses are disjoint with respect to the calling transaction's read and write sets.

Without loss of generality, we assume that escape actions consist entirely of non-transactional code (i.e., they use `RD/WRINVMODE` at their entry and exit to set and restore this status). To support escape actions, we update a thread-local field F prior to beginning the escape action. Should an abort occur during the escape action, the abort handler first checks F: if it is set, the handler memorises information about the abort in another field H, and then uses a `CONTINUE` to immediately resume the escape action. In this manner, the (non-transactional) escape action code will not be aborted while holding locks, or while at some point where invariants may not hold. Upon completion of the escape action, the code registers any undo actions related to the escape action, clears F, and checks H. If H indicates that an abort occurred during the escape action, the program uses the additional information saved by the handler to complete the abort, closely resembling an explicit abort in a software TM implementation.

In the event that the escape action requires a context switch or system call, we seamlessly transition to a more heavyweight suspend/resume operation. This may require the transaction to abort and restart in SAHTM mode, if accesses have not been logged. However, such a transition is only necessary if the system call cannot be emulated; otherwise, the `CONTINUE` would immediately return to the OS trap instruction, which would abort the newly started transaction and return to the abort handler. By switching on the fly, we can suppress aborts for lightweight escape actions, while still supporting escape actions that must be executed from a non-transactional context.

5.1 Hardware Cost and OS Interaction

There is no additional hardware cost to provide support for escape actions. With respect to OS interaction, the common case again requires no support. However, run-time libraries that ought to run as escape actions will require wrapper code to manage the F and H flags. If an escape action must access state modified by the transaction, then the action must be rewritten to check the access logs managed by the transaction, and the transaction will require SAHTM instrumentation. More complex software-based techniques are possible, wherein concurrent transactions are blocked and the transaction executing the escape action temporarily becomes irrevocable. In the absence of workloads requiring such functionality, the cost of this approach is likely too high.

Note, too, that it is not necessary to execute every escape action as a non-transactional operation. With the new instructions to control whether `LOCK` prefixes indicate transactional or non-transactional accesses, binaries (such as libc functions) may be called in either an escaping fashion or through making all their memory references transactional. By storing the current mode of the transaction (inverted / non-inverted, attempt continue / abort) in a thread-local variable, we can ensure that the right strategy is employed in the abort handler, and that the best approach is taken for each library function.

6. Multi-Location Alert-on-Update

Alert-on-update (AOU) is a mechanism that uses transactional read set tracking to generate user-level signals upon certain cache evictions [34]. To synthesise alert-on-update on top of our extended abort behaviour, we begin an ASF transaction via the `SPECULATE` command, use transactional (`LOCK`-prefixed) loads in place of AOU loads, and keep all other memory accesses of the program non-transactional. We also non-transactionally manage a record of all AOU-loaded locations (Figure 6). Whenever a monitored (AOU) location is written to by another core and evicted from the cache, the ASF transaction aborts and jumps to its abort handler, which serves as (or chains to) the alert handler (replacing the abort handling in Figure 3). Note that changes to program state will not roll back on transaction abort, since we have chosen in this case for the default behaviour of loads and stores to be non-transactional.

After the handler finishes resolving the alert, it starts a new transaction, re-adds the monitored location(s) to the working set, and continues execution at the previously aborted location through a `CONTINUE` instruction. Due to the overlapping nature of starting a transaction before executing `CONTINUE` to restore the state of the preceding transaction, we must take care to use alternating buffers

```
1: function ALOAD($address$)        ▷ Adds an alert on update of location $address$
2:     repeat
3:         $val_1 \leftarrow [address]$
4:         $locs \leftarrow locs \setminus (address, *) \cup (address, val_1)$        ▷ Add value early to prevent data race
5:         txload $val_2 \leftarrow [address]$
6:     until $val_1 = val_2$        ▷ Ensure that this was a race-free ALOAD
7:     return $val_1$
```

Figure 6. Implementing alert-on-update with ASF.

for the storage of the clobbered registers. To prevent unbounded abort recursion, we flatten aborts in the overlap region. The abort handler will also be invoked for other reasons than changes in the monitored location(s), such as syscalls and timer interrupts, but those cases can be discerned through the abort condition codes presented to the abort handler by ASF. Continuing the transaction is usually enough to continue execution, but some cases require simple emulation of instructions illegal within transactions (such as I/O-related system calls). Often, the mechanisms discussed in previous sections suffice for this emulation. In other cases, lightweight instrumentation is required to (a) `COMMIT` the `SPECULATE` operation, then perform the operation, and then begin a new `SPECULATE` and restore all AOU loads. Note that this is simpler than suspend/resume, because there is no transactional state that must be protected during the (escaped) syscall.

6.1 Privatisation-safe STM with AOU

To demonstrate the utility of AOU, we consider its use to strengthen the correctness guarantees of an STM algorithm without adding overhead. In general, language-level implementations of transactional memory (TM) require the TM implementation to be privatisation safe [1]. Roughly, this means that execution with transactions appears equivalent to an execution in which all critical sections are protected by a single global lock [20]. This, in turn, boils down to two problems [35]: when committing a transaction T that logically transitions some region of memory R to a state in which other threads can no longer access R transactionally, T must be sure that (a) any transaction that committed or aborted before T committed must not still be cleaning up its changes to R[5], and (b) any transaction still running will not continue to use data in region R. These two problems are sometimes called the "delayed cleanup" and "doomed transaction" problems.

The most general solution to both problems is a heavyweight quiescence mechanism, in which every committing writer transaction must wait for all concurrent transactions to commit or abort *and clean up* before it departs from its commit function. Decoupled solutions to the problem tend to scale better, but these solutions rely on polling to solve the doomed transaction problem: when T commits, concurrent doomed transaction D may be in the midst of accessing R, and will not determine that it should abort until its *next* access of shared memory. The problem with this approach is that when T wishes to deallocate R, it cannot prevent concurrent accesses by D. The only known solution in this case is to change the allocator, so that T's deallocation is deferred until D completes [12]. This mechanism, also popular in RCU synchronisation, can result in an unbounded delay between when T commits and when R is finally reclaimed. Consequently, production STM implementations choose the quiescence approach.

[5] We use the term "clean up" to refer to write-back in STM implementations that use commit-time locking, and also to refer to undo operations at abort time in STM implementations that lock and modify memory before reaching their commit point.

Our implementation of AOU enables the use of decoupled validation without incurring the risk of unbounded delay during reclamation. The key observation is that if transactions use AOU to monitor the notification location that they formerly polled, then they will be notified *immediately* when it changes, due to transactional abort. The notified thread can then ensure its validity with respect to the newly committed transaction. If it remains valid, it can resume; otherwise it will abort. Crucially, the interruption, validation, and abort of a transaction will occur between when T commits and when D might next access R. That is, if T deallocates R, it cannot affect D so long as D's validation does not access R. In practical terms, this prohibits algorithms that use value-based validation [8, 24], but otherwise carries no cost.

In prior work on AOU [34], the AOU hardware monitored some subset of read locations and metadata to avoid validation. In contrast, we use the AOU mechanism as a polling-free, low-latency, immediate cross-core (cross-thread) communication mechanism that invokes the STM's validation handler upon the commit of every writing transaction; this requires AOU tracking of only a *single* location, and is thereby beneficial for even the most capacity-constrained HTM implementations. Since our AOU implementation is built atop our transparent suspend/resume mechanism, it is practical: system calls and quantum expirations will cause a transaction to resume from inside its validation handler, and aborts during calls to lock-based libraries (such as `malloc`) can be delayed safely. Furthermore, the use of AOU-based notification simplifies the STM algorithm, eliminating some comparisons for corner-case behaviours.[6]

6.2 Hardware Dependence

In contrast to the transactional suspend and resume mechanism from Section 4, with AOU we employ the transactions as an auxiliary wrapper to non-transactional code. Therefore, we expect to see non-transactional accesses as the norm, which is well reflected by ASF's (non-inverted) `SPECULATE` instruction. For hardware implementations with small HTM implementations, programming with AOU will side-step the capacity limitations of the HTM, while still gaining some benefits relative to a pure STM library.

As before, the OS / hypervisor can remain oblivious of usage of AOU in application code. This is an improvement over the original alert-on-update proposal [34]. However, in some cases it may be necessary to wrap system calls as escape actions that run outside of a transactional context, in order to prevent infinite aborts at the trap instruction. As discussed above, these escape actions are simpler than those in Section 5: since we aren't using the HTM for data versioning, we can simply commit the `SPECULATE` region, run the escape action, and then start a new `SPECULATE` region.

7. Evaluation

In this section, we evaluate the performance of our extensions. All experiments were performed on Marss86, a cycle-accurate x86 simulator [25]. We extended Marss86 with ASF support based on PTLsim-ASF [5],[7] and then added the features discussed in this paper. Our evaluation of AOU-enhanced STM uses benchmarks from the RSTM [33] open-source library, and we evaluate suspend/resume performance in the TinySTM [5] toolchain. Experiments ran for 10,000 successful transaction commits per thread, and all re-

[6] These simplifications, while useful, are lengthy and limited in novelty. We intend to distribute them via open-source channels, but due to limited space cannot include code listings in this submission.

[7] We ported and heavily extended code from PTLsim-ASF (`https://github.com/stephand/ptlsim/tree/ptlsim_asf`) to work with the new memory hierarchy in Marssx86 and made changes available at `https://bitbucket.org/stephand/marss86-asf`.

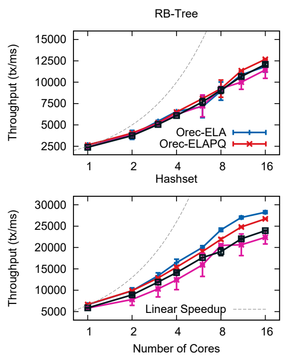


Figure 7. Throughput for RSTM OrecELA with AOU-enhanced privatization safety.

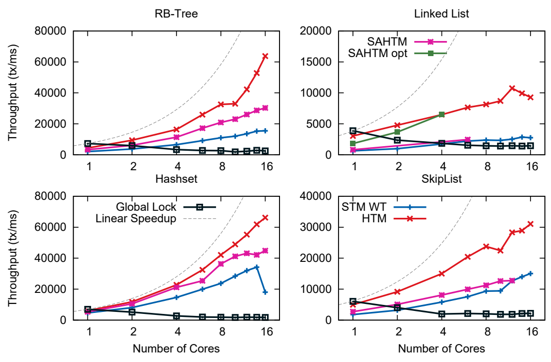


Figure 8. Throughput for Intset benchmark using different TM implementations

sults are the average of three (RSTM RB-Tree and Hashset) or six (RSTM Linked List) trials. Our simulated machine features a modern processor with 16 out-of-order cores, with per-core L1 and L2 caches and a shared L3 cache, and realistic default sizes for them and DRAM. All experiments run in full-system simulation, with Ubuntu Linux 12.04 LTS. All RSTM code was compiled with GCC 4.6.3 and the "-O3 -flto" optimisation options. The TinySTM-based experiments use the Clang version of DTMC compiler [5] using LLVM 3.2 with optimisation "-O3 -flto".

7.1 Alert on Update for Privatisation-Safety

We enhance two variants of RSTM's OrecELA STM implementation with our AOU implementation outlined in Section 6: *OrecELA* and *OrecELA + AOU* use ordered commit departure, *OrecELAPQ* and *OrecELA + AOUQ* perform quiescence to achieve privatization-safety. The AOU-enhanced variants do not require delayed reclamation and therefore are not prone to unbounded delays between memory free and actual availability of the memory. Instead, they can immediately reclaim memory at transaction commit without inducing segmentation violations in doomed readers.

We use three data-structures implementing the integer set interface that are provided with the RSTM source code: ListBench, a singly linked list; HashBench, a hash table; and TreeBench, using a Red-Black tree to store the set. We vary the thread count, fix the total number of transactions and keep other parameters at their defaults (set size of 256 elements, pre-filled to 128 elements, value range 0 - 255, update rate 66%).

Comparing the performance for the three data structures in Figure 7, we find that, overall, performance of the four solutions is comparable. The linked list produces large levels of contention that cause significant jitter in our results at higher thread counts. The AOU-based STMs change the timing of validation due to their synchronous signalling. In the linked list, abort characteristics change for long transactions, because a short committing writer causes an immediate validation and abort of a long, uncommitted transaction that has traversed the list. In the non-AOU STM variants, that long transaction would also be doomed but detect the conflict only when it tries to commit.

In the hash set experiment, all transactions are small and conflict rarely. For the AOU-enabled cases, the frequent writers force many AOU-induced revalidations causing overhead for the short transactions. The net effect is an increase of the effective length of the transactions. The present performance delta reflects this, but we argue that the additional application-level guarantees outweigh the performance impact of this application worst-case.

In summary, we show that our multi-location AOU prototype (based on the extensions to our best-effort HTM ASF from Section 3) works with small overhead and can be used to enhance an STM library. We found that some work in the STM logic was necessary to adapt it to handle the AOU-induced notifications arriving at arbitrary code locations. To mitigate, we selectively switched between synchronous and asynchronous handling (see Section 5).

7.2 Transaction Suspend / Resume

For testing the transaction suspend / resume with SAHTM, we base our work on the HyTM implementation in TinySTM which uses plain ASF HTM with minimal additional software support to implement serial irrevocable execution and memory management [5]. We extend read and write accesses to additionally log accessed addresses and values as described in Section 4. To reduce memory pressure and to keep the log overhead small, we encode additional meta data (size of the accessed data, read/write operation) in the upper 8 bits of the address part of the log entries.

TinySTM contains four benchmarks implementing the integer set interface: linked list, skip list, hash set, and red-black tree. We evaluate each with an initial capacity of 256 elements and an update rate of 20%. Figure 8 shows the throughput achieved for an increasing number of cores for those benchmarks. We compare four different transactional memory implementations: *Global Lock* which uses a single global lock for synchronisation - no accesses inside a transaction are instrumented; *STM WT* a state-of-the-art write through implementation for Software TM [28]; *HTM* the hardware TM implementation using ASF as described above; and our new implementation, *SAHTM*, that adds the logging read/write barriers. *SAHTM opt* is a proof-of-concept *hand-optimised* binary that bypasses shortcomings in LLVM's optimisation passes, in particular hoisting loop invariants across barriers and handling inline assembly with memory operands. We manually performed these optimisations on the SAHTM binary and briefly show first results for linked list.

Looking at the throughput and transactional statistics (not shown due to space limitations), we find that the RB-Tree and Hash Set scale well, and HTM and SAHTM behave very similarly for all core counts. The difference in throughput is due to the additional overhead of the read/write barrier implementation. For higher core counts, (> 8) SAHTM's scalability shows a more similar behaviour to STM. The reason for that is the additional overhead of the barriers and the need to replay the logs in case of contention.

Both Linked and Skip List are a high-intensity workload for all of our TM implementations: the linear scanning loop exposes every bit of added overhead, due to very little other logic and predictable access patterns. SAHTM has a slight performance advantage over the STM implementation but the logging overheads are much more exposed than in RB-Tree and Hash Set. HTM scales better, because its list traversal loop is extremely small and the additional logging in SAHTM consumes memory and core execution bandwidth–causing significant slowdown. The major reason for SAHTM's overhead is missed optimisation opportunities by the compiler (appropriate hoisting and reuse of the pointer to the log structure instead of fetching them for each iteration in the loop) adding superfluous instructions and extra memory traffic. Our manually optimised prototype (SAHTM opt) for Linked List shows the potential benefit from these optimisation which should be performed by the compiler. This implementation is almost on par with the HTM implementation for the thread counts we managed to test.

In summary, our suspend/resume experiments show that it is indeed possible to implement transaction suspend/resume through resurrection of aborted transactions in an OS-transparent way. For data-structures with complex access patterns, the additional logging instructions and memory traffic are apparently effectively hidden in branch mis-predictions, cache misses and available instruction-level parallelism. We see strong hints that carefully controlling the optimisation in our framework can significantly drive down the execution resource demand of our logging code and thus can have low overheads also in simple data access patterns.

8. Related Work

For nearly a decade, researchers have been exploring mechanisms for exploiting bounded HTM resources in more robust and programmable ways. One of the earliest proposals, from Zilles and Baugh [40], introduced suspend/resume as a mechanism for allowing hardware transactions to avoid the size constraints of HTM when executing operations that either (a) are known to never cause conflicts, or (b) are best served with other concurrency control mechanisms (e.g., memory allocation). In that proposal, the hardware still controlled the execution of the transaction, with their extensions serving only as a means of temporarily suspending the transaction. Furthermore, without ASF-style HTM resources, this work required significant changes to the underlying HTM, whereas our implementation can leverage existing ASF support to require only a minimal amount of additional hardware and software extension. Transaction escape actions [22] provided a similar feature in LogTM, though again there was a noticeable hardware cost.

A more aggressive approach to exploiting bounded HTM is exhibited by the many hardware accelerated software TM (HASTM) systems. These proposals typically extended a traditional ISA with features resembling TM hardware, most notably mechanisms for tracking locations accessed within a region [30, 34, 36]. While these features closely resembled those of more complete HTM proposals, the control of transactions was fully delegated to a software library. Additional proposals offered programmable control of data versioning and buffering [21, 31, 32]. This offered low enough overhead to be competitive with full HTM implementations, but typically with less hardware complexity. At the extremes, Casper et al. showed that an out-of-core FPGA-based prototype could deliver strong transactional performance [3], and Carouge et al. showed that HTM resources could make an existing STM algorithm lock-free without affecting performance [10].

Our work on hybrid TM is inspired by a wide variety of algorithms proposed in the literature. These algorithms attempt to build a runtime system in which some transactions are fully controlled by software, and others accelerated by hardware. The benefit of such a system is that there is a graceful fallback for those transactions whose memory accesses or running times extend beyond the limits of the HTM subsystem. However, small, short transactions must, in turn, sacrifice some performance in order to be compatible with these software transactions. Initial hybrid systems focused on correctness and non-blocking progress [16], after which the focus turned to systems in which transactions operated in distinct modes (i.e., software-only, hardware-only, and serial) [5, 18]. Later works showed that true concurrency between hardware-controlled and software-controlled transactions was possible, but that specific characteristics of the hardware (most notably the availability of non-transactional loads and stores within the hardware transaction) was critical to achieving good performance [9, 29]. This paper builds on prior work by showing that minor extensions to the HTM can simplify the implementation of such systems without affecting performance.

Several groups have also explored the use of HTM resources for purposes orthogonal to scalable concurrent execution of language-level transactions. The original AOU paper [34] proposed several uses of AOU outside of TM implementation, such as for reducing the cost of polling in event-based systems and implementing a limited form of active messages [37]. Neelakantam et al. [23] showed that hardware TM extended with a self-abort instruction could be used by compilers for speculative unsafe optimisation. Most recently, Unsal et al. have shown that HTM resources could be used both to detect and prevent transient faults during sequential execution [38], and as a mechanism for lowering power consumption by running a processor at an extremely low voltage, and then using transactional rollback and recovery to compensate for faults that occur during execution [7].

9. Conclusion

In this paper, we presented small modifications to the ASF HTM proposal that change abort handling to allow transaction resurrection, and added two instructions for explicitly managing the polarity of transactional / non-transactional accesses. With only these two minor extensions, neither of which requires extensive hardware verification or changes to cache structures and protocols, we showed that Alert-on-Update, Escape Actions, and Suspend / Resume can all be supported in an otherwise relatively simple hardware TM. We believe that our modifications lie in the same complexity realm as the differences between the various HTM industry proposals and thus can be implemented in actual hardware, for example as an extension to first generation HTM support. This is a promising direction that can turn these *synchronisation* extensions into *synchronisation and speculation* extensions that support a rich transactional programming environment.

Acknowledgments

Stephan received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the ParaDIME Project, grant agreement No. 318693, and developed related, initial ideas while employed at Advanced Micro Devices, Inc. At Lehigh University, this work was supported by the US National Science Foundation under grant CNS-1016828 and CCF-1218530. Martin received funding by Deutsche Forschungsgemeinschaft (grant agreement No. FE 1035/1-2.)

References

[1] A.-R. Adl-Tabatabai and T. Shpeisman (Eds.). Draft Specification of Transactional Language Constructs for C++, Aug. 2009. `http://software.intel.com/file/21569`.

[2] J. Brewer. IBM Unveils zEnterprise EC12, a Highly Secure System for Cloud Computing and Enterprise Data. `http://www-03.ibm.com/press/us/en/pressrelease/38653.wss`, Aug. 2012.

[3] J. Casper, T. Oguntebi, S. Hong, N. Bronson, C. Kozyrakis, and K. Olukotun. Hardware Acceleration of Transactional Memory on Commodity Systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, Calif., Mar. 2011.

[4] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, and S. Yip. Rock: A High-Performance Sparc CMT Processor. *IEEE Micro*, 29(2):6–16, March–April 2009.

[5] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Riviere. Evaluation of AMD's Advanced Synchronization Facility within a Complete Transactional Memory Stack. In *Proceedings of the EuroSys2010 Conference*, Paris, France, Apr. 2010.

[6] J. Chung, L. Yen, S. Diestelhorst, M. Pohlack, M. Hohmuth, D. Grossman, and D. Christie. ASF: AMD64 Extension for Lock-free Data Structures and Transactional Memory. In *Proceedings of the 43rd IEEE/ACM International Symposium on Microarchitecture*, Atlanta, Ga., Dec. 2010.

[7] A. Cristal, O. Unsal, G. Yalcin, C. Fetzer, J.-T. Wamhoff, P. Felber, D. Harmanci, and A. Sobe. Leveraging Transactional Memory for Energy-efficient Computing below Safe Operation Margins. In *Proceedings of the 8th ACM SIGPLAN Workshop on Transactional Computing*, Houston, TX, Mar. 2013.

[8] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming*, Bangalore, India, Jan. 2010.

[9] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. Scott, and M. Spear. Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, Calif., Mar. 2011.

[10] Francois Carouge and Michael Spear. A Scalable Lock-Free Universal Construction with Best Effort Transactional Hardware. In *Proceedings of the 24th International Symposium on Distributed Computing*, Cambridge, Mass., Sept. 2010.

[11] M. P. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, Calif., May 1993.

[12] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. Hertzberg. A Scalable Transactional Memory Allocator. In *Proceedings of the International Symposium on Memory Management*, Ottawa, Ont., Canada, June 2006.

[13] *Power ISA(tm) Transactional Memory*. IBM(R), 2.07 edition, Dec. 2012.

[14] *Intel(R) Architecture Instruction Set Extensions Programming Reference*. Intel Corp., 319433-012a edition, Feb. 2012.

[15] C. Jacobi, T. Slegel, and D. Greiner. Transactional Memory Architecture and Implementation for IBM System z. In *45th Int. Symp. On Microarchitecture*, 2012.

[16] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid Transactional Memory. In *Proceedings of the 11th ACM Symposium on Principles and Practice of Parallel Programming*, New York, N.Y., Mar. 2006.

[17] H. Le, G. Guthrie, D. Williams, M. Michael, B. Frey, W. Starke, C. May, R. Odaira, and T. Nakaike. Transactional memory support in the IBM POWER8 processor. *IBM Journal of Research and Development*, 59(1):8–1, 2015.

[18] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased Transactional Memory. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, Portland, OR, Aug. 2007.

[19] Y. Liu, S. Diestelhorst, and M. Spear. Delegation and Nesting in Best Effort Hardware Transactional Memory. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, Pittsburgh, PA, June 2012.

[20] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. Hudson, B. Saha, and A. Welc. Practical Weak-Atomicity Semantics for Java STM. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[21] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, Calif., June 2007.

[22] M. Moravan, J. Bobba, K. Moore, L. Yen, M. Hill, B. Liblit, M. Swift, and D. Wood. Supporting Nested Transactional Memory in LogTM. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, Calif., Oct. 2006.

[23] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware Atomicity for Reliable Software Speculation. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, Calif., June 2007.

[24] M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, Brasov, Romania, Sept. 2007.

[25] A. Patel, F. Afram, S. Chen, and K. Ghose. MARSSx86: A Full System Simulator for x86 CPUs. In *Design Automation Conference 2011 (DAC'11)*, 2011.

[26] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *Proceedings of the 32nd International Symposium on Computer Architecture*, Madison, Wis., June 2005.

[27] H. E. Ramadan, C. J. Rossbach, and E. Witchel. Dependence-aware Transactional Memory for Increased Concurrency. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 246–257. IEEE Computer Society, 2008.

[28] T. Riegel, C. Fetzer, and P. Felber. Time-Based Transactional Memory with Scalable Time Bases. In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures*, San Diego, California, June 2007.

[29] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer. Optimizing Hybrid Transactional Memory: The Importance of Nonspeculative Operations. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, June 2011.

[30] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural Support for Software Transactional Memory. In *Proceedings of the 39th IEEE/ACM International Symposium on Microarchitecture*, Orlando, FL, Dec. 2006.

[31] A. Shriraman, M. F. Spear, H. Hossain, S. Dwarkadas, and M. L. Scott. An Integrated Hardware-Software Approach to Flexible Transactional Memory. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, Calif., June 2007.

[32] A. Shriraman, S. Dwarkadas, and M. L. Scott. Flexible Decoupled Transactional Memory Support. In *Proceedings of the 35th International Symposium on Computer Architecture*, Beijing, China, June 2008.

[33] M. Spear. Lightweight, Robust Adaptivity for Software Transactional Memory. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.

[34] M. F. Spear, A. Shriraman, L. Dalessandro, S. Dwarkadas, and M. L. Scott. Nonblocking Transactions Without Indirection Using Alert-on-Update. In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures*, San Diego, Calif., June 2007.

[35] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. Ordering-Based Semantics for Software Transactional Memory. In *Proceedings of the 12th International Conference On Principles Of DIstributed Systems*, Luxor, Egypt, Dec. 2008.

[36] S. Stipic, S. Tomic, F. Zyulkyarov, A. Cristal, O. Unsal, and M. Valero. TagTM - Accelerating STMs with Hardware Tags for Fast Meta-data

Access. In *Proceedings of the 2012 Design, Automation & Test in Europe Conference*, Dresden, Germany, Mar. 2012.

[37] T. von Eicken, D. Culler, S. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[38] G. Yalcin, O. Unsal, and A. Cristal. FaulTM: Error Detection and Recovery Using Hardware Transactional Memory. In *Proceedings of the 2013 Design, Automation & Test in Europe Conference*, Grenoble, France, Mar. 2013.

[39] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of intel transactional synchronization extensions for high-performance computing. In W. Gropp and S. Matsuoka, editors, *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*, pages 19:1–19:11. ACM, 2013. ISBN 978-1-4503-2378-9. . URL `http://doi.acm.org/10.1145/2503210.2503232`.

[40] C. Zilles and L. Baugh. Extending Hardware Transactional Memory to Support Non-Busy Waiting and Non-Transactional Actions. In *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Ottawa, Ont., Canada, June 2006.