Speculation in Parallel and Distributed Event Processing Systems

Dissertation
zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingerichtet von

M.Sc. Andrey Brito

geboren am 21.05.1979 in Campina Grande, Brasilien

Gutachter:	Prof. Christof Fetzer, Ph.D. Institut Systemarchitektur Technische Universität Dresden	Prof. Pascal Felber, Ph.D. Institut d'informatique Université de Neuchâtel

Tag der Verteidigung: 10. Mai 2010

Dresden, den 7. Juli 2010

Gedruckt mit Unterstützung des Deutschen Akademischen Austauschdienstes

Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge, it contains no material previously published or written by another person nor material that to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher education, except where due acknowledgment has been made in the text.

Dresden, March 23, 2010

Andrey Brito

Abstract

Event stream processing (ESP) applications enable the real-time processing of continuous flows of data. Algorithmic trading, network monitoring, and processing data from sensor networks are good examples of applications that traditionally rely upon ESP systems. In addition, technological advances are resulting in an increasing number of devices that are network enabled, producing information that can be automatically collected and processed. This increasing availability of on-line data motivates the development of new and more sophisticated applications that require low-latency processing of large volumes of data.

ESP applications are composed of an acyclic graph of operators that is traversed by the data. Inside each operator, the events can be transformed, aggregated, enriched, or filtered out. Some of these operations depend only on the current input events, such operations are called *stateless*. Other operations, however, depend not only on the current event, but also on a state built during the processing of previous events. Such operations are, therefore, named *stateful*.

As the number of ESP applications grows, there are increasingly strong requirements, which are often difficult to satisfy. In this dissertation, we address two challenges created by the use of stateful operations in a ESP application: (*i*) stateful operators can be bottlenecks because they are sensitive to the order of events and cannot be trivially parallelized by replication; and (*ii*), if failures are to be tolerated, the accumulated state of an stateful operator needs to be saved, saving this state traditionally imposes considerable performance costs.

Our approach is to evaluate the use of speculation to address these two issues. For handling ordering and parallelization issues in a stateful operator, we propose a speculative approach that both reduces latency when the operator must wait for the correct ordering of the events and improves throughput when the operation in hand is parallelizable. In addition, our approach does not require that user understand concurrent programming or that he or she needs to consider out-of-order execution when writing the operations.

For fault-tolerant applications, traditional approaches have imposed prohibitive performance costs due to pessimistic schemes. We extend such approaches, using speculation to mask the cost of fault tolerance.

Acknowledgments

First, I am indebted to my supervisor, Christof Fetzer, for giving me the opportunity to join such a great group and for always keeping his door open during the last four years.

I thank all my colleagues from the Systems Engineering group: André Schmitt, André Martin, Claudia Einer, Diogo Becker, Gert Pfeifer, Jons-Tobias Wamhoff, Karina Wauer, Marc Brünink, Martin Süßkraut, Martin Nowack, Robert Fach, Ryan Spring, Stefan Weigert, Stephan Creutz, Thomas Knauth, Torvald Riegel, Ute Schiffel, Zbigniew Jerzak. I thank you all for the great atmosphere at the office and great conversations during lunch breaks.

Special thanks go to Ute, Marc, Ryan, and Diogo, for providing me uncountable suggestions during the writing of this dissertation.

I would like to thank Pascal Felber, from the University of Neuchâtel, for reviewing this dissertation and for all the discussions and feedback.

Some of the work in this dissertation was done in cooperation with other people. Therefore, thanks to Heiko Sturzrehm for his help during early implementations of the event processing engine; Stefan Weigert and Martin Süßkraut for the work with checkers for detecting software bugs; and to Diogo Becker for the discussions about determinism.

Finally, I am specially grateful to my parents and to my wife Esther for the continuous support and motivation and to the DAAD for the financial support.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Many modern computing systems produce vast amounts of data continuously. In some cases, data need to be stored, but due to the high volumes, storage is impractical if not impossible. In other cases, data need to be processed and trigger adequate response actions in (soft) real-time. In both scenarios, a continuous high-rate flow of low-level data (e.g., individual sensor reads) need to be processed to generate a lower-rate flow of useful information (e.g., averages). The high-relevance information, now at a reduced rate, can then be stored or processed in a timely manner. This task is the goal of event stream processing (ESP). In this chapter we give a summarized view of this dissertation. We first informally define ESP systems and give an example that will be used throughout the text. After that, we enumerate the main challenges addressed and, finally, give a roadmap for the rest of the work.

1.1 Event stream processing systems

Event processing started as an initiative to automate actions in databases. Instead of periodically checking whether a table was updated and then triggering an action, extensions were proposed to allow databases to *actively* react to table updates. This new type of systems was named *active databases* [DBB+88, Day94]. As applications got more complex, more features were proposed to allow, for example, that actions would only be triggered when a set of specific updates occurred. The original definition of an event was, therefore, "an update in the database". As an example, HiPAC [DBB+88] allowed the programming of sets of events-condition-actions (ECA) rules that were checked each time an update event occurred in the database.

Nevertheless, the main goal of databases is to store information persistently in a way that allows efficient subsequent accesses. There is, however, a range of applications in which the priority is to analyze the data in real time. Persistent storage of the data is secondary (e.g., only for archiving purposes), or even impossible due to the high-volume and continuous arrival of new data. Recently, the development of cheap and low-latency communication and the explosion of interconnected devices caused a rapid growth in these applications. The increase in interest led to the development of a new research area focused more on low-latency high-throughput processing and less on storage and indexing of data. A summarized list of key differences between these two approaches is shown in Table 1.1.

Event stream oriented systems	*Databases*
Dynamic data and static queries	Static data and dynamic queries
Data stream is infinite	Storage capacity is bounded
Single (or limited) look at data	Data is persistent and repeatedly accessed
Latency prioritized (e.g., real time)	Throughput prioritized (e.g., batching)
Approximative results predominant	Exact results predominant
Often distributed processing	Often centralized processing

Table 1.1: Comparison between ESP and DB systems.

In ESP systems queries are static, long lived, and the system is traversed by the data. The arrival of a piece of data is then a potential trigger for the emission of a result. In contrast, in databases, the queries are dynamic, short lived, and access static data on the storage. In databases, the trigger for the emission of results is the arrival of a new query. In addition, because of the continuous arrival of data (i.e., the stream is infinite) and the low-latency requirements, ESP system assume that the whole stream cannot be kept and randomly accessed. As a consequence of this limited access of events (e.g., only recent or sampled events are considered), results are often approximations in ESP systems. Finally, because the data is flowing through the system, an ESP application is normally composed of several intermediate computations in a pipeline. These intermediary computations can be spread among different nodes in a distributed system (see example below). In contrast, in conventional databases the complete processing of a query is normally executed where the data is.

These event stream-oriented systems have been known by many different names. Commons terms are complex event processing (CEP) [Luc01], event stream processing [Luc01], and data stream processing [BBD+02, GO03]. Sometimes the different names reflect slight differences in the approaches taken. For example, CEP focuses more on the search for patterns of events (a complex event is composed by a set of regular events) than in the low-latency high-troughput processing. However, different names often reflect only different motivating applications.

ESP applications are commonly architected in the form of an acyclic graph as exemplified in Figure 1.1. On the extreme left side of the graph are the *producers*, which generate the events. Typically, the events compose a continuous stream of low-level pieces of data. On the extreme right are the *consumers*, which are interested in high-level information extracted from the low-level data. The central part of the graph is composed by the *operators*. They are responsible for transforming high-volume low-level data into a lower volume of high-level information. The events traverse the operator graph and are processed in each node by different types of computations. Examples of such computational tasks are: filtering, transformation/convertion, enrichment (e.g., addition of offline information), aggregation (summarization of multiple events in one with higher-level information, e.g., average), join (combination of events coming from different streams to produce a single event combining information from the source events), and union (merge of events from different streams into a single stream containing all events).

Events can be seen as messages that are not explicitly addressed and carry some information that is relevant for the application. The transport of the events can be implemented by a publish/-

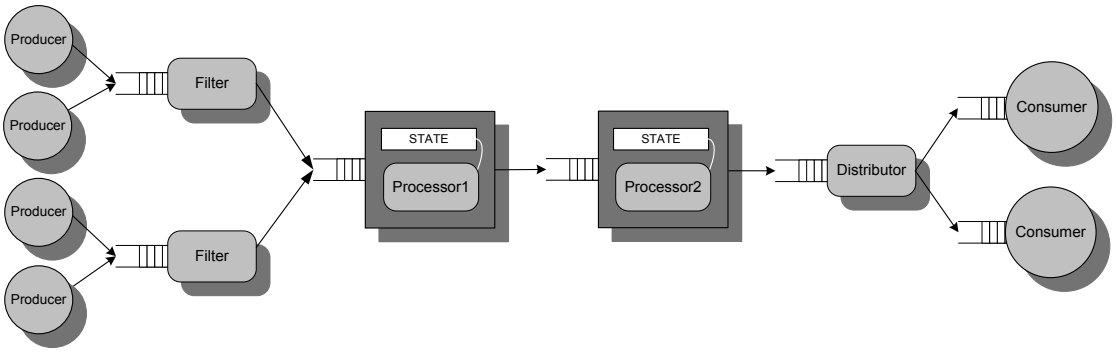


Figure 1.1: A simple prototypical ESP application graph.

subscribe system [EFGK03] or simply by having static connections between components. For example, the producer nodes in Figure 1.1 could monitor the temperature of a set of machines. Events could be generated either at fixed intervals or when there is a temperature change. The operators in this case could analyze the measurements to achieve goals as simple as detecting overheating and shutting down machines, or as complex as predicting failures and generating early alarms. In general, ESP systems are very popular in monitoring applications. Further applications that increasingly rely on ESP systems are process control, algorithmic trading, network monitoring, and sensor networks.

It is also often the case that ESP systems are structured as a tree, with many producers being distributed over a larger area and events being successively processed by less operators until the final information is used by a few consumers. In closed loop control systems, the consumers can then command distributed actuators that will initiate actions that feedback the distributed application.

Finally, as previously mentioned, the stream of events is assumed to be infinite. This assumption applies even in some cases where the stream may be finite. For example, when the data in the stream can not be fit in the memory available in the system or when the length of the stream (i.e., its duration) is such that it is undesirable that operations wait to see the complete stream before producing outputs. As a consequence, operations in an ESP application need to be designed to work with bounded subsets of the events in the stream.

In a summary, ESP systems address the problem of producers generating large amounts of raw data that, without processing, are useless to humans. Therefore, the goal of an ESP system is to process and analyze the data, producing models and detecting behavior of interest in an automated way.

1.2 Running example

In order to help identifying the issues and illustrate our contributions, we will make use of a running example. This example application consists of an algorithmic trading system. Although the requirements for the algorithmic trading systems are the same as for other monitoring ap-

plications (e.g., process control, network monitoring), finance applications have been a major motivating force to the development of event stream processing systems (e.g., [Str10, Pro10]).

A high-level view is shown in Figure 1.1. There is a number of information sources generating data at high rates. The data is filtered and then aggregated in components, which make decisions based on current and past events. At the end of the dataflow, consumers act based on the decisions made.

The application comprises six stages. The first stage consists of `Producer` nodes. These nodes are sensors attached to different stock exchanges and collect stock update ticks from each of them. As an example, one sensor could be attached to the US-based NASDAQ stock exchange, another to the German DAX, another to the Brazilian BOVESPA, and so on.

In the second stage, simple filters can reduce bandwidth by dropping events for stocks that are not of interest. The third and fourth stages contain the core business logic for this application. The third stage is stateful, that is, outputs depend on the state accumulated by the processing of previous events. One such operator could be to continuously compute several moving averages of a stock and to emit an output event when the current price exceeds these averages by certain percentages. Another typical stateful operation would be to detect frequent events or changes in event frequency. This could be used to select only stock quotes that, recently, have been very often negotiated.

In the fourth stage, the output from the previous stages, now consisting of more relevant stocks, could be analyzed by a more expensive algorithm. This algorithm decides on an action to be taken for that stock price update (e.g., ignore, buy, sell). For example, the fourth stage could compute the expected value of an option for that stock or decide which stocks are more promising based on behavior of the same or similar stocks in different markets.

Finally, in the fifth stage the events are reformatted, billing information is added, and then they are sent to the consumers. The consumers, in stage six, carry out the actions encapsulated in the events (e.g., execute the buy or sell orders).

1.3 Challenges and contributions

The example above has several issues that should be addressed when building a high performance and robust system. First, consider the `Filter` operators in the second stage. They may execute some message format convertion, maybe including signature verification and decryption, which are expensive operations. These are stateless operations and can be trivially parallelized by creating multiple replicas of each. Nevertheless, one downside is that different replicas may take slightly different times to process different events and this difference will change the ordering of events (even for events that come from the same source). Then, if the `Processor1` operator is order sensitive, the order needs to be restored. Restoring the order requires buffering events and waiting. Thus, it increases the end-to-end processing latency.

Second, note that the `Processor1` operator is stateful. For this reason, it cannot be parallelized by simple replication, as we did with the filters. Parallelizing a user-defined stateful operator is not trivial. On the one hand, parallel algorithms based on coarse-grained locking are easier to implement, but provide little parallelism. On the other hand, algorithms based on fine-grained locking and lock-free approaches provide high parallelism, but are difficult to

implement and error prone. Specially if development time is a concern, the time to parallelize and test the parallel algorithms by hand can be much higher than for the sequential algorithms, making the parallelization unfeasible. Thus, automated parallelization is desired in such cases.

Another disadvantage of parallel algorithms is that the interference between threads during execution may cause operations to be nondeterministic. Determinism is important when testing algorithms and also important for postmortem analysis, where inputs are replayed to understand results obtained during the original execution.

Third, fault tolerance is essential to critical applications. In our example, both `Processor1` and `Processor2` operators depend on an accumulated state. Their states need to be protected from failures. *Checkpoints*, alone or in combination with logging can help by periodically saving the state to a stable storage. Nevertheless, an operator needs to ensure checkpoints are stable on disk before emitting output events, i.e., checkpoints must be *synchronous*. Otherwise, inconsistencies may occur. To understand why, consider that the operator `Processor1` is order-sensitive and does not do synchronous checkpoints. The checkpoints could be even done asynchronously or synchronously, but not for every outputted event. This operator will occasionally output events for which the state used during computation is not yet in stable storage. If a failure occurs at such a point, after recovery the operator will be in a state previous to the state that generated the event. Even assuming that the event stream can be replayed, it is not possible to guarantee that events coming from different `Filter` operators will be received and processed in the same order (e.g., because of slightly different network latencies). As a result, computations after the recovery will be different and may even be incompatible with previous outputs. Thus, the reconstructed state in `Processor1` operator may be inconsistent with `Processor2`'s view of `Processor1` (which was constructed based on the outputs `Processor1` generated before the failure). However, if we consider synchronous checkpointing (or logging) before each output, considerable latency is added to the processing time.

Finally, in case of time-critical applications, downtime can be directly or indirectly mapped to concrete losses. For example, in our application example, missing market opportunities maps directly to financial losses. In other applications, downtime may affect credibility or safety, which can be then indirectly mapped to financial losses. In such situations, fault tolerance as detailed above is not appropriate because it both adds considerable processing latency and requires a lengthy recovery phase for restoring checkpoints and reconstructing state after a failure. *Active replication* can be used to eliminate the recovery phase by using redundant computation in replica nodes. In this case, replicas must process the same inputs in the same order and accumulate the same state. During failure-free operation, outputs from one replica are used and the other replicas are simply ignored. If one replica fails, the outputs of another replica can be used, masking the failure completely.

Active replication has also some disadvantages, however. First, it requires that replicas process events in the same order. Reliable ordered delivery normally requires coordination among replicas. Second, it restricts the computation in the replicas by requiring deterministic computations, which make parallelization difficult.

In this dissertation we address the issues enumerated above. We show how speculation can be used by an operator to process events out of their normal order and also to process events in parallel. In both cases we preserve sequential semantics, which simplifies the development

process. We also show how to use speculation in the interaction between operators and thereby mitigate the latency costs of fault tolerance, both in checkpoint-based fault tolerance and in active replication.

1.4 Outline

The rest of this dissertation is organized as follows. In Chapter 2, we provide basic concepts in the various sub-areas involved in this work. We explain the basics of stream processing operations, give the intuition behind software transactional memories, and review some important concepts in fault tolerance in distributed systems. In Chapter 3, we refine our motivation and goals and define our models and assumptions. After that, we divide the contributions in two chapters. Chapter 4 addresses the first two issues from the previous section: how speculation can be used to enable out-of-order and parallel processing within individual operators. Chapter 5 addresses fault-tolerant ESP systems and how speculation can reduce the latency costs of fault tolerance. The related work is detailed in Chapter 6. Finally, we conclude in Chapter 7, with a summary of the achievements and possible topics for future work.

Chapter 2

Background

In this chapter we introduce basic concepts from the various subareas involved in this work. We start with an overview of operators for event stream processing (ESP) applications. We discuss their goals and their internal structure. Then, we explain software transactional memory (STM). We finish by reviewing basic concepts of fault tolerance in distributed systems, including failure models and failure detection.

2.1 Event stream processing

An event-driven architecture (EDA) [Luc01, CCC07] is a software architecture paradigm where systems are built around events. In such systems, events represent state changes that are significant to the application at hand. Because of this loose definition, events can be from messages containing attribute-value pairs to objects in an objected-oriented language. In addition, events are not explicitly addressed. Events are transported from producers to consumers by an event channel. A sophisticated event channel can route events based on types or preregistered interests (e.g., a publish-subscribe system [EFGK03]). Alternatively, simpler event channels consist in fixed connections between components of the systems.

ESP is a set of techniques to build event-driven systems. In the previous chapter we presented an example ESP system and illustrated this system as a graph of operators (see Figure 1.1). The graph was composed of producers, operators, and consumers. A producer, or a *source*, is a component with one or more outputs and no inputs. The sources can either produce the events themselves or be an interface between the external world and the ESP system. In the latter case sources can be also named *input adapters*. For example, an input adapter could be a component that communicates with an external system and generates events with the received data. An *operator* has at least one input and one output. It can be seen as a program that is executed each time an event is received. Finally, a consumer, or a *sink*, is a component with one or more inputs, but no outputs. Similarly to sources, sinks can also be seen as an interface between the ESP system and the external world. In such cases, they can be also named *output adapters*. As an example, an output adapter could be a component that transform events in commands that are sent to actuators in an external system.

Components in the ESP system can be also classified as *stateful* or *stateless*. Stateful

components keep a persistent state between events. As we will detail below, this state can be from a single value up to a collection of events or a set of complex data structures. The outputs of a stateful component depend not only on the current input but also on its accumulated state. On the contrary, a stateless component keeps no state and its output depends solely on the current input event. The state in a stateful component is private to that component. Therefore, communication between components can happen only through events.

2.1.1 State in operators: Windows and synopses

There are several reasons for an operator to have a state. One reason is that the information expected at its output is of a higher level than the data available at its input. Consider for example a sensor monitoring whether a resource is busy or idle. The sensor collects a hundred samples per second. These reads do not carry much useful information to the human user of the system, who is not able to keep up with such a fast changing value. The information the user really cares about is the percentage of time that the resource was busy in the last seconds. Thus, an event processing operator that computes the *mean usage* of the resource will transform low-level, low-relevance ones and zeros into a high-level useful percentage value.

Like the example above, there are many others in which the information of interest cannot be computed based on the single, most recent input. Furthermore, in ESP systems, the only way to allow an operator to consider previous input data is by keeping a persistent state between executions.

Because ESP systems process raw data to produce useful information, they are often pictured as inverted databases. In database systems the data is kept in storage while the queries are submitted to the system and are processed to produce results. ESP systems take the opposite approach, the set of queries is fixed, predefined, and data is submitted to the system causing results to be emitted[1]. Thus, many operations that are common in databases are also expected in ESP systems. However, another feature of ESP systems is that the stream of input events is infinite. Having an infinite stream prevents the usage of any holistic operators in ESP queries (i.e., operators that consider the whole data for the computation). Take as an example the computation of the average usage of a resource. In order to compute it, we have to consider all inputs, but because the stream is infinite, this computation will never finish. This problem is referred to as *blocking* of operators.

Besides the problem of blocking operators, having infinite streams may also require unbounded memory to store the state and unbounded computational time to process this state. In addition, for some applications only the more recent events in the stream are of interest and events that are too old can be discarded.

Event windows

Event windows [BBD+02, GO03, PS06] aim at solving the four issues discussed above, namely: blocking operators, unbounded memory, unbounded computational time, and irrelevance of old

[1]Dynamic replacement of queries is also desirable for ESP systems. Nevertheless, it is an orthogonal issue that will not be addressed in this dissertation.

events. A window is a bounded list of past input events that is kept as state in an operator. The operator will then consider only the events currently in its window to carry out the computation.

The boundaries of a window can be defined by time, by a counter, or by a predicate. If a *time window* is used, the length of the window is defined by a fixed time interval. The time used as reference can be either wall time, i.e., arrival time of the events, or a timestamp in the events themselves. For example, a time window of length 6 seconds (wall time) will contain all the events received within an interval of 6 seconds, regardless of the number of events. On the contrary, a *count window* will have its length defined by a fixed number of events, regardless of their timestamps. Thus, a count window with length 10 will contain exactly 10 events. Finally, a *landmark window* is defined by a start predicate (P_{w_start}) and an end predicate (P_{w_end}): once an event satisfies the start predicate, this event and all following events will be added to the window until an event satisfying the end predicate is received.

In addition to the length of the event window, for time and count windows it is also necessary to define when new windows are created. For time windows, a *progression step* of 1 second indicates that a new window is created each second. Thus, for a time window with length 6 seconds and progression step 2 seconds there will be three concurrent windows (i.e., $6/2$). Hence, each event will take part in 3 windows. Once a window is closed the operator will run and consume all events in that window.

The progression step for a count window is defined in number of events. Thus, a count window defined with length 10 and progression step 2 implies that 5 windows will exist simultaneously and that an event will have taken part in 5 windows before being discarded. Again, the operation will be executed whenever the window is closed.

Synopses

Event windows addressed the problem of restricting the state of the operator by restricting the number of events to be considered in computations. However, there are cases where the number of events in a window still is too large. For example, in a network monitoring application we may want to compute some metric that considers the events (e.g., headers of incoming packets) seen by a router in the last hour. In this case, the window of events to be considered is one hour. Nevertheless, if the router is connected to gigabit links it may be impractical to store all headers and to consider them in a computation. For such cases, using approximations of the metric can be more practical and still satisfactory. Thus, instead of considering all events that occurred in the time window of one hour, the metric may consider only a *synopsis* of these events.

The term synopsis refers to a summarized representation of a set of events. The goal of a synopsis is to represent a set of events in a reduced space, but still with enough information to compute the operation in hand. Therefore, the exact type of synopsis used by an operator depends on the operator itself. The key properties of a synopsis to be used in ESP are [AY07]: broadness of applicability (how easy is to use the same synopsis for different operations); one-pass constraint (the construction of the synopsis should require only one pass over the events); efficiency in construction time and used space (be sublinear in the number of items it represents, ideally have a fixed cost); robustness (provide error guarantees, some synopses can be optimal when globally analyzed, but poor when estimating single points).

There are many methods for summarizing events. The following are the most commonly used ones: sampling, histograms, wavelets, and sketches.

Sampling is the simplest form of summarizing streams. A commonly used sampling approach for streams is the *reservoir-based sampling* [Vit85]. This approach uses a storage with fixed size n and adds events to this storage with a continuously changing probability in order to keep the sample representation unbiased. The first n events seen are added with probability 1. After that, the t-th event is added with probability n/t and, if an event is added, a random event from the storage is removed to keep its maximum size of n. An advantage of sampling is its applicability. Because it does not change the representation of the data, it is easy to adapt operations to use sampling. In addition, it has provable error guarantees as events have equal probability of being considered [AY07]. As a disavantage, it is not adequate when rare events may be very relevant as they may not be even seen by the operator. The pseudocode for implementing such a sampling is shown in Listing 2.1

```
1  list_t r;
2  int t = 0; // counts how many events were seen

4  // Initialize reservoir
5  init_reservoir() {
6     r = list_create();
7  }

9  // Update reservoir
10 insert(x) {
11    if (t++ < MAX_SIZE) add_to_list(r, x, 0);
12    else if (random() < (MAX_SIZE/t)) { // 0 < random() < 1
13       list_remove_index(r, random()*MAX_SIZE);
14       list_add(r, x, 0);
15    }
16 }
```

Listing 2.1: Reservoir-based sampling using a simple list (see Listing 2.2).

Histograms are also a simple representation, but they work only for simple applications (e.g., using static histograms) because maintaining optimal histograms requires super-linear space and time [AY07]. *Wavelets* are able to keep hierarchical representations of the data. In other words, a wavelet synopsis maintains simultaneous representations for different levels of details. However, the one-pass requirement of event streams complicates the construction of wavelets that are both space efficient and robust with respect to the error guarantees.

Finally, sketches are projections of a part of the stream (e.g., a large window) on a reduced space. Some sketch structures (e.g., [CM05, CCFC04]) are able to keep a fixed-size representation of the stream and to process updates in constant time. In this case, the error margins will depend on the number of events considered and the amount of space. In Listing 2.3 we shown an example of such a sketch synopsis, which we will later use in our example operators.

The sketch algorithm shown above is named *count sketch* [CCFC04]. The intuition for its workings is as follows (for proofs and more details see [CCFC04]). The goal is to estimate the number of occurrences n_i of events that have the same type as e_i. The sketch basically keeps

```
// Create a new list
list_t list_create();

// Insert the element at the specified position of the list
list_add(list_t list, element_t element, int position);

// Remove the element at the specified position of the list
element_t list_remove_index(list_t list, int position);

// Retrieve the element at the specified position of the list
element_t list_get_element(list_t list, int position);

// Query the size of the list
int list_get_size();
```

Listing 2.2: Interface of a simple list.

```
int matrix[MAX_HEIGHT, MAX_WIDTH]; // Synopsis
int count = 0;

// vector of hash functions, mapping from "int" to "int"
int hash[MAX_HEIGHT](int);

// vector of hash functions, mapping from "int" to +1 or -1
int bin_hash[MAX_WIDTH](int);

// Update matrix, returning an estimate for the number of occurrences
count_sketch_insert(x) {
   int i;
   int estimates[MAX_HEIGHT];
   for (i = 0; i < MAX_HEIGHT; i++) {
      int column = hash[i](x) % MAX_WIDTH;
      matrix[i, column] += bin_hash[column](x);
      estimates[i] = matrix[i, column];
   }
   return median(estimates)*bin_hash(x);
}
```

Listing 2.3: Sketch for summarizing the number of occurrences of an element in the stream [CCFC04].

a counter c that is added $bin\_hash(e_i)$ everytime an event e_i is received, where $bin\_hash()$ is a pair-wise independent hash function that maps from the domain of e_i to $+1$ or -1. The main idea is that Equation 2.1 gives an approximation for n_i.

$$estimate = c \cdot bin\_hash(e_i) \tag{2.1}$$

To understand why, note that given c and $estimate$ as shown in Equations 2.1 and 2.2, the expected value of the estimate is given by Equation 2.3. Note also that when $j = i$ in Equation 2.3, the product of the two hashes will be $+1$. But when $j \neq i$, the product will be approximately as many times positive and as it will be negative because the hash function is pairwise independent. Therefore, the positive and negative values are expected to cancel themselves and the expected value is n_i, the number of occurrences of e_i.

$$c = \sum_{j=1}^{m} n_j \cdot bin\_hash(e_j) \tag{2.2}$$

$$\begin{aligned} E(estimate) &= E(c \cdot bin\_hash(e_i)) \\ &= E(\sum_{j=1}^{m} n_j \cdot bin\_hash(e_j) \cdot bin\_hash(e_i)) \qquad (2.3) \\ &= n_i \qquad (2.4) \end{aligned}$$

The problem with the estimation above is that the variance of the estimate is very large. To reduce the variance, we could keep several counters and use different hash functions for each of them and, then, when computing the estimate, take the average of these counters. Nevertheless, very frequent types (where n_j is big) would still heavily interfere with low-frequency ones. To solve this problem, each counter can be replaced with a hash table. We now have a matrix, each line is a hash table and each event updates only one entry in each line. Consequently, as long as different lines use different hashes (to implement its hash table), different interferences will occur in each line. Finally, the estimate n_i is computed by computing the estimate for each line and then taking the median of all estimates.

2.1.2 Types of operators

Because ESP systems have much of its origins from the database domain, it is also common to consider ESP operators as special versions of database operators. In ESP, operators can implement one of the following basic functionalities.

- **Filter**: forwards or discards an event based on the evaluation of a predicate on that event. A filter can be stateless or stateful. A stateless filter could be, for example, "drop event e if $e.value < 10$". Alternatively, eliminating duplicated events, for example, would require a stateful filter. Given an input event e and a user-provided predicate $P()$, the output event e' is given by:

$$e' = \begin{cases} e, & \text{if } P(e) = true \\ \varnothing, & \text{otherwise} \end{cases} \tag{2.5}$$

- **Map**: maps one input event into one or more different output events. This family of operators include format conversions, cryptographic signatures, and function computations over attributes of the input events (i.e., transformation of events). As with the filters, map operators may be stateful if the computation depends on previous events. Given an input event e with type T and a user-provided function $f : T \longmapsto T'$, the output event e' with type T' is given by:

$$e' = f(e) \tag{2.6}$$

- **Aggregate**: combines information from multiple events into a single event. Typically, the resulting output event summarizes the input events. For example, an `Average` operator may output the average of a set of input events. Observe that because the stream of input events is assumed to be infinite, these aggregates are not computed over the whole stream, but only over a finite subset of events, namely, event windows. Defining an aggregate requires the following parameters: an event window W containing all the events that will be combined (all with type T), including the current input event e_i; a user-provided function $f : T^n \longmapsto T'$ that maps n events into a single event e' with type T'; and a user-provided predicate $P()$ that evaluates to *true* whenever an output should be produced. The output event e' with type T' is then given by:

$$e' = \begin{cases} f(e_0, e_1, ..., e_n) \mid e_j \in W(0 \leq j \leq n), & \text{if } P(W) = true \\ \varnothing, & \text{otherwise} \end{cases} \tag{2.7}$$

- **Join**: combines two events that come from two different streams and together satisfy a predicate into a new event combining information from both source events. Defining a join requires the following parameters: two windows of events W_1 and W_2 (for streams S_1 and S_2, respectively) with the events that will be combined, including the current input event e_i; a user-provided function $f : T^{S1} \times T^{S2} \longmapsto T'$ that maps two events with type T^{S1} and T^{S2}, for the events arriving in streams $S1$ and $S2$, into a single event e' with type T'; and a user-provided predicate $P()$ that evaluates to *true* whenever an output should be produced. The output event e' with type T' is then given by:

$$e' = \begin{cases} f(e_1, e_2) \mid e_1 \in W_1 \text{ and } e_2 \in W_2, & \text{if } P(e_1, e_2) = true \\ \varnothing, & \text{otherwise} \end{cases} \tag{2.8}$$

2.1.3 Our prototype system

We have built a prototype system to evaluate the concepts shown in this dissertation. This prototype is a framework built in the form of a static library, which is compiled with the files that define the application. The library contains basic functions for creating, transporting, and

destroying events, helper functions for defining and connecting components, as well as accessory functions and the speculation support that we will detail later.

In our prototype the specification of an operator is composed of three functions: an `init()` function that is called on system startup and is responsible for allocating and initializing resources for the operator; a `process()` function that is called for each event received by the operator; and an `exit()` function that can be used to shutdown or release resources and is called during system tear down. The functions are regular C code augmented with special library calls (e.g., for event allocation and generation).

In Listing 2.4, we present a simplified version of the main file of our example application. This file specifies the `main()` function, which will be compiled as a usual C program. The file also includes headers containing the interfaces of the components (or implementation, detailed below) and defines all the components by specifying the functions that should be called for initializing the component, for processing inputs, and for shutting down the component. The definition of components (lines 25 to 33) contains also configuration flags which may indicate some special behavior or requirement. For example, in the listing, the `Source1` component is declared using the `CP_INPUT_ADAPTER` flag, which tells the framework that this component is indeed a source. Similarly, the `Processor1` component specifies the `CP_ORDERED_INPUT` flag, which forces events from the same source to be delivered in order (i.e., the component is order sensitive). At last, we specify the placement of the components. In the example, we place one of the sources in the machine with IP 192.168.1.1.

In Listing 2.5 we exemplify the details of a source component. Different source components can use the same or different code and will execute in different processes. In this listing, the initialization function is used to create a connection with the middleware that transports stock market events. Then, the processing function continually gets stock market messages and creates the respective output events. Finally, the termination function shuts down the connection with the stock market middleware.

In Listing 2.6 we show the details of the *top-k* operator. This operator is a stateful filter that drops events that are not among the k most frequently seen events. It is based on the count-sketch synopsis and the list interface presented earlier (see Listings 2.2 and 2.3). It works by keeping two data-structures in its state. The first one is a count-sketch synopsis that stores a bounded representation of the frequency of all events seen so far. The second data-structure is a list with the top-k events seen so far. When a new event arrives, it is inserted in the count-sketch data-structure. Next, the algorithm checks whether the event is in the top-k list. If this is the case, its frequency counter is updated. The algorithm also keeps track of the event with the lower frequency count that still is in the top-k list. Then, if the estimated frequency of the new event becomes high enough and it is not yet in the top-k list, the least frequent event in the list will be removed and the new event will be inserted. In both cases, if the event is or becomes one of the top-k events, it is forwarded together with its frequency estimate. Otherwise, it is silently dropped. In a practical system this operator would probably consider a window of events instead of the whole stream, but we omit the window details here for simplicity.

If components have more than one input, the inputs streams are implicitly merged. Nevertheless, join operators are still simple to implement, as shown in Listing 2.7. The example join operator is customizable by providing three functions: (a) *expired*() evaluates if an event is

```
#include <stdio.h>
#include <epf_component.h> /* Framework functions and macros */

#include "types.h"   /* Definition of event types */
#include "source.h"  /* Specification of the source interface */
#include "filter.h"  /* Specification of the filter interface */
#include "processor1.h"
#include "processor2.h"
#include "distributor.h"
#include "sink.h"

/* Declare, connect and configure components */
int main(int argc, char **argv) {
  component_t *source_1, *source_2, ...;
  component_t *filter_1, *filter_2;
  component_t *processor_1, *processor_2;
  component_t *distribute, *sink_1, *sink_2;

  init_epf(argc, argv); // framework processes command line arguments

  printf(" Starting application.\n");

  epf_set_message_level(1); // how verbose should the framework be

  source_1 = epf_declare_component("source1", source_init, source_process,
      source_exit, CP_INPUT_ADAPTER);
  source_2 = epf_declare_component("source2", source_init, source_process,
      source_exit, CP_INPUT_ADAPTER);
  ... // other source definitions omitted

  filter_1 = epf_declare_component("filter1", NULL, filter_process, NULL, 0);
  filter_2 = epf_declare_component("filter2", NULL, filter_process, NULL, 0);

  processor_1 = epf_declare_component("processor1", processor1_init,
      processor1_process, processor1_exit, CP_ORDERED_INPUT);
  ... // other definitions omitted

  epf_connect(source_1, filter_1);
  epf_connect(source_2, filter_1);
  epf_connect(filter_1, processor_1);
  ... // other connections omitted

  epf_set_address(source_1, "192.168.1.1");
  ... // other placements omitted

  start_epf(); // start the execution

  return 0;
}
```

Listing 2.4: Simplified specification of our example application.

```
int source_init() {
  ... // Connect to external stock communication middleware
  return EPF_SUCCESS; // component initialization successful
}

int source_process() {
  event_t *ev;
  stock_message_t *msg;

  EPF_ALLOCATE_EVENT(ev, event_t);
  msg = get_and_parse_stock_message();

  ev->stock_name = msg->name;
  ev->stock_value = msg->value;
  ...

  EPF_GENERATE_EVENT(ev);
  return EPF_SUCCESS;
}

void source_exit() {
  ... // Shut down connection
}
```

Listing 2.5: Pseudocode for the sources.

```
list_t top_k_list;

// Filter events that are not among the top-k most frequent
process(ev) {
   int estimate = count_sketch_insert(ev);
   int i, smallest_count = MAX_INT, smallest_index;
   for (i = 0; i < get_list_size(top_k_list); i++) {
      if (list_get_element(i)->object == ev) {
         list_get_element(i)->counter++;
         EPF_GENERATE_EVENT(ev, list_get_element(i)->counter);
         return;
      }
      if (list_get_element(i)->counter < smallest_count) {
         smallest_count = list_get_element(i)->counter;
         smallest_index = i;
      }
   }
   // Not yet in the list
   if (smallest_count < estimate) {
      list_remove(top_k_list, smallest_index);
      list_add(top_k_list, create_element(ev, estimate), 0);
      EPF_GENERATE_EVENT(ev, estimate);
   }
}
```

Listing 2.6: Top-k operator based on a count-sketch synopsis (see Listing 2.3).

```
1  list_t lists[2]; // Join windows for each of the channels

3  join_init() {
4     list[0] = list_create(); // assuming two sources for the events
5     list[1] = list_create(); // one with source_id 1 and other with 0
6  }

8  remove_expired_events(list_id) {
9     event_t ev;
10    bool finished;
11    int pos;

13    while (true) {
14       pos = list_get_size(lists[list_id])-1;
15       ev = list_get_element(pos);
16       // expired() depend on the window types (e.g., time or count)
17       if (ev != NULL && expired(ev)) {
18          list_remove_index(lists[list_id], pos);
19       } else {
20          return; // all expired elements removed
21       }
22    }
23 }

25 process_join(ev1) {
26    int i;

28    remove_expired_events(0); // Remove expired events from window 0
29    remove_expired_events(1); // ... and also from window 1

31    list_add(lists[ev1.source], ev1, 0);
32    list other_list = lists[(ev1.source+1)%2];
33    for (i = 0; i < list_get_size(other_list); i++) {
34       ev2 = list_get_element(other_list, i);
35       // predicate() and generate_joined_event() are user-defined
36       if (predicate(ev1,ev2) == true) generate_joined_event(ev1,ev2);
37    }
38 }
```

Listing 2.7: Sample structure for join operator.

already expired (e.g., it has been in the window for more than the window length); (b) *predicate*() evaluates if the join condition is satisfied (e.g., *ev*1 and *ev*2 have a specific attribute value); finally, (c) once the predicate is satisfied with events that are not expired, *generate_joined_event*() selects which attributes from the joining events will be present in the output event.

2.2 Software transactional memory

In this section, we give an overview on how a conventional STM works. This discussion is based on TinySTM [FFR08] (version 0.7.3), which was used as a base for the implementation of our speculation mechanism. TinySTM is an open-source STM implementation in C. Nevertheless, the general ideas are valid for many other implementations of word-based transactional memories[2].

2.2.1 Overview

STM was introduced as a synchronization mechanism that is easier to use and potentially more scalable than locks. Consider a fragment of code that accesses shared memory. With locks, this code is surrounded by lock acquisition and release. Thereby, it is guaranteed that only one thread executes this *critical section* at a time. With STM, the fragment of code is surrounded by the start and end delimiters of the transaction. The STM guarantees that the actions in the transaction are executed in a way that appears to be atomic from the point of view of other transactions. One major difference between memory transactions and critical sections protected by locks is that transactions are typically optimistic. Threads may execute the transactions concurrently and the STM monitors memory accesses directed at shared memory locations. If the threads indeed access the same positions, they conflict and one of them may need to roll back and reexecute (i.e., *abort*). Otherwise, if their accesses do not conflict (e.g., threads accessed different entries of the same array) both can complete the transaction (i.e., *commit*). Thus, instead of having the programmer explicitly lock individual memory locations in a conservative manner, STM provides automated fine-grained runtime control.

The possible statuses of a transaction are shown in Figure 2.1. A transaction is *idle* when it is created and becomes *active* when it starts processing. Then, it may be aborted or committed. If it is *aborted* its actions are undone and it restarts, becoming idle again. Otherwise, if it is *committed* its effects are permanently incorporated into main memory and the transaction is considered finished.

In our case, TinySTM isolates the execution of a transaction by intercepting shared-memory accesses (i.e., reads, writes, memory allocations and releases). Intercepting the memory accesses is done by having a compiler pass that replaces regular memory accesses by STM accesses when the target address is a potentially shared memory position (for TinySTM, this can be done with Tanger [FFM+07]). At the end of the transaction, the STM validates the execution, i.e., all reads and writes must be consistent so that the complete transaction appears to execute atomically. If validation is successful, the transaction completes. Otherwise, it is rolled back and retried.

[2]Word-based transactional memories access memory with word granularity, in contrast to object-based transaction memories, which use object granularity.

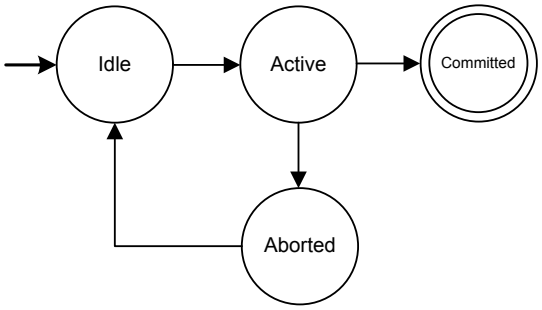


Figure 2.1: Possible transitions for the status of a transaction.

Because only memory changes are rolled back, most STMs do not permit the execution of actions with external effects such as I/O within a transaction.

TinySTM has two modes of operation: write-through and write-back. With the *write-through* approach, memory writes are intercepted and the original values are copied before locations are updated for the first time. Successive accesses to an updated location will directly read and update the modified values. In case the transaction is aborted, the copied value is used to restore the location to the original value. This approach has the drawback of exposing intermediary values from non-committed transactions to non-transactional code.

When using a *write-back* STM, memory writes within a transaction are intercepted and redirected to a private copy specific to that transaction, leaving the original values intact. Read accesses to a previously updated locations are similarly redirected to the private copy. Later, if the transaction commits, the original values are overwritten by the local copies. In contrast to the write-through approach, write-back keeps the memory consistent, that is, memory position contain only committed values. In what follows, we assume that the STM uses a write-back approach.

2.2.2 Memory operations

In order to understand how the memory operations are implemented, and how they will be later extended, it is important to understand the major elements of the STM. As mentioned previously, the write accesses are redirected to a storage private to the transaction. This storage is called *write set*. The write set contains the addresses of the memory positions that a transaction modified during its execution and their new values. Similarly, when a transaction reads a position, it adds an entry to another set, the *read set*. The information in the read set is used to confirm that the transaction considered a consistent snapshot of the memory. For that purpose, each memory position has a *version* number. Each time a modification to a memory position is committed, the version number is changed to the timestamp of the transaction that last modified the position. When a transaction reads a position, the version of that read value is stored in the read set.

For memory positions that were not modified by currently active transactions, the versions numbers are stored in a table named *lock array*. Because it would be too costly in terms of space to keep version numbers for each memory position, several memory positions share a single

version counter. In order to minimize the impact of this sharing, a hash function is used to map the position of memory words into entries in the table.

On the contrary, for memory positions that were written by a currently active transaction, the version number for that memory position is replaced by a pointer to a *lock* object. The distinction between a pointer to a lock and a version number is done by setting the least significant bit in the written value. The least significant bit is then ignored when the pointer is referenced or when the version number is used.

While the position is locked, the version number is moved and kept into a field in the lock. If the transaction later aborts, this version number is moved back to the table. If the transaction commits, the lock will be replaced by the new version number. As mentioned above, the version is the timestamp of the transaction, which in its turn is derived from the number of (update) transactions that have previously committed. Once the lock is replaced by a version number, the position is unlocked.

The main elements of an STM are summarized in Table 2.1. Based on these key elements of an STM, the pseudocode in Listing 2.8 illustrates the major steps involved in a write operation. The first step is to check whether the destination memory position of the current write operation is already locked. If the position is locked by the transaction itself, it simply updates the value for that position in its write set. If the position is locked by another transaction, the current transaction aborts and restarts the computation in the hope that the other transaction will finish soon. Finally, if the position is not locked, a lock is created and placed in the lock array, and an entry is added to the transaction's write set.

Element	*Description*
Current time	Logical time representing the number of transactions committed so far.
Lock array	A table containing either a lock or a version number for a memory position (in fact, a set of positions); each memory position is mapped to an entry in this table by a hash function.
Lock	An entry in the lock array (acquired during write operations).
Read set	The set of positions previously read by a transaction (with the corresponding version numbers).
Status	The current status of the transaction (idle, active, aborted, or committed).
Timestamp	Logical timestamp of a transaction, computed based on the current time at the moment of commit.
Version	The timestamp of the last transaction that modified a specific memory position.
Write set	The set of positions written by a transaction (with the values last written and the original version numbers).

Table 2.1: Key elements of the STM.

The pseudocode in Listing 2.9 illustrates an STM read operation. After realizing that a memory position is locked, the transaction checks if it owns the lock. If this is the case, it reads the value that it has previously written. Otherwise, another transaction has the lock and the current transaction aborts and retries. If the position was not locked, the value and the version number of that memory position are retrieved in a single step. In addition, the transaction must check if the version read can be used. This check consists in testing if the version number is within the

```
1  stm_store(tx, addr, value) {
2     restart:
3     lock = get_lock(addr); // returns either a lock or a version
4     if (lock.locked) { // is locked?
5        if (lock.owner == tx) {
6           update_writeset(tx, addr, value); // we wrote to it before
7        } else {
8           self_abort(tx); // conflicted, let the other proceed
9        }
10    } else { // Not locked, lock now
11       lock2 = create_lock(tx, get_version(lock));
12       replace_lock(lock, lock2);
13       add_to_writeset(tx, lock2);
14    }
15 }
```

Listing 2.8: Writing to a memory position through the STM.

validity interval of the transaction and, if not, if the validity interval can be extended to allow the read. The validity interval and its extension are detailed below. Finally, if the check fails, the transaction aborts and restarts. Otherwise, an entry is created in the read set and the requested value is retrieved.

As just mentioned, each transaction has a validity interval. The goal of this interval is to guarantee that a transaction only sees a *consistent snapshot* of the memory, i.e., a collection of values that would have indeed coexisted if the effects of transactions would really be atomic. The validity interval is defined by start and end attributes. When starting a transaction, both fields are assigned the *current time*. When a transaction reads positions in memory with versions lower than the start of its own validity interval, the transaction is allowed to proceed. Intuitively, this check indicates that any modification to that positions occurred before the transaction started and, therefore, they all existed when the transaction started. However, when a transaction reads a position for which the version is higher than the end of the interval, this means that some other transaction modified this position while the current transaction was already running. It becomes then necessary to check if the new version coexisted with the versions from other positions previously read. If these other positions are still valid to the current time (i.e., they were not updated since the transaction read it), the validation is successful and the validity of the transaction can be extended until the current time. Otherwise, if any of these other positions have already been updated by a concurrent transaction, validation fails and the transaction aborts.

Examples of valid and invalid snapshots are shown in Figure 2.2. Each timeline shows the history of modifications for a memory position, with the numbered marks meaning when new versions were installed. First, consider hat transaction tx_A starts in time 1 (the validity interval is $[1;1]$). It then immediately reads position pos_1 from the memory, which has version 1. Then, it reads pos_2, with version 2, and must validate and extend its validity. Because at that point, pos_1 was not yet modified. Validation succeeds and the validity interval is extended to $[1;2]$. After that, tx_A reads pos_3 with version number 1, which is already inside the transaction's validity interval. Now consider transaction tx_B, which starts at time 3. It reads pos_1 with version 3 and its

```
1  stm_load(tx, addr) {
2     restart:
3     lock = get_lock(addr); // returns either a lock or a version
4     if (lock.locked) { // is locked ?
5        if (lock.owner == tx) {
6           value = get_value_from_wset(tx, addr); // tx wrote to position
                before
7           return value;
8        } else {
9           self_abort(tx); // conflicted, let the other proceed
10       }
11    } else { // not locked
12       (version, value)  = get_version_and_value(lock, addr);
13    }

15    if (version > tx.end) { // is tx allowed to use this version?
16       if (extend_validity(tx) == FAILED) // no, try to extend
17          self_abort(tx); // extension failed
18    }
19    add_to_readset(tx, addr, version); //yes, it is allowed, proceed
20    return value;
21 }
```

Listing 2.9: Reading a memory position through the STM.

validity interval is [3; 3]. It then reads pos_2 with version 2, which is still lower than the validity interval's end. tx_B finally reads pos_3, which has version 5, and tries to validate its previous reads and extend its validity interval. However, this time, the value previously read for pos_2 (which had version number 2) is not anymore current (it now has version number 5). This indicates that the version number 2 of pos_2 and version number 5 of pos_3 would never have coexisted if effects of all transactions actually had taken place atomically. Transacton tx_B has to abort and reexecute.

When a transaction runs to its end without aborting, it tries to commit. This will succeed if, at commit time, all positions read by the transaction form a consistent snapshot. Then, to avoid unnecessary (and potentially long) validations, a check is made to verify if another transaction has committed since the current one started. In our example from Figure 2.2, consider tx_A tries to commit at real time t and that the (logical) current time is 3. The validity interval of tx_A is [1; 2], which means a start time of 1. Then, because current time is 3 it means that some other transaction committed changes to memory after tx_A started and some of its read values could had been modified since its last validation. During the validation, the STM detects that pos_1 has now version number 3 and, thus, tx_A used an old version and must reexecute.

In addition to memory reads and memory writes, the STM also keeps track of memory allocation and memory releases. Memory allocations are executed immediately and the allocated positions are later released if the transaction aborts. Memory releases are delayed until the end of the transaction. They are executed if the transaction commits, or discarded, if it aborts.

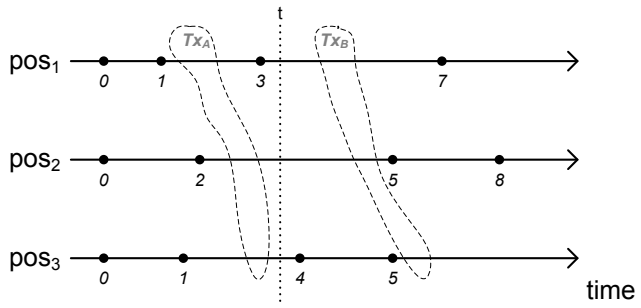


Figure 2.2: Examples of transactions using a valid (tx_A) and an invalid (tx_B) snapshot.

2.3 Fault tolerance in distributed systems

Fault tolerance is a basic requirement for any system used in critical applications. The definition of a critical application is broad and even systems used exclusively for leisure can be considered critical if their unavailability may cause financial losses to the provider (e.g., because customers change providers or forgo using the service). Nevertheless, a distributed system is by definition a set of nodes that cooperate to carry out a common task; the more nodes compose the system, the higher is the probability that at least one will fail, potentially compromising the whole system if no special care is taken. In the next subsections, we review fundamental concepts in fault tolerance that will be relevant in later chapters.

2.3.1 Failure model and failure detection

The first important item to be addressed when considering fault tolerance is the failure model. In practice, there are two commonly used failure models. On the one hand, the *crash-failure model* [VR01] assumes that nodes fail by halting. In addition, when recovered, nodes come back with different ids. Thus, this is equivalent to having permanent faults. On the other hand, in the *arbitrary-failure model* [VR01], nodes may behave outside their specification in arbitrary ways. They may halt as in the crash failure model, proceed too slow or too fast (timing failure), produce wrong results, and even produce specially-crafted wrong results that can cause other nodes to fail.

When choosing a failure model, it is important to choose the narrowest failure model possible as this greatly reduces the complexity and cost of the system. For example, assume that nodes can be arbitrarily slow and excessive slowness should be detected as a failure and handled. This kind of failures clearly includes crash failures (i.e., nodes can be infinitely slow) and, thus, appear to require a broader failure model. However, in the arbitrary-failure model, simple problems like having distributed nodes to agree on a single value (i.e., consensus [CT96]) will require $3 \cdot f + 1$ nodes in order to tolerate f failures [LSP82]. Nevertheless, it may be possible to use techniques to transform these timing failures into crash failures. For example, local hardware watchdogs could be used to crash nodes that are slower than a threshold [Fet03]. Using this transformation the crash-failure model could be used and then, $f + 1$ nodes suffice to solve the

agreement problem stated above. Similarly, value failures caused by hardware errors (e.g., bit flips) can be detected by encoded processing techniques [WF07] before any output is produced. This can be used to effectively transform value failures into crashes. These two examples argue that although simplistic, the crash-failure model is useful in many practical scenarios.

In the portion of this work that addresses fault tolerance, we assume the crash-failure model. Then, to detect crashes we consider a failure detector based on the one proposed by Fetzer [Fet03]. The failure detector works as follows. The protocol considers a system with three nodes and implements a failure detector that is able to detect one failure and never suspects a node that did not fail. Consider initially the system with nodes p_1, p_2, and p_3, each equipped with a watchdog (either a hardware watchdog or a software watchdog as available in common Linux distributions) and with a local clock with a bounded drift rate ρ from real time. Each watchdog is programmed to force the local nodes to crash (or restart) if the watchdog is not reset before $T \cdot (1 + \rho)$ time units passed. Besides that, nodes have to acquire a lease from at least one of the other nodes to be allowed to reset their watchdogs. Finally, when granting or requesting leases, nodes exchange a list of other nodes they directly granted a lease.

To illustrate how the protocol above detects failures, assume node p_1 is granted a lease from p_2. If either p_1 or p_2 has granted node p_3 a lease in the last $T \cdot (1 + 2 \cdot \rho)$ time units (according their local clocks), both p_1 and p_2 learn that p_3 may be still alive. However, if neither p_1 or p_2 granted node p_3 such a lease, the node crashed (or was forced to crash by its watchdog, as it did not have a lease). This protocol can be extended to work with more than three nodes by executing multiple concurrent 3-node instances. A node is then considered failed when all the instances that include that node detect it as failed. On the contrary, a node is able to reset its watchdog as long as it is able to acquire a lease in at least one of the instances.

2.3.2 Recovery semantics

A system that is able to recover from failures may offer different types of recovery, depending on the amount of information that is guaranteed to survive a failure. In this work, we focus on precise recovery. *Precise recovery* means that failures are completely masked with respect to semantics. Results generated after a failure will be exactly the same as if no failures had occurred. The only possible visible effect of a failure is then a slight decrease in performance.

Alternatives to precise recovery are *rollback recovery* and *gap recovery* [HBR+05]. Rollback recovery guarantees that no input information is lost, all input events are considered, even in case of failures. This definition implies also that accumulated state is preserved. However, in contrast to precise recovery, the execution after a failure may follow different paths and, thus, achieve different decisions. Finally, gap recovery accepts that inputs and accumulated states are lost due to failures. In this case, after a failure, a system providing gap recovery is allowed to restart with a fresh state and start processing from the latest input.

2.3.3 Active and passive replication

In later chapters, we address fault tolerant approaches for ESP systems. There are two classic approaches for fault tolerance. The first option is to have the relevant state of the application being periodically saved in stable storage. A *stable storage* is a storage that is able to survive

the failures that are expected in the system. The second option is to have multiple nodes in lock step so that all accumulate the same state. The state will be stable as long as tolerated failures do not affect all nodes simultaneously (i.e., nodes fail independly). These two options describe the basic approaches for fault tolerance, namely, *passive* [BMST93] and *active replication* [Sch90], respectively.

Choosing between passive and active replication requires considering several factors. On the one hand, active replication deals with failures by having redundant nodes, named replicas. Replicas repeat computations and output equivalent results. Because outputs from all replicas are equivalent, any of these can be used and the others ignored. Therefore, failures are transparently masked. A clear advantage is that there is no recovery phase. For the same reason, a clear disadvantage is the amount of resources wasted in failure-free runs. Another disadvantage is that computations need to be deterministic as in a state machine. The next state of the computation must depend exclusively on the current state and the current input. Hence, active replication is also known as the state machine replication approach. Requiring determinism affects several common operations like reading wall-time clocks and the use of multithreading (if the result of these operations may affect the state of the replica). In addition, the state transitions are likely to depend on the order of the inputs, requiring totally ordered communication protocols (e.g., atomic broadcast [CASD95, CT96]) to be used to ensure all replicas process the same messages in the same sequence. Ordered broadcasts are much more expensive than simpler ones (e.g., reliable broadcast [VR01, CT96]) as they require coordination among nodes, incurring extra resource costs to the system.

On the other hand, passive replication [BMST93] and, similarly, rollback-recovery approaches [EAWJ02] deal with failures by having the state of the nodes being periodically checkpointed in a stable storage. For example, if nodes are expected to fail only due to software crashes (either from processes or from the operating system) and are expected to recover after failing (e.g., by having a watchdog that reboots the system if it hangs), then the local disk can be considered a stable storage if synchronous writes are used. Alternatively, if, for example, hardware failures may compromise nodes permanently, stable storage requires writing to a non-local storage (e.g., a disk or memory of a remote node).

In addition to checkpoints, passive replication can also use logging between checkpoints to save information that is relevant for replay [EAWJ02]. When a checkpoint is taken the logs can be cleared. For example, nodes may do periodic checkpoints of the complete state and, between checkpoints, log the messages they process, possibly together with any nondeterministic decisions taken, such as clock values read and scheduling decisions. On recovery, nodes restore the latest checkpoint and then replay messages from the log.

A clear advantage of passive over active replication is that there is no redundant work being done and, thus, less resources are used. In addition, nondeterministic decisions are allowed as long as they can be saved to the log and enforced on replay. As disadvantage, passive replication requires a recovery phase that restores the checkpoint and replays logs, which can take considerable time.

2.4 Summary

In this chapter, we discussed basic concepts in event processing, transactional memory, and fault tolerance in distributed systems. We first discussed how operators are implemented and how they keep state. After that, we looked into the internals of the STM. Understanding how operators keep state and how the STM protects critical sections of code is the basis to understand how the STM can be used to automatically parallelize some stateful operators.

As we will see later, using a write-back STM enables that high priority transactions quickly abort less priority ones. In addition, a write-back STM keeps the memory consistent and, thus, simplifies the process of checkpointing the state of operators.

We then looked at passive and active replication and the reasons to select one or the other. In the next chapter, we discuss how we address some of the limitations in these two techniques using a speculation that is based on STM.

Chapter 3

Extending event stream processing systems with speculation

In this chapter, we motivate the extension of an event processing system (ESP) system with speculation. We define the goals of speculation and discuss high-level features that a system would have to offer to enable these benefits.

3.1 Motivation

As discussed in the previous chapters, an ESP system is a specialized distributed system. This specialization can be seen, for example, in the communication patterns: data messages flow only in one direction, from the so-called *upstream* nodes to the *downstream* nodes. In addition, the computations executed in each node are simpler than general programs. Isolated operators have reduced functionalities because it is advantageous to factor a complex computation into several stages in order to benefit from pipelining.

ESP systems are a very useful simplification of distributed system. They are useful because they address many applications that are increasingly common. Increasingly more electronic devices behave like distributed data sources producing vast amounts of low-level data. For example, cell phones transmitting their geographic coordinates allow a broad scope of applications, from enabling friends to detect each other to modelling movement behaviors. Because of the continuous growth in the number and sophistication of inter-networked devices, the trend is that there will be ever more generators of low-level data. Consequently, there is a growing need for technology that helps making sense of this data in a timely manner.

Another recent trend regards processor technology. Fundamental problems like the power dissipation limits led to a freeze of the growth of sequential speed of processors. The micro-processor industry has changed directions and progressively migrates from single heavy-weight cores (with deep pipelines, prefetching, speculative execution) to multiple simpler and more efficient, but slower cores. As a consequence of this parallel hardware, there is a growing amount of computational power that is difficult to be harnessed (e.g., see [GN08, ABD+09, JAAS09]).

This dissertation is motivated by the growing attention received by ESP systems and the new

trade-offs that such a new paradigm imposes on known techniques. We propose approaches that extend traditional techniques to exploit these new opportunities. More specifically, we focus on speculation and how it can help improving performance of distributed ESP systems.

3.2 Goals

In this work, we revisit many known ideas in speculation for computation systems, for example: virtual time [Jef85], optimistic recovery [SY85], thread-level speculation [SCZM05], transactional systems [WV01]. We extend them using software transactional memory [ST95] and targetting multi and many-core machines [ABD+09] in order to build low-latency fault-tolerant ESP systems. The goal is to evaluate benefits derived from the use of speculation.

We implement a foundation for speculation: (a) the ability to process events concurrently or out-of-order and to detect whether data dependencies are violated; (b) the ability to capture writes from the speculative processing of an event, providing low-cost rollback when necessary.

On the one hand, we improve the performance of operators through parallel and out-of-order processing by speculating that events close to each other in time do not always have common dependencies. On the other hand, we reduce the cost of fault tolerance in failure-free runs by speculating that failures do not occur and using the efficient rollback mechanism to avoid inconsistent states.

We make the following contributions:

1. **Out-of-order processing:** some operators require events to be processed in order and ordering events coming from different sources requires buffering and introduces latency; speculative out-of-order processing enables the overlapping of this waiting with useful processing.

2. **Optimistic parallelization:** some types of operations are parallelizable, but still, developing parallel operators is expensive and error-prone; speculative parallelization allows parallelism to be discovered and explored at runtime while requiring no modifications to the original sequential code; furthermore, our parallelization does not introduce nondeterminism in the execution, in other words, it preserves *sequential semantics*, which helps developing, testing, and debugging applications.

3. **Speculation in passive replication:** passive replication and rollback recovery require operators to record changes in the state before each output and waiting these recordings to finish adds considerable delays; we show how to build an efficient rollback mechanism based on speculation and how it enables operators to use asynchronous logging; nodes can, therefore, output results before the recording of state changes reaches the disk.

4. **Early processing in active replication:** active replication requires that replicas use message ordering protocols that are costly; we use speculative events to allow replicas to start processing using an optimistic delivery that happens much earlier than the ordered delivery; in addition, we also allow results to be propagated earlier as speculative, enabling downstream nodes to also start early.

5. **Multithreading in active replication:** active replication does not usually supports multithreaded operators because of determinism; we enable multithreading by exploiting the atomic behavior of the speculation mechanism to mask nondetermism caused by multithreading.

3.3 Local versus distributed speculation

In order to enable low-latency ESP systems, we make use of local and distributed speculation. *Local speculation* refers to the use of speculation within a single component (i.e., the optimistic assumption is made on a local hypothesis). Optimistic parallelization is implemented through local speculation, for example. *Distributed speculation* refers to the case in which an operator makes an assumption on a statement being evaluated at another node. For instance, an operator starts a checkpoint asynchronously, but forwards results speculating that no failures will occur; downstream nodes use these results in speculative computations.

On the one hand, when using local speculation, we assume that inputs and outputs of an operator are not speculative. We also require that no output reflects information from incorrect speculations. These requirements establish that no result that was produced by an optimistic computation has any difference to the result that would be produced in the conservative case. In other words, when using out-of-order processing or optimistic parallelization, no speculation is visible outside the operator and the history of modifications to the state is exactly the same as in a sequential operator.

Nevertheless, we can achieve parallelism. Consider the concurrent processing of two events e_1 and e_2 in a stateful operator. If the processing of e_1 does not consider any memory position that was modified by e_2, e_1 and e_2 can be processed in parallel. Similarly, for out-of-order processing, we reduce latency as follows. Consider that events e_1 and e_2 have timestamps 1 and 2, respectively, and that events should be processed in the order of their timestamps. If event e_2 is already available for processing, but event e_1 not, e_2 can be processed before e_1 as long as processing event e_2 does not read (or write to) any memory position that will be written (or read) by the processing of e_1.

Distributed speculation is more flexible than local speculation. It allows operators that still have pending optimistic assumptions to output speculative results to downstream operators. As a consequence, as long as sink operators do not expose speculative results to consumers outside the ESP application, the speculative results can be used to advance computations also in the downstream operators.

3.4 Models and assumptions

In the following sections, we discuss some assumptions we make about the operators, events, and failures.

3.4.1 Operators

When adding speculation capabilities to an operator it becomes necessary to change its model and the assumptions about its behavior. The first aspect to be observed is that the memory operations (read, write, allocate and release) are done through the STM. Similar to conventional STMs, this can be accomplished by a compiler pass that replaces the original accesses with STM wrappers [FFM+07, CCD+10]. Nevertheless, for that to be possible, source code for the complete operator, including used libraries, has to be available. The development of this transactifying compiler is an issue that is orthogonal to the problems addressed in this dissertation. In the current prototype of the speculation infrastructure, the instrumentation of the memory accesses is done manually and implemented through macros that translate either to the original memory access or to the call to the STM wrapper.

We also do not consider binary instrumentation (like in [Wam08]). Therefore, if libraries for which no source code is available are used and calls to functions in these libraries may cause side effects, it is important to consider these calls to be external actions. An *external action* is an action that causes side effects that cannot be automatically rolled back by the speculation infrastructure (i.e., the STM). For example, a call to a library function for which no source code is available, but that may modify some state within the operating system's kernel or a file on disk is an external action. External actions require special treatment and providing as less restrictions as possible to them is also a challenging problem. In this work, we provide means to enable users to provide custom actions to handle external actions.

Due to the nature of speculation, operators may see values in memory that they would never have seen in a nonspeculative execution. For example, in a given application an event may represent a connection being closed. Due to out-of-order processing, it may be the case that this event is processed before the event that represents the connection being open is processed. As a consequence, arbitrary operators may get into inconsistent states that cause problems such as infinite loops, segmentation faults, and state corruption. This requires recovery actions to be programmed around the operators in order to catch such exceptions.

Misbehaviors that cause state corruption are not a problem because computations that corrupt the state will be undone. For other problems, we require the usage of mechanisms that can interrupt the operator (e.g., when the operator executes an infinite loop) and trigger a rollback (e.g., after a segmentation fault). Therefore, a signaling mechanism like Linux signals [BC02] must be available for the speculation infrastructure. Linux signals can be used to catch exceptions and interrupt computations.

3.4.2 Events

An event is a representation of something that happened in the system. Events have unique identifiers that enable distinguishing between two similar, but distinct, phenomenons. However, due to the nature of speculation an event may not be an accurate representation for the phenomenon it refers to. In that case, we classify events as speculative. *Speculative events* were generated by speculative operations and there are some not-yet-confirmed optimistic assumptions that were used to generate them. Because the underlying assumptions may be wrong, speculative events may be reviewed and corrected. Therefore, speculative events share a common id (as they refer

to the same phenomenon), but have distinct versions. A version is a counter that represents how many times the event was reviewed. A higher version of an event can be seen as being less speculative than an earlier version as it was reviewed more times and there are maybe less pending optimistic assumptions.

Events can be also classified as final. *Final events* have the usual meaning. The data in such an event is not going to be later corrected or review. In addition, a final event is the last version for all speculative events with the same id.

Regarding the transport of events, in our system, connections between operators (i.e., the event channels) are static. These are defined in the source files for the event processing application (see Chapter 2.1).

3.4.3 Failures

During our investigation of out-of-order processing and optimistic parallelization in Chapter 4, we assume that no failures occur. Later, in Chapter 5 we allow crash failures and evaluate the use of speculation in passive and active replication. Finally, we show how our system can be extended to consider active replication with a failure model that allows crash and value failures caused by certain classes of common software bugs. We consider only benign software faults. *Benign software faults* include faults that are caused by software bugs that can be detected by a runtime checking approach such as bounds checking [RL04]. We do not address hardware failures and do not consider malicious attacks, in which inputs or operators can be specially crafted to compromise the system. In addition, we assume that there is always one available replica that is able to detect the software fault to be tolerated.

When handling failures, we aim at precise recovery. *Precise recovery* requires that outputs in case of failures are the same as outputs produced if the failure had not occurred (see Chapter 2.3.2). Although some applications can cope with less strict recovery models (e.g., results after after a failure may be inconsistent with results produced before the failure), many applications require precise recovery either because their critical importance (e.g., algorithmic trading) or because high user requirements.

Chapter 4

Local speculation: parallel and out-of-order processing

In this chapter we present and evaluate the use of speculation in the scope of a single operator in a distributed event stream processing (ESP) system. Thus, speculation is not visible outside an operator. Still, speculation is used internally to process events optimistically in parallel or out of their normal order.

4.1 Overview

The graph of operators that composes an ESP application may contain many different types of computations. Some of these processing tasks will need only short stateless computations, others will require persistent state and costly computations. In addition, some of these tasks will be standard operations (e.g., computing a simple moving average, where the only parameters are the type and size of the window), but others will need some custom code (e.g., to aggregate or summarize events according to a custom algorithm).

When operations with high computational costs are used, operators need to be optimized or parallelized in order to allow good system throughput and low processing latency. Parallelizing stateless operators is a trivial task. It can be achieved simply by creating several instances of the same operator and letting them process events in parallel. However, stateful operators are not trivial to parallelize. Basically, the operator designer needs to make sure that the parallel instances will access the shared state in a coordinated form that is in accordance with the operator's specification.

There are many difficulties in parallelizing stateful operators. First, it may not be possible to parallelize an operator in a cost-effective way. Second, the lack of repeatability of parallel programs makes development and testing considerably more difficult.

Manually parallelizing an operator can be costly. If one uses lock-based approaches to synchronize threads, coarse-grained locks (e.g., protecting a complete phase of the algorithm or a data structure, such as a hashmap, with a single lock) are relatively easy to implement, but contribute with little parallelism. However, using fine-grained locking is difficult, because:

(*i*) locks should be acquired only when strictly needed; (*ii*) they should be released as soon as possible; and, (*iii*) the same lock should not be used to protect independent pieces of the state (e.g., two positions that are not always accessed together).

The eagerness to minimize the code area protected by a lock leads to many bugs in software. Consider, for example, two bank accounts in a transfer operation. If all the accounts are protected by the same lock (i.e., coarse locking), no parallelism is available. However, if each account is protected by a different lock and only one account is locked at a time, it is possible that during a transfer from one account to the other, another running thread computes the balances and sees a snapshot where the money is in none of the accounts (i.e., a program invariant could have been broken). Finally, if accounts are protected by different locks and a transfer operations needs to get both locks before executing the operation, two threads that acquire the same locks in different orders may deadlock.

Although the example above is simple and solutions may be intuitive, similar problems become quickly non-trivial if, for example, the invariants are complex or if it is not straightforward to decide the correct order for lock acquisition (e.g., the second lock to be acquired depends on some information learned after the acquisition of the first). In addition to bugs being more frequent in parallel code, this type of code is also more difficult to test and debug. According to a survey among expert developers inside Microsoft [GN08], 75% of the developers interviewed consider concurrency bugs hard or very hard to reproduce. The respondents also stated that they would strongly benefit from better libraries with prepackaged concurrency mechanisms. In a summary, developing a parallel version of an operator may require much more resources then developing a sequencial one.

The other issue with parallelization of operators is the repeatability and ordering of operations. In many domains it is desired that executions are repeatable. For example, in algorithmic trading systems, it is a common requirement that decisions can be later replayed for postmortem analysis [ScZ05]. This is difficult to achieve if the arbitrary interleavings of threads may influence the results. In fact, the difficulty to write, test, and debug parallel code is largely due to nondeterminism in concurrency [BAAS09]. Nevertheless, locks carry no ordering information and, thus, runtime scheduling decisions may cause order inversion in a multithreaded operator.

Repeatability and *in-order processing* (i.e., events are processed in the order of a physical or logical timestamp) are closely related. In-order processing can be used to provide repeatability. For example, if there are many events at the input of an operator, repeatability requires that this operator processes the events according to some predefined order (like according a timestamp attribute in the event). Using the arrival order would not suffice, because in different executions events from different sources can suffer slightly different communication delays and arrive in different orders. Another reason for in-order processing is when the semantic of the computation depends on a timestamp of the event. As an example, events could represent requests for a resource and the resource should be allocated to requests with lower timestamps. Also in this case, different communication delays suffered by two events could lead to an event with a higher timestamp being received shortly before the event with the lower timestamp. The need for repeatability and in-order processing implies that deterministic execution is required, in combination or not with parallelization.

Because of the above reasons, for standard operations to be implemented as part of a library,

it may still be reasonable to pay the extra development costs to provide code with fine-grained locking that preserves ordering when necessary. However, especially when development time is critical, algorithms change frequently, or cost-effective solutions are desired (concurrent programming also requires more experienced programmers), automated parallelization techniques for user-defined operations are needed.

In this chapter, we propose the usage of speculation to allow both automated parallelization and an out-of-order preprocessing that still guarantees in-order, sequential semantics. This is achieved by encapsulating the processing of the events in transactions. Transactions have timestamps derived from the desired processing order (e.g., a timestamp or id attribute from the event) and will be executed using a Software Transaction Memory (STM) that intercepts memory accesses directed at the operator's state. At the end of a transaction, there will be two sets containing all the memory positions that were read or written during the processing of the event. These sets are named read and write set, respectively. When two events are processed out of their normal order, in-order semantics is ensured by asserting that the reads and writes of the two transactions do not intersect. Further, if multiple CPUs are available, parallelization can be achieved by executing two transactions in parallel. In this case, sequential semantics is also ensured by guaranteeing that the read and write sets do not intersect. Optimistic parallelization and out-of-order processing are based on the same mechanisms and can be used in separate or combined.

In the following sections, we first introduce the functionalities required from the speculation infrastructure and how they are implemented. After that, we discuss applications, possible extensions, and results.

4.2 Requirements

4.2.1 Order

Ordered transactions is not a native feature in most STMs. In order to provide this, we need three basic mechanisms: (*i*) ordered commits, (*ii*) priority transactions, and (*iii*) interruptible transactions. Having ordered transactions implies that the effects of the processing of events obey the predefined order. In addition, TinySTM's concurrency mechanism requires that transactions are sequentially committed. This combination provides not only partial ordering, but a total ordering among transactions. For simplicity, we assume initially that timestamps in events (and, consequently, in the transactions that process them) are unique and gapless logical timestamps. We later show how to drop this assumption and consider timestamps in which duplicates and gaps are possible (like with physical timestamps).

First, with the regular TinySTM, write accesses are redirected to the write set of the transaction and just exposed after the commit. Consider a transactions tx_t, created to process the input event with timestamp t. The *ordered-commit* requirement forces that the effects of this transaction can only be made visible after the effects of all transaction with timestamps lower than t are visible. In addition, it is also required that tx_t considers the latest versions (at time t) of all memory addresses it has read.

Second, the *priority transactions* requirement aims at having transactions for events that

are already in-order to have their executions minimally disturbed by transactions processing out-of-order events. The highest-priority transaction is the *next-to-commit* transaction. Because of the total ordering, there is at most one transaction timestamp that is allowed to commit at any time. Ideally, the next-to-commit transaction should not be delayed by lower-priority transactions. Active transactions interfere with each other when they contend for memory positions. Consider two transactions tx_1 and tx_2 that execute concurrently. Both transactions access the same memory position and the timestamp of tx_1 is lower than the timestamp of tx_2. Thus, tx_1 has a higher priority. The possible interference patterns are shown in Table 4.1.

Case	*Contention type*	*Impact on* tx_1	*Impact on* tx_2
1	tx_1 reads before tx_2 has read	None	None
2	tx_1 reads after tx_2 has read	None	None
3	tx_1 reads before tx_2 has written	None	None
4	tx_1 reads after tx_2 has written	None	None
5	tx_1 writes before tx_2 has read	None	Aborts on read
6	tx_1 writes after tx_2 has read	None	Validation eventually fails
7	tx_1 writes before tx_2 has written	None	Aborts on write
8	tx_1 writes after tx_2 has written	Forces tx_2 abort	Aborts on next operation

Table 4.1: Contention scenarios for two concurrent transactions accessing the same memory position.

As shown in Table 4.1, for two concurrent reads (cases 1 and 2) there is no interference. Multiple transactions can read from the same positions concurrently. Then, in cases 3 and 4, if tx_2 had modified the position, because we use a write-back approach, the value in memory is still the original and can be read by tx_1. In case 3, the version number of the value read from tx_1, which is needed later for validation, can be retrieved either from the lock array. In case 4, the version number can be retrieved from the write-set of tx_2 (recall that tx_2 moves the version number from the lock array to its write-set descriptor when it writes, and consequently, locks, a position).

Cases 5 and 6 handle scenarios where tx_1 writes to a position that is read by tx_2. In case 5, tx_2 aborts as soon as it learns that tx_1 has the lock to the position. As we will detail later, tx_2 then waits until tx_1 commits before retrying. In case 6, tx_2 will have already read the value and will realize it did not use the latest version only during a later validation (e.g., during its commit). It will then abort and reexecute using the updated value.

Finally, in case 7, tx_1 locks a positions and later, similarly to case 5, tx_2 tries to access that position and aborts when it sees the lock. In case 8, tx_1 finds a position locked by tx_2 and replaces the lock by its own. In addition, tx_1 has to set a flag in tx_2's transaction descriptor (which is obtained through the lock) to force tx_2 to abort. Note that tx_1 needs to force tx_2's abort because a read from tx_2 after tx_1's commit would not see that tx_2 itself wrote to that position previously.

The third and last needed mechanism is the ability to pause transactions (i.e., interruptable transactions). Normally, a transaction that executes until its end is allowed to commit. In addition, an aborted transaction restarts immediately after being aborted. The ordering constraint detailed above requires that a finished transaction waits until it becomes the next-to-commit transaction. When the transaction becomes the next-to-commit, it revalidates and commits. Similarly, a transaction is aborted because it conflicts with a transaction that has higher priority. Thus, it is

normally not advantageous to reexecute an aborted transaction until the conflicting high-priority transaction commits. These paused transactions are kept in a priority queue, named wait queue. The *wait queue* uses as ordering key the timestamp that needs to be reached before this transaction is tried again. This timestamp used as ordering key can then be the transaction's own timestamp when it is simply waiting to become the next-to-commit. Alternatively, the timestamp can be the next immediate timestamp after the commit time of the conflicting transaction (with higher priority). Once the time used in this ordering is reached, the transaction is allowed to commit or reexecute.

4.2.2 Aborts

A transaction may abort for four reasons:

i) It is about to read from a memory position that does not form a consistent snapshot with previously read values (i.e., the value that is about to be read would have never coexisted with previous reads if all transactions were really atomic).

ii) It is about to read from or write to a memory position that is locked by a transaction with lower timestamp (cases 5 and 7 in Table 4.1).

iii) It realizes it did not used the latest version of a position it has read (case 6 in Table 4.1).

iv) It holds a lock for a position that is to be modified by a transaction with lower timestamp (case 8 in the table).

For reasons *i* to *iii*, the transaction itself realizes it has to abort and reexecute. Then, it first releases all the locks it holds in the lock array and frees all memory allocations executed inside the transaction. After that, it creates a new transaction descriptor and inserts it in the wait queue. It then eventually restarts the execution using this new descriptor.

The old transaction descriptor contains the write-set entries that were created in the previous execution and have to be preserved because some other transaction may have gotten a reference to such an entry during a conflict. As an example, assume tx_2 wrote to memory position p, which led to the acquisition of lock l_p in the lock array. If before tx_2 aborts, another transaction tx_1 tries to access that position, tx_1 will recover the pointer to tx_2's descriptors and examine it to decide which transaction has priority. If the descriptor would have been recycled or freed, it could be the case that the memory pointer is suddenly invalid. Therefore, a quiescence-based garbage collector is used: a transaction descriptor tx is kept alive until all transactions that are currently active were born after tx was aborted or successfully committed and, thus, could not have obtained a pointer to tx.

The last reason for an abort (reason *iv*) is that the required abort is not detected by the transaction itself, but by another transaction that has priority over the current one. The priority transaction forces the other transaction to abort by setting a flag in the transaction descriptor. The next time that the to-be-aborted transaction initiates a memory operation through the STM, it will see the flag and abort itself.

Regarding reexecution, whenever the abort was caused by the direct interference between two transactions (reasons *ii* and *iv*), the current transaction is put aside in an ordered queue, the wait queue, as detailed in the previous section. Next, it waits for the conflicting transaction to commit. If the interference was indirect (reasons *i* and *iii*), the transaction reexecutes immediately after the abort.

4.2.3 Optimism control

During speculative processing, too much optimism can be counterproductive. Bad speculations can waste computational resources and delay, directly (e.g., by contending for the same memory addresses) or indirectly (e.g., by consuming memory bandwidth), more promising computations.

There are two basic ways of restricting speculation: (*i*) *limit the amount of CPU resources* used by speculation and (*ii*) *limit the amount of optimism*. Our speculation mechanism implements the first option by restricting the amounts of threads that can process events. In this case, speculative processing will be carried out as long as there are available inputs and one of the threads is idle. A thread is idle if previous speculations finished processing, even if they are not committed yet (as discussed in the previous sections, in our system transactions may be put aside while waiting for commit).

Controlling speculation by limiting the level of optimism is based on the fact that events have timestamps and should be committed in the order of these timestamps. In this case, as a matter of fact, lower timestamps have a higher chance of success than higher timestamps. Consider, for example, two events e_t and $e_{t'}$, with timestamps t and t', respectively. In general, if $t < t'$, then there are less transactions that will eventually commit before the transaction processing e_t than transactions that will eventually commit before the transaction for $e_{t'}$. As a consequence, the chance that the processing of e_t will conflict with a transaction with higher priority (which would render the computation of e_t useless) is lower than for $e_{t'}$. When considering this approach, a user specifies the maximum distance in clock ticks away from the next-to-commit transaction that the timestamp of an event must have in order for the event to be speculatively processed. We call this maximum distance *speculation horizon*.

The two simple approaches above are static. In order to maximize the speculation efficiency for a new operator, the user has either to experiment or to have a good enough understanding of the operator's algorithm to set the speculation approach and its parameters. In addition, if the opportunities for parallel processing in the operator vary with the workload (e.g., the mix of events that cause the operator's state to be only read and the events that cause updates varies), a static adjustment is both nontrivial and unlikely to be optimal.

To solve this configuration problem, we propose two straightforward controllers. We name these controller modules *conflict predictors* as, ideally, they would regulate the amount of speculation by predicting how likely future transactions are to have conflicts with each other. For both controllers, we enable the adjustment of the maximum number of threads to be used by speculation. Assuming the processing nodes are dedicated, this number can be the number of hardware threads[1] in the machine (as it is counter-productive to have context switches in

[1]Some processors, like the Intel Xeon, have one single thread per core. Others, like the Sun UltraSPARC T1 may execute several threads in parallel at the same core with a total performance that is higher than if only one thread was

CPU-bound computations). In addition, both conflict predictors (or simply "predictors") actuate by dynamically regulating the speculation horizon.

The first predictor uses an *abort-driven* approach as follows: (*i*) the predictor periodically (e.g., for each 100 committed transactions) checks the number of transactions that aborted because of a conflict; (*ii*) if this number exceeds a certain threshold, the speculation horizon is reduced, reducing the amount of speculative computations; (*iii*) otherwise, if the number is below the threshold, the speculation horizon is increased, enabling more speculative work.

The threshold for increasing or reduce speculation in an abort-driven conflict predictor is specified by the user and is a hint of how much processing resources can be wasted by speculation. If during the dynamic adjustments, the speculation horizon reaches 1 (i.e., no speculation), it is periodically set to 2 (i.e., besides the next-to-commit event, one more event is speculatively processed). This enables the system to continuously keep track of the conflicts and quickly detect when parallelism becomes available. A possible optimizations for such cases is to turn off speculation (reducing processing delays by avoiding STM overheads) and only periodically turn it back on to check if parallelism has become available.

The second dynamic predictor uses a *throughput-driven* approach. As with the abort-driven predictor, it runs periodically. At each execution, it measures the throughput of committed transactions. Then, it compares the current thoughput with the throughput measured in the previous execution. In addition, it also records if it has increased or decreased the speculation horizon in the previous execution. After that, it updates the speculation horizon as follows: (*i*) if the throughput has increased between the last and the current execution, it repeats the same action it took previously; for example, if it increased the speculation horizon, it will increase it again; (*ii*) otherwise, if the throughput has decreased, it takes the opposite action; for example, if it has previously increased the speculation horizon, it will now decrease it. Finally, because of small variations in the throughput (in two time intervals it is unlikely that the exact same number of transactions are committed), once it reaches the upper or lower limit for speculation, it will still oscillate and continuously increase and decrease the speculation horizon. As a consequence, it will quickly discover when the amount of parallelism varies.

4.2.4 Notifications

Low-level occurrences in a transaction can be monitored through transaction hooks. Transaction hooks give the application programmer the opportunity to get notified about relevant status changes in transactions. The main motivations for having such constructs are to enable external accesses from within a transaction and for collecting statistics. An external access consists of an access to any resource that is stateful and that is not instrumented by the STM. For example, if a file is created by code inside a transaction, it is also desired that this file is deleted if the transaction aborts. Another example is the use of special approaches for concurrent data structures (such as transaction boosting [HK08]). These data structures are not instrumented by the STM, but instead, they handle concurrency themselves. In this section, only the transaction hooks are detailed, examples of uses for building special data structures are detailed in Section 4.4.

The hooks are implemented through callbacks. The framework provides the following hooks:

being executed.

- `on-abort`: This is called when the transaction aborts. The transaction will be reexecuted at the earliest after returning from this call.
- `on-commit`: This is called after the final validation during commit, but before the new versions of modified memory positions are installed and the next-to-commit timestamp is advanced. At this stage all the consistency checks were been performed and only an explicit request (e.g., by having code that calls `abort()`) can cause an abort.
- `on-post-commit`: This is called after the transaction is committed and the timestamp is advanced.

It is important to note that the execution of the on-commit hook is totally ordered with respect to other transactions and no other transaction can commit while an on-commit hook is being executed. If total order is not necessary, the on-post-commit hook should be used as it does not block other transactions from committing. However, for the same reason, there is no guarantee that the on-post-commit callback for one transaction will be executed before the one for a later transaction.

4.3 Applications

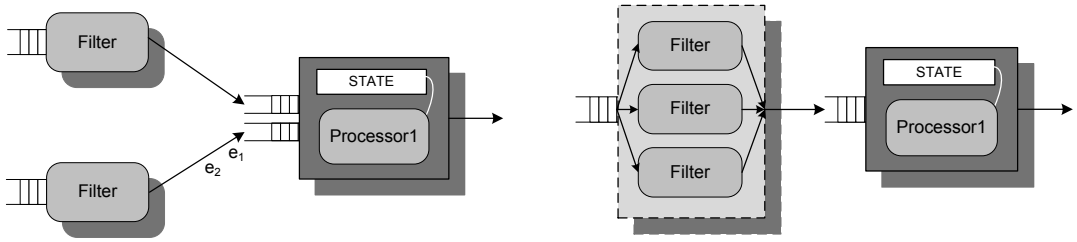
In this section we detail the use of the features previously discussed to provide out-of-order processing and optimistic parallelization.

4.3.1 Out-of-order processing

The general problem can be formulated as follows. Recall our running example: one scenario where out-of-order processing is often important is in the interaction between the `Filter` operators and the `Processor1` operator. As highlighted in Figure 4.1(a), under normal conditions (i.e., the system is not overloaded), it can be the case that, during some interval, events from only one channel are received. However, if in-order processing has to be guaranteed, either because the application semantics requires it or because the operator should behave in a deterministic way, the `Processor1` operator cannot process the events received until it learns whether the upper `Filter` will emit an event with a lower timestamp (which needs to be processed first). For the case of two sources, events from one channel node will wait, on average, half of the mean period of the events from the other channel (i.e., on the worst case, they wait a whole period, and on the best case, not at all).

Nevertheless, note that even for an operator with a single input channel, order may still be a problem. Consider, for example, that an event format needs to be converted (e.g., from XML to simple key-value pairs). This operation is stateless and reasonably costly, but can be simply parallelized by replicating the stateless operators as in Figure 4.1(b). However, different converter replicas will take different times to process events and the stream can become unordered.

Now consider the more detailed view of operator `Processor1` in Figure 4.2. The operator is single-threaded and stateful and its core is the `process()` function. This function takes events from the input queues and puts resulting events in the output queue. In the figure, the operator's



(a) (b)

Figure 4.1: In-order processing challenges: (a) multiple input streams and (b) out-of-order streams.

state is composed of 6 entities (e.g., a vector with 6 positions). Besides that, the `process()` function is currently processing the event with timestamp 11. Also shown in the figure is the wait queue for transactions that are still not able to commit, and the `NEXT` counter, which keeps track of the next-to-commit transaction. Note that although event 9 was already committed, because the next timestamp from the upper channel is not known, the next-to-commit transaction cannot be updated.

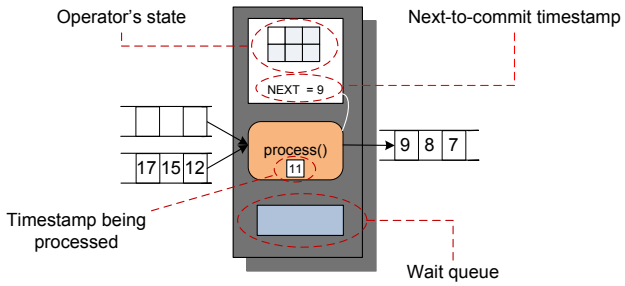


Figure 4.2: Structure of an operator with out-of-order speculation.

The main idea of the proposed approach is to process events speculatively out-of-order and keep the transaction open until the correct order is established. As soon as the order is established, the speculations can be validated and committed or reprocessed. An example is shown in Figure 4.3.

The example in Figure 4.3 runs as follows: (*a*) because the upper input queue is empty, after committing (the transaction processing) event 9 the operator does not know which event is the next to be committed and the *NEXT* field is not updated; (*b*) the operator decides to process event 11 speculatively; during this processing, one position of the operator's state is accessed; (*c*) when the processing of event 11 is finished, the associated transaction is put on the wait queue and the operator starts processing event 13 speculatively; right after that, event 12 arrives; (*d*) with the new event from the upper channel (12), the *NEXT* field can be updated and event 11 can be committed; (*e*) event 12 can be processed and committed, and event 15 is processed speculatively;

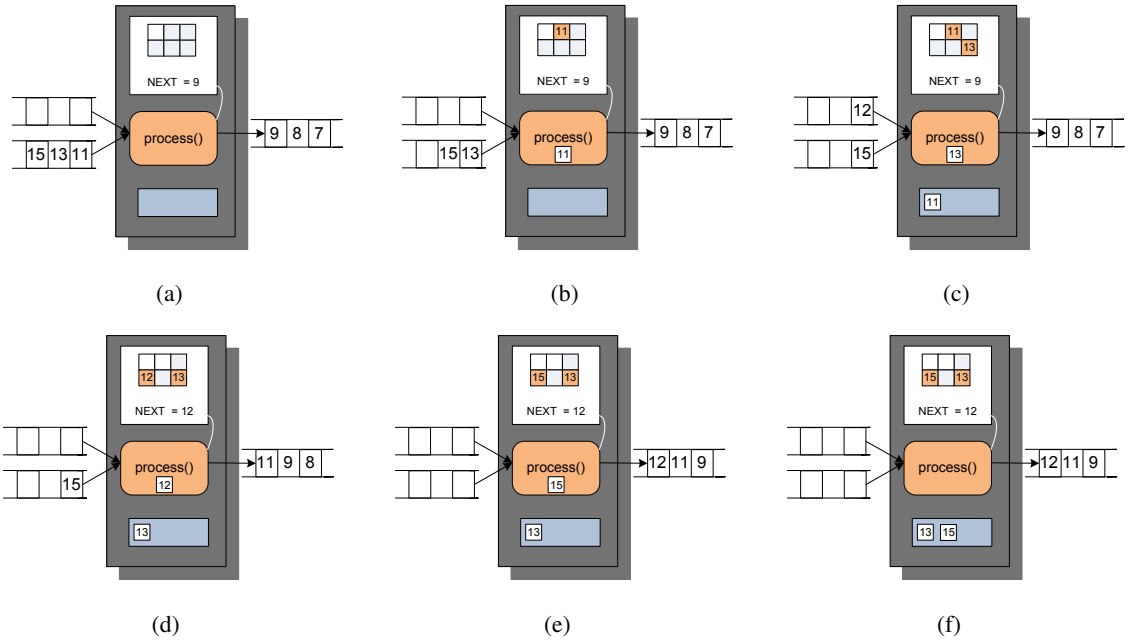


Figure 4.3: Example of speculative out-of-order execution.

finally, in (f), all queues are empty and the operator waits for information that allows it to commit pending transactions.

Note that, in the example above, because events 11 and 13 were already processed when event 12 arrived, waiting time was overlapped with useful computation. Note also that event 15 accessed the same part of the state as event 12. Therefore, as discussed in Table 4.1, depending on the access types (read or write), if event 12 had arrived after event 15 was processed, the operator could had to abort and reprocess event 15.

In addition, timestamps may have gaps. For instance, there was no event 10. As long as we can compute the next-to-commit timestamp and the timestamps are unique, the proposed approach will work. In this example, we assume each channel is ordered. Between two channels, the next timestamp can be computed by picking the lowest timestamp in the two queues. Moreover, to make timestamps unique, we can apply a bitwise shift on the original timestamp and insert the number of the channel (or the id of the source) in the lowest bits.

Finally, as can be inferred from the example, the success rate of out-of-order execution will depend on the available parallelism in the workload-operator combination. Later, in Section 4.5, we evaluate how effective speculation is for common operators.

4.3.2 Optimistic parallelization

The problem of optimistic parallelization is very similar to the out-of-order processing problem: given idle processors, is it possible to process events other than the next-to-commit event while still preserving sequential semantics? In practice, the real benefits of the speculative parallelization are that: (i) the operator code does not need to be parallelized by hand, which is a potentially long and

error-prone procedure; (*ii*) the main difficulty of concurrent programming is the nondeterminism (reasoning about correctness includes data races, deadlocks, memory models, etc.). Enforcing deterministic semantics simplifies composing, porting, debugging, and testing of parallel software [GN08, ABD+09, JAAS09]. Therefore, by having sequential semantics and information about the hot spots in the code, it is much easier to reason about how to extend sequential algorithms to perform better under speculative parallelization. Extracting the hot spots in the code with the STM requires simply enabling it to log which memory accesses caused the most aborts.

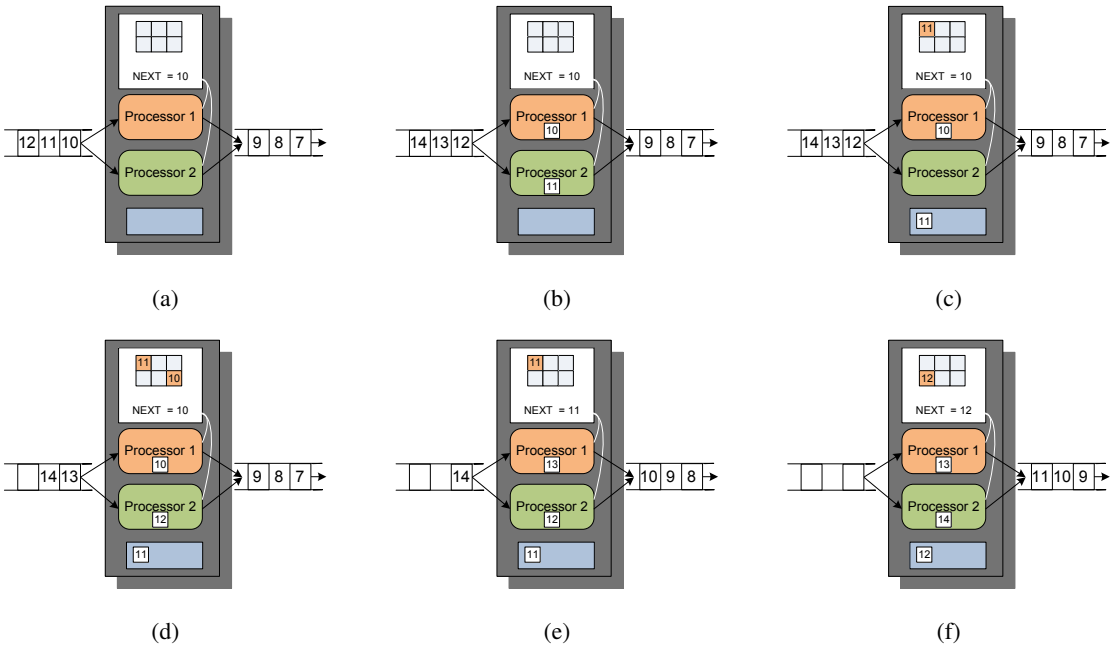


Figure 4.4: Optimistic parallelization example.

Similarly to the example in the previous section, the example in Figure 4.4 illustrates a concurrent execution in a speculative operator. It runs as follows: (*a*) two processors are available for processing events for the stateful operator; because the stream is ordered, the next-to-commit timestamp is known (alternatively, the events could have logical timestamps, which carry this information implicitly); (*b*) each processor takes an event from the input queue; (*c*) the processor with event 11 makes progress first and finishes processing; because timestamp 11 is not the next-to-commit, the associated transaction is put in the wait queue until the $NEXT$ counter reaches its timestamp; (*d*) the processor with event 10 makes progress and modifies a position in memory different than the one used by event 11; (*e*) the processing of event 10 is committed, the modification is incorporated into the state, and the result is outputted; (*f*) now the transaction for event 11 is the next-to-commit and its modification to the state is incorporated and the result outputted.

4.4 Extensions

One problem with STM is that some data structures are not speculation friendly. In such cases, transactions are (sometimes) unnecessarily aborted and, as a consequence, parallelism is not explored. Therefore, considering this issue is important to achieve good parallelization levels. In this section we consider two simple, but common, examples representing two very distinct cases. In the first case, which considers a *set* data structure, a transaction aborts when it reads a memory position (in fact, a pointer) that another transaction modified, even if this modification is not relevant. In the second case, which considers a single (shared) counter, concurrent transactions are aborted because there is indeed a dependency. We show how our framework can be used to provide better exploitation of parallelism in the presence of these two cases.

4.4.1 Avoiding unnecessary aborts

Consider a set data structure that is implemented with a sorted linked list. The interface of this set is shown in Listing 4.1. To understand under which circumstances an unnecessary abort occurs, consider two concurrent transactions tx_1 and tx_2 that add elements to the set. If the transactions add distinct elements, the operations should not interfere. Nevertheless, a word-based STM looks for reads and writes to memory positions. In addition, when adding a element, a transaction reads elements from the list until finding the correct position for the new element. One of the two elements will be inserted in an earlier position of the list and a pointer of an existing element will be modified during this insertion. However, the modified pointer was read by the other transaction, which inserted its element in a later position. Therefore, the two transactions do not successfully execute in parallel.

```
1  // Create a new set
2  set_t set_create();

4  // Insert the element in the set, duplicates are not allowed
5  // (returns true if element is added and false otherwise)
6  boolean set_add(set_t list, element_t element);

8  // Remove the element from the set
9  // (returns true if element was removed and false if it was not present)
10 boolean set_remove(set_t set);

12 // Query if an element is present on the set
13 boolean set_contains(set_t set, element_t element);
```

Listing 4.1: Interface of a simple set.

One simple and effective solution is to have the event processing framework provide a set data structure that is internally implemented with a hash table, which is a much more parallelizable data structure as it does not suffer from the aforementioned problem. Nevertheless, there are cases (specially for complex data structures) where the user has an efficient version of the data structure (e.g., lock-free, wait-free). In this case, we would want to enable reuse of the existing code.

An approach for transforming concurrent data structures in transactional ones is transactional boosting [HK08]. *Transactional boosting* considers that the original code of the data structure will not be instrumented by the STM, instead, wrappers will be added to the interface functions to provide the transactional features. Boosting is based on the idea that some of the operations on the data structure at hand are commutative. In addition, it assumes that interface operations have an inverse. On the one hand, knowing the commutativity relations is needed in order to determine which operations can be executed in parallel. On the other hand, knowing the inverse operations is needed for the case that the transaction aborts and the operation on the data structure have to be undone.

For a simple set, the inverses and the commutativity relations are defined in Table 4.2. The table on the left defines, for instance, that the inverse of an `add(x)` operation that returns *true* is the `remove(x)` operation. Further, no inverse (i.e., no undo action) is needed if it returns *false* (because no element was actually inserted). As another example, the `contains()` method requires no inverse because it is a read-only operation. In addition, the table on the right side defines which operations are commutative with each other.

Operation	*Inverse*
insert(x)/false	no-op()
insert(x)/true	remove(x)
remove(x)/false	no-op()
remove(x)/true	insert(x)
contains(x)/-	no-op(x)

Commutativity		
Op. 1	*Op. 2*	*Condition*
insert(x)/-	insert(y)/-	$x \neq y$
remove(x)/-	remove(y)/-	$x \neq y$
insert(x)/-	remove(y)/-	$x \neq y$
insert(x)/-	contains(y)/-	$x \neq y$
insert(x)/false	contains(y)/-	(always)
remove(x)/-	contains(y)/-	$x \neq y$
remove(x)/false	contains(y)/-	(always)

Table 4.2: Set data-structure specification for generating a speculation-aware set with transactional boosting (from [HK08]).

Based on the tables above, wrappers can be either manually or automatically generated for the data-structure interface functions. The wrapper should ensure that two operations that do not commute will not be executed concurrently. In [HK08] this is done by acquiring an abstract lock that is released when a transaction commits or aborts. Having an array of locks and a hash function mapping operations to locks would suffice. For instance, insert operations would get a lock that depends on the hash of the object being inserted. Thus, two insert operations with the same object would always map to the same lock and be mutually exclusive. The wrapper also adds the inverse operation to the `on-abort` hook, so that the effects can be undone if the transaction aborts.

4.4.2 Making aborts unnecessary

A common piece of code in stream operators is the increment of a single shared counter. For instance, the counter could be incremented for every event and used as a unique identifier. In other cases, it could be necessary for the operator to know how many events have already been

processed. Having such a counter is a problem for speculative parallelization. There will be a single point where all concurrent transactions conflict and only the highest priority one will be able to finish successfully. Thus, even for a very parallelizable type of operator, the achieved parallelism would be compromised.

To handle this problem, the counter operator (e.g., a *count*++ operation in C) could be replaced by a library call. For cases where simply a unique identifier is needed, the statement could be replaced by a atomic increment operation that is not instrumented by the STM. In this case, the counter would be incremented for each execution of the transaction, even the ones that end up in aborts. Nevertheless, because the counter is still strictly monotonic, the desired goal would still be achieved without compromising parallelization.

Similar to the case above is when an *approximate* counter is needed. For example, in the traditional reservoir approach for keeping non-biased samples of a stream (e.g., [Vit85]), the probability of a new event being added to the sample should be proportional to the number of events already seen. In this case, the above counter implementation would suffice. If a strictly monotonic counter value is not required, an even better approximation is to atomically increment the value during the execution and remember to decrement it if the transaction aborts.

Finally, it could still be needed to have a precise counter. One way to achieve this is to use an approach based on predictive log-synchronization (PLS [SS06]). PLS was originally proposed as a replacement for STM in some applications. Here, it can be used to improve STM efficiency in cases where one of many statements in an operator causes most of the conflicts. The original idea is that instead of modifying the shared object, threads would atomically log their requests. Then, they would use a base version of the object in conjunction with the log to derive how the object would look like if the threads really had executed the intended operation. The original assumption is that appending the request to the log is faster than executing the original operation.

In our case, appending the request to the log is not faster than incrementing a shared variable, but logging does not cause the complete transactions to abort and execute sequentially. Thus, the consequently available parallelism can overcome the added overhead. The idea works as follows. Instead of reading and updating the counter, when an operator thread wants to increment the counter it:

i) logs its request to increment the counter; the log entry also contains the timestamp of the transaction;

ii) the thread predicts the counter value by looking to the base value of the counter and simulating all the requests already in the log (it disregards log entries from transaction with higher timestamps);

iii) the predicted value is returned to be used by the transaction, but before that, the thread adds an action to the `on-commit` hook, so that the value is rechecked during commit; similarly, it adds an action to the `on-abort` hook to decrement the counter in case the transaction aborts;

iv) later, during the execution of the `on-commit` hook, which is serialized with respect to all other threads, it can compute the value it should have originally seen (i.e., all transaction

with lower timestamps will have already committed and will not add entries to the log anymore); if this check succeeds, it commits, otherwise, it aborts and reexecutes;

v) finally, periodically one of the threads locks both the counter and the log and apply the logged modifications, truncating the log.

In a summary, replacing the incrementing counter statement by a more parallel approach is feasible. In addition, if the user does not use one of the special calls, the transactifying compiler can automatically replace the original increment statement by the precise approach (PLS). Further, PLS can also be applied to other data structures in a similar way and although it has obvious overheads, it achieves very high performance in several cases (see [SS06] for details).

4.5 Evaluation

In this section we first evaluate the overheads introduced by the speculation infrastructure. Then, we show the cost of a misspeculation (caused, for example, by a conflict) and the real cost of the reexecution of an aborted task. After that, we show how optimistic parallelization, out-of-order processing, and the conflict predictors perform in simple micro benchmarks. We then finally evaluate speculation with two common ESP operators.

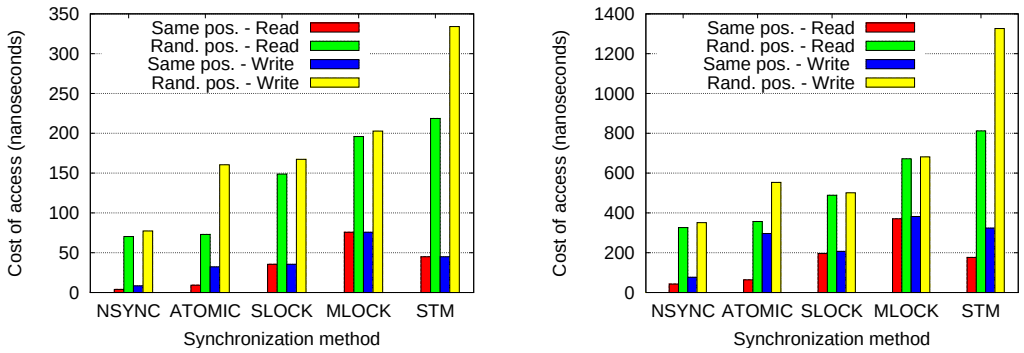
The experiments in this section were executed in two machines. The first is an Intel machine equipped two Xeon processors and 4 GBytes of RAM. In this machine, the total number of cores is 8 (2 chips with 4 hardware cores each) and they run at 2 GHz. Each 2 cores share a L2 cache of 6 MBytes (L1 is private to each core and has 32 KB for instruction and 32 KB for data). The second machine is a SUN T1000 machine equipped with a UltraSPARC T1 (Niagara) processor and 16 GBytes of RAM. This machine has 8 cores in a single chip running at 1 GHz and has four hardware threads per core. The four hardware threads in each core share the L1 cache (16 KB instruction; 8KB data) and the 8 cores share the L2 cache (3 MB). The Sun machine is an example of use of many simple and efficient processors, which is expected to be an increasingly more common approach. Both machines run Linux Ubuntu version 8.04.

4.5.1 Overhead of speculation

For the next experiments, we consider an operator with 512 Mbyte of state and we execute batches of 100 operations (unless otherwise noted). The duration of each batch is measured with the timestamp counter (TSC) in the Intel machine and the tick counter in the Sun machine. In both cases, the counters offer good precision for time measurement as they count the number of clock cycles and the machines have clocks in the order of gigahertz. We then consider the arithmetic mean of these measurements.

We start our evaluation with an experiment that illustrates the mean cost of a single memory access. The experiment is set up as follows. The operator reads or increments (i.e., a read plus write) several positions of this state. Two cases are considered. In the first case, the accesses are always directed to the same position. This increases the likelihood that the destination position is already cached. In the second case, the reads are directed to random positions, which significantly reduces cache hits. The experiment is done for both read and write accesses and considers several

synchronization methods for protecting the state: no synchronization at all (*NSYNC*), atomic operations (*ATOMIC*), mutex locks (*MLOCK*); spin locks (*SLOCK*); and the STM (*STM*). The results are depicted in Figure 4.5. In this experiment, only a single thread is running and, thus, there is no contention for the accesses.



(a) Intel Xeon (b) Sun Niagara

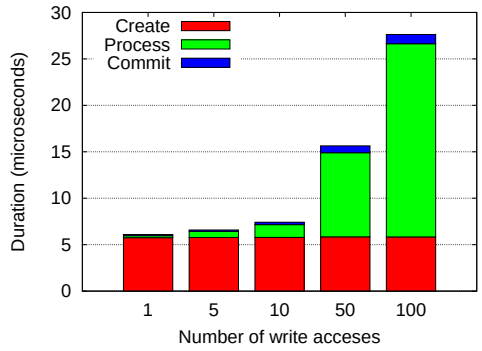
Figure 4.5: Mean cost of a memory access.

As expected, the results show a considerable overhead introduced when accessing memory positions through the STM. The overheads are specially high for random accesses. In this case, not only accessing the destination memory position is costly, but the entry in the STM's *lock array* monitoring this position is likely not in cache either.

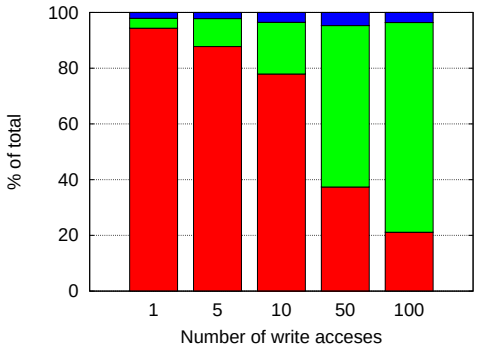
The previous experiment considers only the mean cost of a single memory access. It does not consider the costs of creating and committing the surrounding transactions. Although the time for a single memory access is the most important factor when considering expensive operators, for operators with few accesses the costs for setting up and committing the transaction may be dominant. In the following experiment, we decompose the duration of a transaction considering different numbers of accesses to the operator state. The results are shown in Figure 4.6. The left-hand side figures show the decomposition of the mean transaction duration in microseconds. On the right-hand side of the figure, the phases of the transaction are normalized to the total duration of the transaction. The high overhead for the creation is mainly due to several memory allocations (e.g., transaction descriptor, read and write sets) and the cost of periodic garbage collection.

In the next experiment, we contextualize the costs of memory accesses by evaluating their impact in processing tasks with different sizes. The size of a task is defined in terms of busy-waiting time. This busy-waiting represents a local computation that does processing and accesses local variables, but does not require shared memory accesses. The results when considering only a few write accesses to the state (5) are shown in Figure 4.7. The results for a greater number of accesses (100) are shown in Figure 4.8. The figures show the slowdown introduced by the synchronization methods in comparison to the non-synchronized option (i.e., the bare access, no lock, no atomic operation, no transaction). The accesses are writes and are directed to random positions in the operator's state.

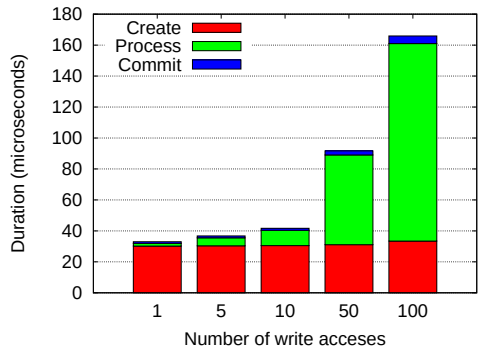
Although the impact of the other synchronization approaches is small, for very short transac-



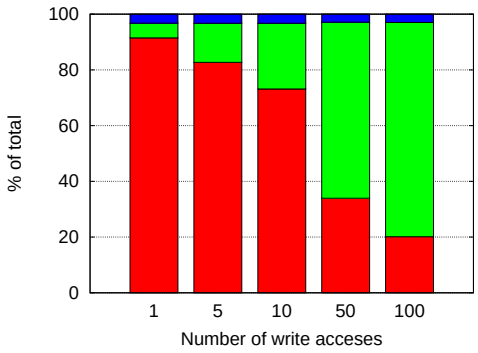
(a) Intel Xeon



(b) Intel Xeon

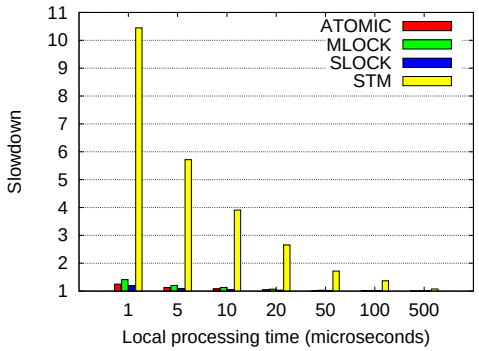


(c) Sun Niagara

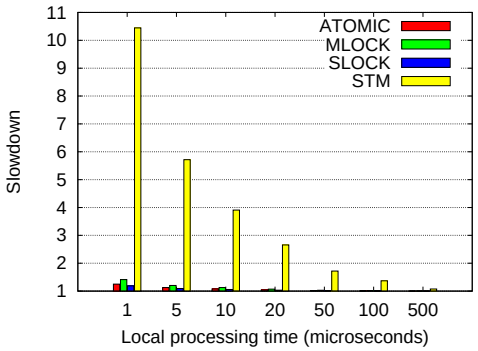


(d) Sun Niagara

Figure 4.6: Duration of major phases for transactions with different sizes.

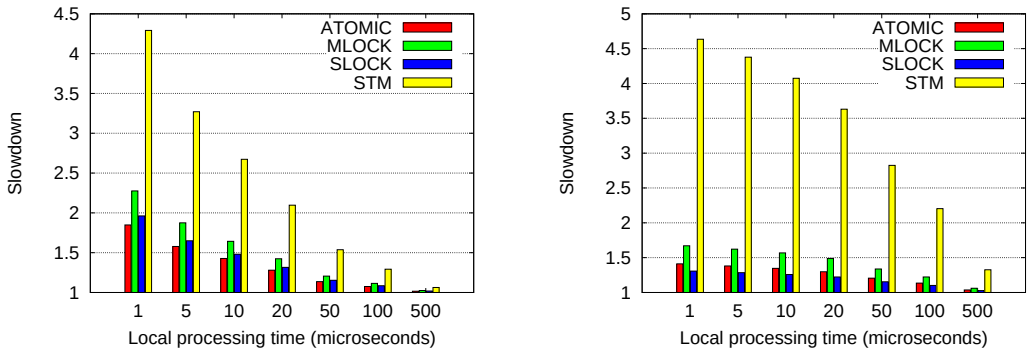


(a) Intel Xeon



(b) Sun Niagara

Figure 4.7: Impact of cost in the total processing time for different task sizes (i.e., processing times) and 5 shared memory accesses per task.



(a) Intel Xeon (b) Sun Niagara

Figure 4.8: Impact of cost in the total processing time for different task sizes (i.e., processing times) and 100 shared memory accesses per task.

tions the impact of the STM is very large. This is mostly due to the large overhead for creating transactions in our prototype (see also Figure 4.6). The impact of using transactions becomes smaller when there are enough local computations to be carried.

Finally, in the last experiment of the section, we show the effect of contention in the cost of memory accesses. In this experiment we consider two threads. The first thread either reads or writes to a set of memory positions. This thread then blocks and a second thread may modify the exact same set of positions. Finally, the first thread runs again and re-reads or re-writes the same set of addresses. Four interference patterns are considered in the Intel machine: (*i*) *No interference*, this is the only case in which the second thread does not write to the memory positions accessed by the first thread (this is the base reference case); (*ii*) *Shared L2 cache*, the two threads run in cores that share the L2 cache; (*iii*) *Same chip*, the two threads run in cores in the same chip (they do not share the L2 cache); (*iv*) *Different chip*, the two threads run in cores located in different chips. For the Sun machine (single chip), we consider three patterns: (*i*) *No interference*, as with the Intel machine; (*ii*) *Same core*, the two threads run in two hardware threads in the same core; (*iii*) *Same chip*, the two threads run in different cores (shared L2 cache). The costs of write accesses from the first thread are depicted on the left-hand side of Figures 4.9 and 4.10. The cost for reads is shown in the right-hand side of the same figures. The goal of the experiment is to evaluate the effect of contention on the metadata used for synchronization of a single memory access. The numbers do not consider the creation and commit of the surrounding transactions in the STM case. In some cases, the STM accesses are faster because they involve only atomic operations; no locks need to be acquired.

4.5.2 Cost of misspeculation

In the next step, we consider the cost of an undo operation and how this reflects on the speculation process. Computations are undone due to incorrect speculation and, thus, these computations need to be redone. The cost of an undo operation is illustrated in Figure 4.11. Because we consider a write-back STM approach for implementing our speculation, the undo operation simply consists

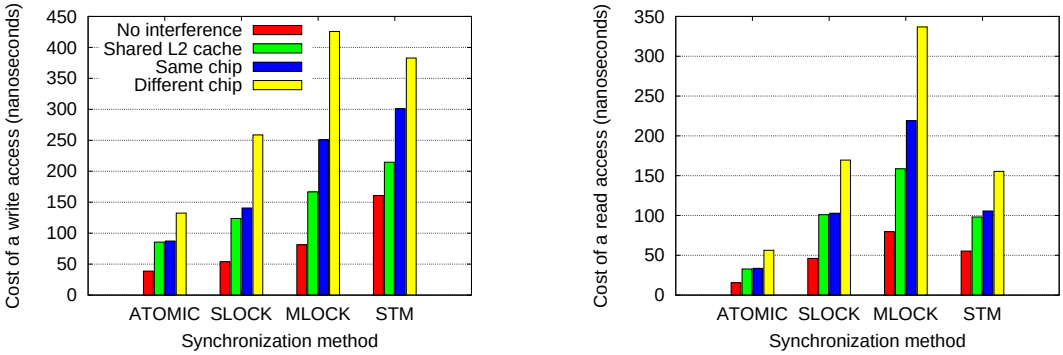


Figure 4.9: Impact of contention in the memory access time (Intel Xeon).

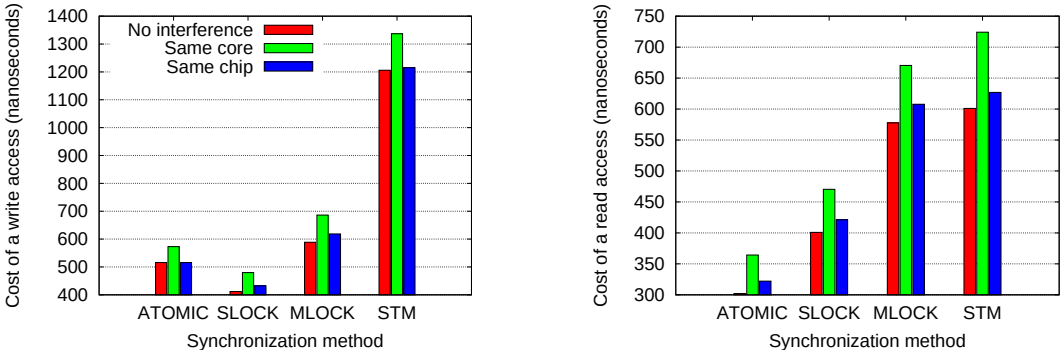
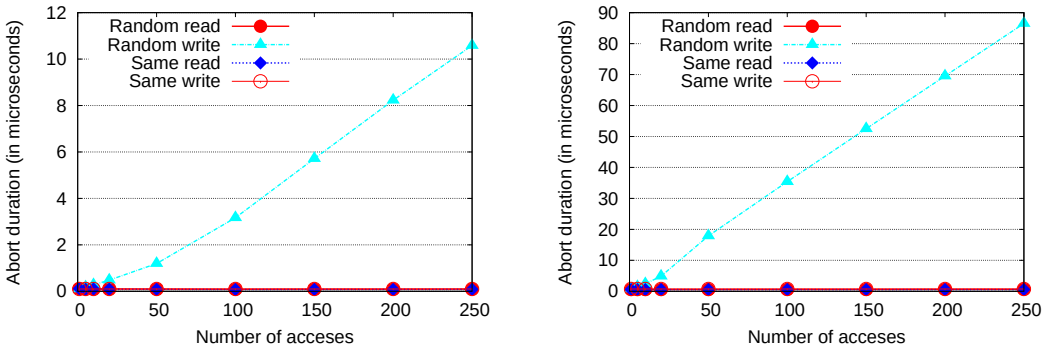


Figure 4.10: Impact of contention in the memory access time (Sun Niagara).

of aborting the current task. As shown in the figure, the undo cost depends on the number of distinct write accesses executed by the task. This is due to the atomic write operation executed on the global lock array. For the read cases no entry in the lock array needs to be released. For writes directed at the same positions only a single release must be done.



(a) Intel Xeon (b) Sun Niagara

Figure 4.11: Cost of an abort/undo operation.

Although a misspeculation requires a task to be completely reexecuted in order to consider updated values, a reexecution can also be faster than the first execution as some data will be in cache. The next experiment illustrates this effect. In this experiment, a task executes a number of memory reads directed to the state of the operator. Then, the processing is aborted due to one or more positions being modified by a concurrent transaction with higher priority. The processing overhead is then given by the total time of an execution followed by an abort and a reexecution, in comparison to the duration of the first execution. This can be expressed as:

$$T_{total} = T_{first} + T_{abort} + T_{reexecution} \tag{4.1}$$

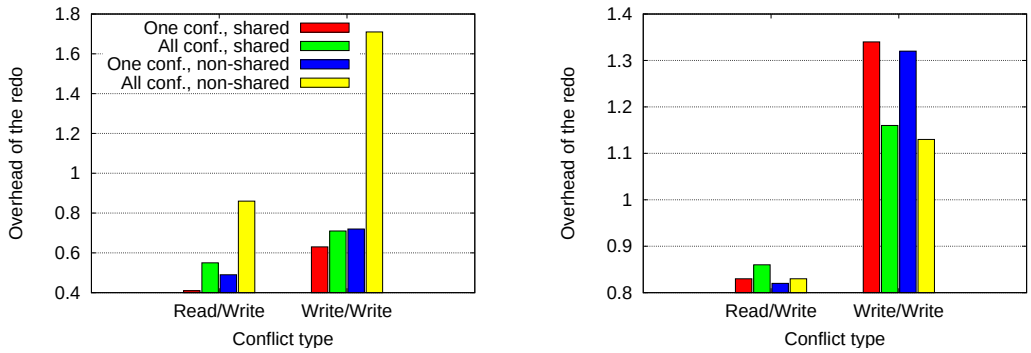
$$Overhead = (T_{total} - T_{first})/T_{first} \tag{4.2}$$

where T_{total} is the total duration time, T_{first} is the duration of the first execution, T_{abort} duration of an abort (undo), and $T_{reexecution}$ is the duration of the re-execution.

The results of the experiment are shown in Figure 4.12. The experiment also considers tasks with different amounts of positions that actually changed between the first and the second execution (either 1 or 100 positions conflict). The amount of resources wasted by a misspeculation depends on the number of positions that indeed changed, referred to as conflicts. In the Intel machine, for cases where just one position changed, misspeculation has only moderate additional costs (i.e., much less than the intuitive 100%). In this machine, we consider the two opposite cases where the two threads either share the L2 cache (i.e., they are in neighbor cores) or they are in different chips.

In the Sun machine (right hand side of Figure 4.12), we consider the cases where the threads either run in hardware threads sharing the same core or they are in different cores. In addition, differently from the Intel machine and from the read-write case, the proportional overhead for write-write interferences is higher when only one address is changed by the interfering thread.

This effect is caused by two factors. First, the Sun machine has very small L1 caches, so the reexecution of a transaction does not benefit from a previous execution. Second, in the case that all addresses of the current transaction were used by the other transaction, the locks were stolen and the current transaction does not need to release them anymore. By not having to release addresses during the rollback, transactions save time (e.g., costly compare-and-swap operations). Therefore, the difference between the all-conflict and the one-conflict executions is due to the cost of the rollback.



(a) Intel Xeon (b) Sun Niagara

Figure 4.12: Overhead of a rollback and a reexecution.

4.5.3 Out-of-order and parallel processing micro benchmarks

In this section we consider a micro benchmark that uses the simple operator depicted in Listing 4.2. This operator works as follows. For each input event the operator modify one entry of its local state. The entry to be modified depends on *value* field in the event, which is set to a random integer by the event generator. Then, if we want to simulate a longer task, we simply do some busy-waiting for the desired amount of time. Therefore, the two base parameters of this benchmark are: (*i*) the size of the state (the bigger the state is, the lower the chance that two speculations conflict); and, (*ii*) the amount of local work to be done (the longer the work, the lower will be the impact of the transaction creation and commit overheads). In the following experiments, we use this operator to evaluate the optimistic parallelization, out-of-order execution, and the optimism control.

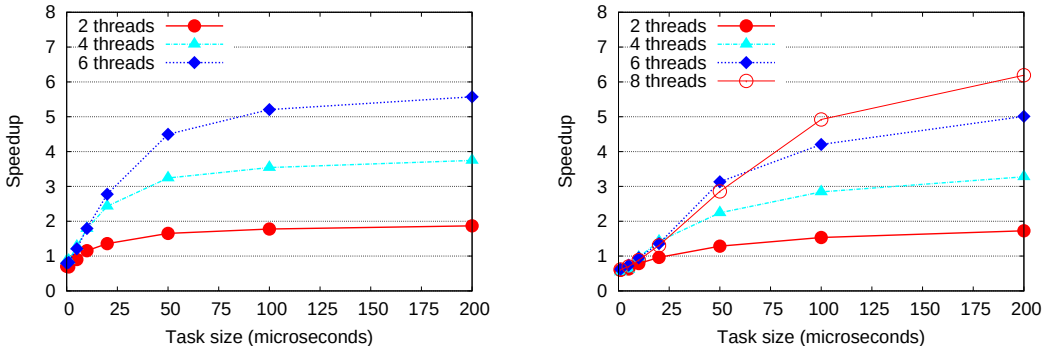
The first experiment is shown in Figures 4.13 and 4.14. The x-axis represents the varying simulated work. For the former figure, the state size is set to 100. For the latter, the state size is set to 10. As shown in the figures, the achieved parallelization depends on the amount of local processing done and on the probability of conflicts. We consider that the operator uses up to 6 threads in the Intel machine[2] and up to 8 threads in the Sun machine.

Next, the experiment for the out-of-order is set up as follows. The operator receives events from two sources and processes them according to their timestamps (as detailed in Section 4.3.1).

[2]If 8 threads were used, there would be competion between the event-processing threads, other threads from the framework, and system processes.

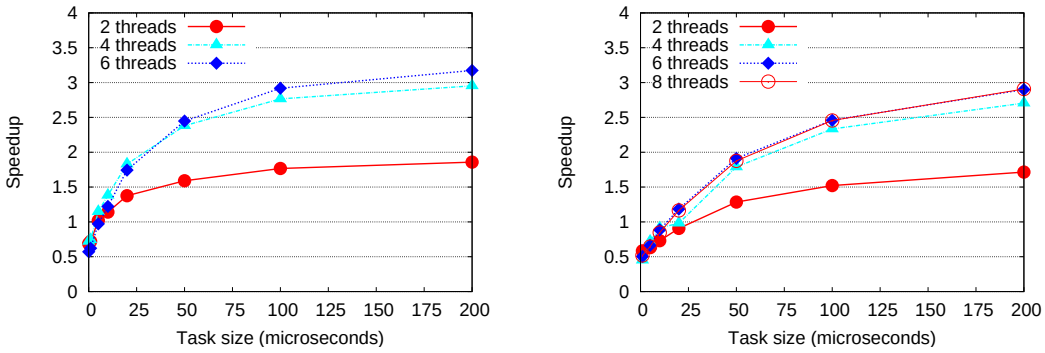
```
1  long *local_state; // Operator state

3  int operator_init() {
4     local_state = calloc(state_size, sizeof(long)); // Allocate state
5     if (local_state == NULL) return EPF_FAILURE;
6     return EPF_SUCCESS;
7  }

9  int operator_process(void *input) {
10    Event_t *in = input;
11    in->result = READ(local_state[in->value % state_size]); // Read an entry
12    INC(local_state[in->value % state_size]); // Modify an entry
13    if (simulate_work > 0) tsc_sleep(simulate_work);
14    return EPF_FORWARD;
15 }
```

Listing 4.2: Simple operator for micro benchmarks.



(a) Intel Xeon (b) Sun Niagara

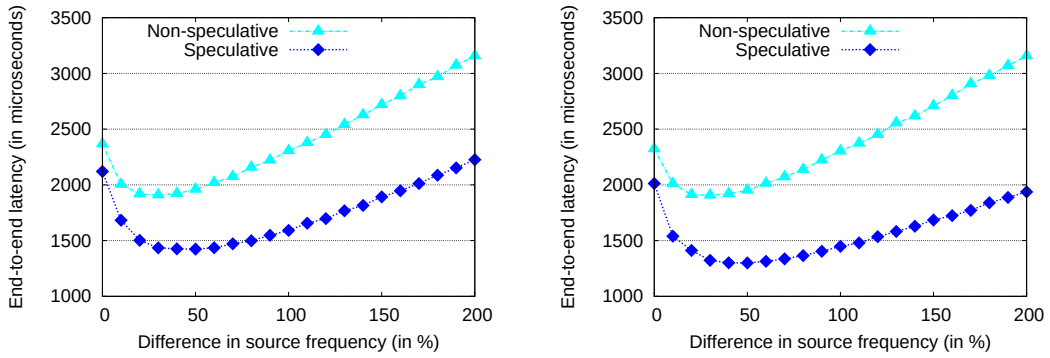
Figure 4.13: Parallelization micro benchmark: 1% of the state is updated.



(a) Intel Xeon (b) Sun Niagara

Figure 4.14: Parallelization micro benchmark: 10% of the state is updated.

Thus, the main parameter is the difference in the generation rate of the two sources: if one source emits far less events, the operator will have to block longer in order to ensure events are indeed being processed in the correct order. Nevertheless, because the suitability to out-of-order processing also depends on the available parallelism in the combination workload-operator, we again consider two different state sizes (10 and 100). We set one source to emit events in a fixed frequency (mean of 1000 events per second, with a random interval between events) and the other to emit events with decreasing frequencies (also with random intervals between events). The x-axis show the difference in the rates, for 0% both sources produce events in the same frequency. Similarly, for 200% the faster source produces three times more events. The amount of simulated work is set to slightly less than half of the period of the fastest source, so that the operators are able to keep up with the events stream when the sources are in the maximum frequency. The results are shown in Figure 4.15. In this experiment, we consider only a single thread in the operator and results are similar in both machines.



(a) 10% (b) 1%

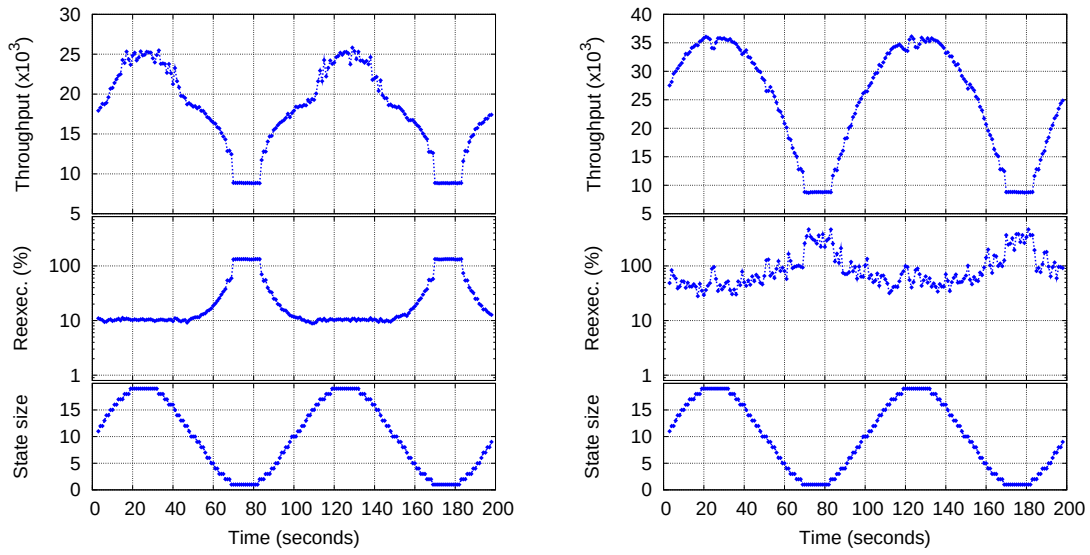
Figure 4.15: Out-of-order micro benchmark: 1% and 10% of the state is updated (Intel Xeon).

As seen in Figure 4.15, there is a peak in latency when sources produce events in the same rate. This is due to the fact that in this case the operator is busy almost 100% of the time. Thus, an arriving event is likely to not be processed immediately, even if the event has a suitable timestamp. For example, consider that the sources are almost perfectly synchronized. Now consider that the operator receives an event e_{t_0} (with timestamp t_0) from `Source1`. Then, soon after that it receives an event e_{t_1} from `Source2`. Because $e_{t_0} < e_{t_1}$, the operator is now able to process e_{t_0}. One period later, the operator receives e_{t_2} from `Source1`, which unblocks e_{t_1}, and shortly after that, e_{t_3} from `Source2`, which unblocks e_{t_2}. Nevertheless, the operator is still busy processing e_{t_1} and e_{t_2} suffers an unnecessary delay. When the operator is mostly busy, it is not possible to mask this latency (even with speculation as there is no time available for speculation). On the contrary, when the slower source produces far less events, an arriving event will encounter an idle operator and may be processed immediately.

In the last experiments of this section we evaluate the conflict predictors as an approach to make speculation more efficient. In this experiment we try many possible state sizes by making the state size vary with a sinusoid. The goal is to show how the system reacts to a workload in which the available parallelism varies. Nevertheless, even when the available parallelism of a

workload-operator combination does not vary, we still need to estimate a good speculation level for a new operator. The additional experiment parameters are the following: events have logical timestamps (i.e., fixed distance between the timestamps); a single source produces all events; the operator is allowed to use 6 threads; and the simulated work is 100 μs.

We try two different conflict predictors: the abort-driven and the throughput-driven predictors. In Figure 4.16, we plot the state size, the percentage of reexecuted transactions, and the total throughput of the operator. As mentioned above, the state size varies with a sinusoid (with a period of 100 seconds) and the larger the value, the more parallelism is available. In the sinusoid maximum, the state has 20 positions and, in the minimum, only 1 (i.e., no space for speculation). As expected, when there is no parallelism left there are many aborts, even for the abort-driven predictor. This happens because it repeatedly tries to find speculation opportunities. Similarly, the throughput predictor is able to achieve higher throughput as it does not care for wasted speculations as long as they do not reduce throughput.

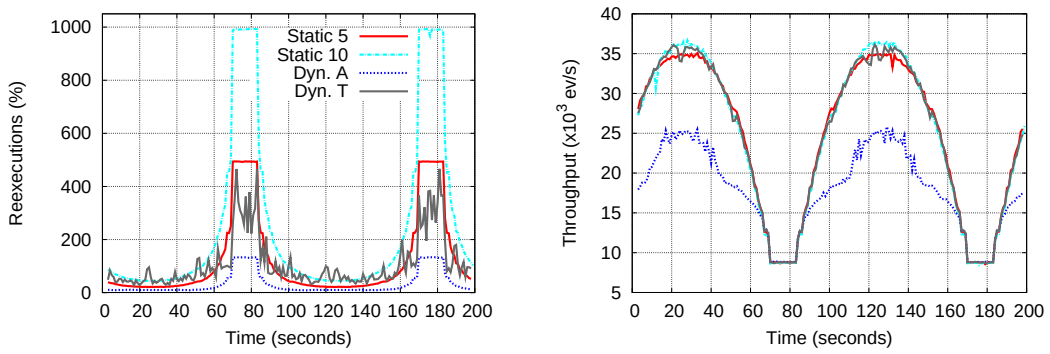


(a) Abort-driven control

(b) Throughput-driven control

Figure 4.16: Micro benchmark for the abort-driven and throughput-driven predictors (Intel Xeon).

Then, in Figure 4.17 we compare the two conflict predictors, which use a dynamic speculation horizon, with statically-set speculation horizons. The static predictors are referred to as *Static 5* and *Static 10* for horizons of 5 and 10, respectively. The dynamic predictors are referred to as *Dyn. A* for the abort-driven and *Dyn. T* for the throughput-driven. We use the same (sinusoidal) set up as in the previous experiment. On the one hand, the abort-driven predictor has the lowest number of reexecution, as expected. On the other hand, the throughput-driven predictor has throughput very close to the static predictors, but with lower conflict rates, confirming its usefulness.



(a) Re-executions (b) Throughput

Figure 4.17: Comparison between dynamic and static speculation horizons (Intel Xeon).

4.5.4 Behavior with example operators

In this section, we evaluate the speculation support with two example operators. The first is the top-k operator discussed in Chapter 2.1. The second is a *nearest neighbors* operator.

The nearest neighbors problem [Ary95] is a fundamental problem in computational geometry and is very commonly applied in classification and pattern recognition. Given a set of points S and a query point p, the goal is to find a point in S that is the neareast neighbor to p. In practice, instead of the single nearest neighbor, a few points are retrieved in order to handle noise (e.g., p's classification will be done based on the average of the n nearest neighbors). Solving the nearest neighbor problem in a space with few dimensions (e.g., in a bidimensional or a tridimensional space) is a simple problem (in contrast to much larger dimension numbers, where the distribution of the distance between two points tend to be more homogeneous [Ary95]). The trivial algorithm would be to put all points in a list and for each input event, compute the distance to all other points and select the n closest points. This is very innefficient, however. An improved version is the bucketing algorithm [Wel71]. In this case, the space is divided in equally-sized buckets and the points stored in these buckets. When a new point is added, it is added to a single bucket. Then, to compute the nearest neighbors of a new point p, we can compute the distance of p to other points in the same bucket and after that look into neighbor buckets.

In our experiments, we configure our top-k to return the 5 most frequent occurrences of a certain integer attribute in the events. This attribute could represent, for example, a hash of a more complex object. For the nearest neighbors, the events contain a bidimensional coordinate and we want to query for the 3 nearest neighbors. In addition, we divided our space in a 100×100 grid and keep up to 10 points in each bucket. Limiting the number of points kept enables us to keep the space and computational costs bounded and uniform among buckets. With these configurations, we obtained two reasonably different operator profiles. The top-k has a short to medium computation and some, but not much available parallelism. The nearest neighbors is a longer, very parallelizable (because of the grid) computation.

Shown in Table 4.3 are the processing costs of the operators with different configurations. The original cost in a nonspeculative, sequential execution is listed in the lines marked as *Seq*.

For speculative executions, we consider 2, 6, and 8 threads (2 and 6 in the Intel and 2 and 8 in the Sun) and also list the percentage of aborts. The overhead caused by the speculation support can be seen by comparing the computational costs of the multithreaded executions and the sequential ones. Node, however, that speculative executions may put events aside when these are processed, but cannot be committed yet. The processing time in the table considers the total time between the dequeueing of the event until its commit. Therefore, putting events aside increase the measured values.

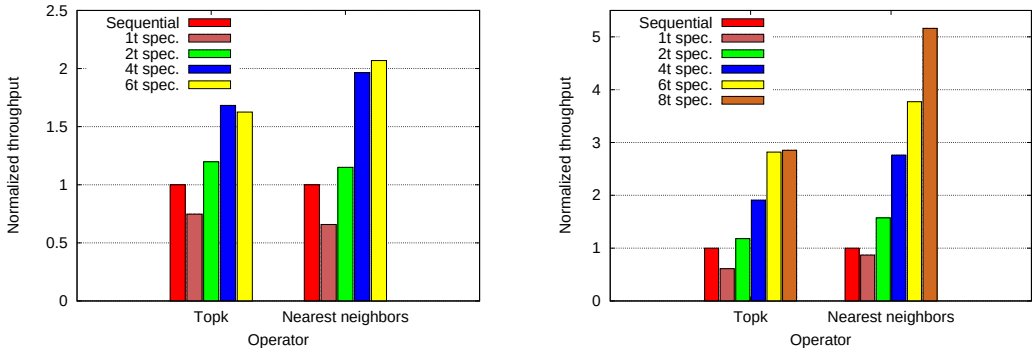
Operation	*Machine*	*# threads*	*Proc. time (μs)*	*Conflict rate*
Top-k	Intel	Seq.	12.42	-
		2	25.69	1.25%
		6	144.81	16.46%
	Sun	Seq.	77.20	-
		2	120.99	1.30%
		8	328.57	14.76%
Nearest Neighbors	Intel	Seq	26.68	-
		2	49.72	0.06%
		6	159.27	0.64%
	Sun	Seq.	732.54	-
		2	902.35	0.05%
		8	1155.89	0.36%

Table 4.3: Computational costs of the operators (multithreaded executions use speculation).

In the next experiment we measure the throughput gained by using speculation for optimistic parallelization. In Figure 4.18, we plot the speedup of the parallel version in relation to the sequential (nonspeculative) version. Note that if only one thread is used the overhead of the speculation reduces the total throughput. Nevertheless, already with two threads we surpass the initial deficit and achieve some speedup for both operators.

Next, we compare the speedup of the speculation with the one achieved with fine-grained locking (with *pthread mutexes*). In the locking version of the top-k operator, there is a lock for each counter in the count sketch synopsis and a lock for the 5-position array that keeps the top-5 elements. For the nearest neighbors, there is a lock for each bucket. The results are shown in Figure 4.19. As shown in the figure, for the top-k, there is a clear advantage for the lock-based version. Nevertheless, our approach still have the advantages that it both does not require any concurrent programming knowledge and still is deterministic. The quick drop in performance with 6 threads in the lock-based version also shows that selecting the correct amount of threads for the fine-grained locking may not be straightforward.

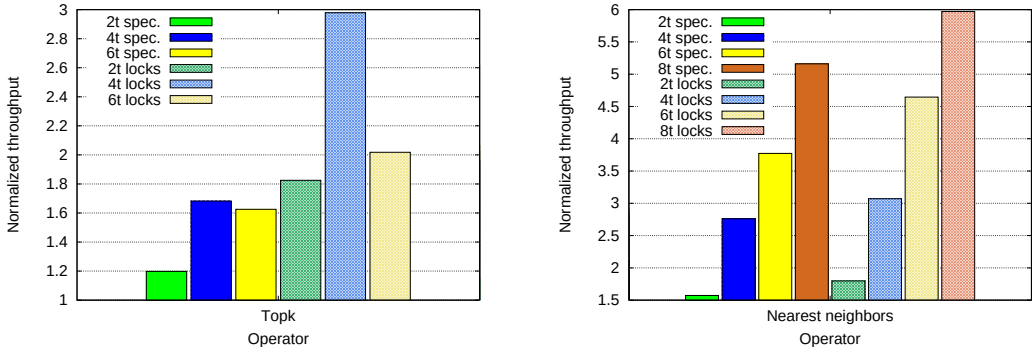
On the right-hand side of Figure 4.19 we have the comparison between the performance of the lock-based and speculation-based versions for the nearest neighbor operator. The computation of the nearest neighbor operators is both long enough to mask the creation and commit overheads and has a low conflict rate, which allows more transactions to successfully run in parallel. As a



(a) Intel Xeon (b) Sun Niagara

Figure 4.18: Optimistic parallelization speedups for commons stream mining operators (base: sequential nonspeculative execution).

result, both approaches achieve a similar gain.

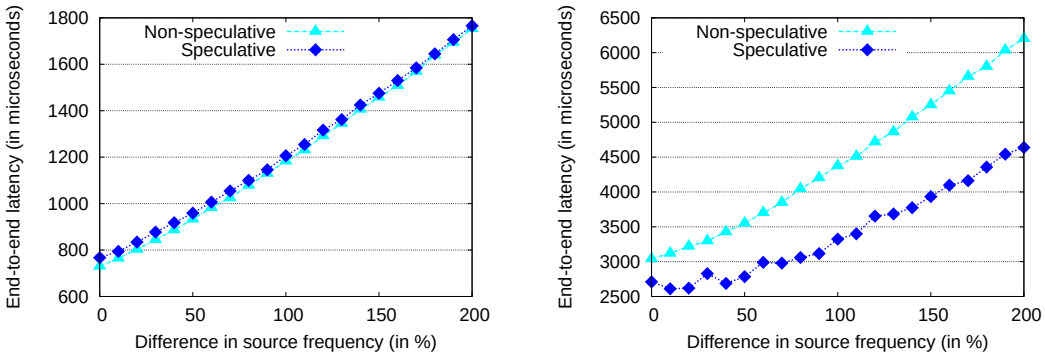


(a) Top-k (Intel Xeon) (b) Nearest Neighbors (Sun Niagara)

Figure 4.19: Comparison between speedups from optimistic parallelization and fine-grained locking for two common stream mining operators (base: sequential nonspeculative execution).

Next, we evaluate the out-of-order processing. On the left-hand side of Figure 4.20, we show a single-threaded execution of the top-k operator. In this case, the speculation overhead in comparison to the computation cost of the operator masks the gain of the early processing. Because we do not forward speculative results, the speculative version can process event in advance, but has to wait to be able to commit the computations. Nevertheless, on the right-hand side of Figure 4.20, with the nearest neighbor operator in the Sun machine, the overhead is proportionally lower and we see results similar to the micro benchmarks in the previous section.

Finally, in our last experiment, we compare the performance of our conflict predictors with an statically set speculation horizon. Results are shown in Table 4.4. The results show that having a dynamic conflict predictor helps to stay close to optimal points, but it is not a deciding factor.



(a) Top-k (Intel Xeon)

(b) Nearest Neighbors (Sun Niagara)

Figure 4.20: Latency of a system with two sources and varying event generations rates.

Operation	*Metric (mean)*	*Dynamic (A)*	*Dynamic (T)*	*Static (5)*	*Static (10)*
Top-k	Aborts	2.23%	9.05%	7.08%	16.37%
	Throughput (ev/s)	61379	75640	78300	73439
Nearest Neighbors	Aborts	1.58%	0.92%	0.30%	0.67%
	Throughput (ev/s)	58237	57148	57141	55975

Table 4.4: Throughput and abort rate for the stream mining operators with different conflict predictors (Intel).

4.6 Summary

In this chapter we discussed the usage of speculation inside an event stream processing operator. In order to make an operator speculative, we surround it with a modified software transactional memory and instrument its operations directed at the state. In comparison to conventional STMs, ours needs several added features, being ordered commits, transaction pausing, and optimism control, the most important. Our goal was to provide an easy way to parallelize operators (increasing their throughput) and to enable them to preprocess events out of the normal timestamp order (reducing their latency).

Optimistic parallelization provides an easy way to improve parallelism in a stateful operator. Producing parallel code is known to be a difficult problem and it is easy for the programmer to make wrong assumptions about synchronization safety [GN08, ABD+09, JAAS09]. Optimistic parallelization is safe for the beginner programmer: it preserves sequential semantics and is independent of the number of cores. For expert programmers, it allows them to quickly have a parallel version and then, if necessary, increase parallelism by, for example, replacing data structures with more speculation-friendly implementations. The determinism enabled by the sequential semantics also helps: according to a survey among expert developers inside Microsoft [GN08], 75% of the interviewed consider concurrency bugs hard or very hard to reproduce.

In addition, as shown by Porter et al. [PHW07], even code that is highly optimized for fine-grained locking, like recent Linux kernels, still show opportunities for optimistic concurrency.

Our optimistic parallelization also supports incremental refinement of implementations. The STM can be used in a profile mode, in which it outputs where most of the conflicts were caused. Then, a user could detect whether there are structures that are unfriendly to speculation and decide about the next improvement. As a simple example, consider a counter that is incremented for every input event. This counter will cause all optimistically processed events to abort as they all contend for the counter. Nevertheless, as shown in Section 4.4, this problem can be mitigated by using approaches such as predictive log-synchronization [SS06]. Further, the counter may not be essential to the computation. Then, by receiving the feedback from the STM stating that the counter causes too much aborts, the programmer may decide to simply remove it.

Regarding our evaluation, we have seen that although individual memory accesses through the STM are very expensive, the impact of these costs is amortized if the transactions include considerable processing that accesses only local-scope variables. We have also shown that this happens for two common ESP operators (top-k and nearest neighbors). Then, by having parallel executions of the operator we can mitigate the overhead and still achieve parallelism. Moreover, when considering the amount of computational resources used versus the achieved speedup, a speedup linear in the number of cores is not the only metric. For example, one approach is to consider the speedup, the relative gain in throughput, versus the *costup*, the relative increment in hardware cost (i.e., the cost of adding the extra cores) [WH95][3]: whenever the ratio between speedup and costup is greater than one, parallelization is a meaningful and cost-effective approach. In our case, in the Intel Xeon machine and using 4 cores, we achieved a mean speedup of 1.67 for the top-k and 1.96 for the nearest neighbors. For the Sun Ultrasparc T1 we achieved speedups of 1.91 and 2.76 for the top-k and nearest neighbors, respectively. In all cases, the extra hardware costs when moving from a single core to a 4-core processor are clearly lower than the speedups achieved. Nevertheless, we still have an advantage not considered in [WH95]: we require no additional development cost.

Finally, we have seen that local speculation (specially the out-of-order processing) may not be advantageous in computationally cheap operators because the commit is still delayed until the speculation conditions are learned (e.g., ordering). Then, if the commit and the original (nonspeculative) computations have similar costs, there will be no profit even with low conflict rates. In the next chapter, we show that in distributed scenarios, speculation is interesting even for computationally cheap operators. We will also detail why the lack of determinism in lock-based parallelization is a serious problem for fault-tolerant distributed ESP systems.

[3]The costup metric was initially proposed to consider the cost of multiprocessors from the perspective of the memory costs. Nevertheless, the idea of evaluating the added benefit of a system in relation to the added cost is clearly widely applicable.

Chapter 5

Distributed speculation: low-latency fault-tolerance

In this chapter, we investigate the use of distributed speculation to build distributed fault-tolerant event stream processing (ESP) systems. The goal is to minimize the performance impact of common fault-tolerance techniques. We present speculative approaches for techniques based both on checkpoints and on redundant computations. In contrast with the previous chapter, where gain was a function on parallelism available in the workload, available parallelism plays a smaller role here. In this chapter, gain is achieved when failures are rare.

5.1 Overview

ESP systems have attracted significant attention in recent years and have being used in a growing number of applications. Besides throughput and processing latency, availability becomes a major requirement as the use of ESP becomes widespread. An ESP system typically runs on a network of computers. According to studies on the availability and failure rates in large computer clusters, one can expect that several computers crash every day. Nevertheless, only recently have researchers started to design and implement fault-tolerant distributed ESP systems [SHB04, HBR+05, HCZ08, BBMS08].

High availability is obtained through fault tolerance. Fault tolerance is a non-trivial requirement, especially for systems that require precise recovery. With *precise recovery* the results in case of failure should not be semantically different from the results that would be obtained in the failure-free case. Adding precise recovery to an ESP system can add considerable latency costs for the failure-free runs. This is a consequence of requiring that any nondeterministic information used by an operator be saved to enable later recovery.

In this chapter, the optimistic assumption is that failures are rare. Speculation can then help to minimize the latency costs of fault tolerance in ESP systems. Assuming the crash-failure model, in order to enable precise-recovery nondeterministic decisions that influence a computation (e.g., arrival order of events, scheduling decisions) must be secured on disk (as in rollback recovery) or by informing them to a remote node that replicates the computations (as in active replication).

In both cases, the operator needs to wait for an acknowledgement that the information is stable before outputting any result events. Not waiting for the acknowledgement may lead to output events that depend on a state that cannot be reconstructed during recovery. If downstream nodes consume these outputs, the system may end up in a inconsistent state after a recovery. Our speculation mechanism helps by optimistically assuming no failures occur and, thus, not waiting for the acknowledgments. Nevertheless, the speculation mechanism still enables operators to be efficiently rolled back when this assumption of no failures is wrong.

The optimistic approach above requires additional features from the speculation mechanism. It is still desirable that an operator processes events as soon as possible, but now it should delay commits until it learns that also remote optimistic assumptions behind used events have been confirmed. When events are made visible outside the scope of the operator while still having unconfirmed optimistic assumptions, we name them *speculative events*. In addition, because failures are not expected to occur frequently, we also allow that later transactions use the results of earlier transactions that finished processing. We name this feature *speculative accesses*.

Finally, one limitation of many failure models, including the crash-failure model, is that failures of different components are assumed to be independent. This is not the case if, for example, failures are caused by software bugs. Therefore, we show how to extend a replicated operator to handle software faults caused by common bugs. This approach also uses speculation to reduce performance impacts introduced by fault tolerance.

In the following sections, we discuss the additional features needed in our event processing systems. After that, we show how these features can be used to implement low-latency fault tolerant ESP systems.

5.2 Requirements

In order to hide the costs of fault tolerance in a distributed ESP application, we need to be able extend speculation to other nodes. Therefore, we need to enable that speculative results are send and need to ensure that these speculative computations will only be committed after all pending speculations have been confirmed.

In this section we first show how to support speculative events. After that, we consider an optimization that consider that distributed speculation is normally correct and, thus, enables transactions to read values from other transactions. We name such operations speculative accesses. Finally, we discuss a basic requirement for active replication, reliable ordered broadcast, and show how it needs to be extended to make use of speculation.

5.2.1 Speculative events

In general, speculative events are events that were produced based on some optimistic assumption that may later be proven wrong. In addition, a speculative event is not guaranteed to be replayable. Thus, as before, an event represents something that happened in the system, but a speculative event will contain information that may be changed. Speculative events have two extra fields: the *version* attribute enables deciding which, between two speculative events, is the most recent

update; and, a *final* flag, which indicates that no speculative assumptions were made by the processing that generated this event.

Enabling stateful operators to support speculative input events requires special features from the speculation infrastructure:

i) When an operator carries out a computation that uses data from a speculative event, this computation cannot be committed; even if this is the next event to be committed.

ii) When an operator receives an speculative event it has to derive a timestamp for the transaction that will process this event.

iii) When an operator receives an speculative event with a version that is higher than the version used in previous computations, the previous event is overwritten with the new one. During this overwrite, the system should detect if any memory position that was previously read was modified. If that is the case, the computation should be reexecuted.

iii) When an operator receives an older version of an event it has already seen, this event should be discarded.

Generating speculative events

Discussing fault tolerance and the handling of speculative events requires additional detail of the internals of the events. The contents of a speculative event are summarized in Table 5.1.

Field name	*Type*	*Description*
txid	integer	id of the upstream transaction (i.e., the transaction timestamp) that generated the event;
id	integer	id unique among the events generated by the same transaction in the same execution (e.g., a counter).
oid	integer	id of the operator that produced the event.
rid	integer	id of the replica that produced the event.
seq	integer	Unique event sequence number for that incarnation of the operator.
ts	integer	Physical timestamp for the event.
version	integer	Version number for the event. Concatenation of the incarnation and the number of aborts of the transaction that generated that event.
final	boolean	$false$: event is speculative; *true*: event is stable.
app_evt	buffer	Application-level event.

Table 5.1: Contents of a middleware-level event.

Once a computation is finished in a speculation-enabled operator there are two possibilities. If the computation can be committed, the output events can be sent as final (i.e., the $final$ attribute flag is set). Otherwise, if the computation cannot be committed, the events will be sent as speculative. When sending events as speculative, the *version* field is a concatenation of the reincarnation of the operator (i.e., how many times it has failed and recovered) and the number

of aborts for the computation that generated that event. For example, the version number is a integer where the higher bits represent the reincarnation and the lower ones represent the number of aborts. This scheme guarantees that version numbers are strictly monotonic.

Speculative events (or final events that update a previous speculative event) carry an id for the transaction that generated the event. For the cases that the transaction generates multiple events, it carries an additional *id* field to identify itself among the other events generated. We initially assume that an input event generates either zero or one output event. We later show how to extend our approach to consider multiple output events.

Finally, we include the id of the operator that produced the event and, for active replication, the id of the operator replica. Events also carry sequence counters that are unique for the operator that produced it. This field is useful, for example, when requesting the replay of events or reestablishing a broken connection. In addition, in some applications events are processed in timestamp order. Therefore, events have a physical timestamp. Timestamps can be made unique by concatenating a timestamp that is unique within an operator with the id of the operator that assigned that timestamp.

Generating the transaction timestamps

When a speculative event arrives in an operator, a timestamp must be created for the transaction that processes it. This timestamp must be unique and must map different versions of the same event to the same transaction.

Our approach is to create a mapping between the tuple $(oid, txid)$ and a *task descriptor*. When an event arrives, the system checks if there is already an entry associated with the source operator id *oid* and the (upstream) transaction *txid*. If no such descriptor exists, one will be created. In order to guarantee that two entries for the same $(oid, txid)$ are not created in the case of multithreaded operators, the procedure of creating and initializing the operator acquires a lock. To avoid locking the whole list, an array of locks is kept, and the lock is chosen by hashing the $(oid, txid)$ tuple. In addition, this same lock is also acquired for other operations that read or modify this descriptor, as it will be described below.

The creation of the descriptor consists in two steps. First, we increment a counter in order to obtain a new unique timestamp. Second, we create a transaction with the new timestamp. This descriptor will then contain:

i) a pointer to the newly-created transaction;

ii) the timestamp of the newly-created transaction;

iii) the version of the input event;

iv) the value of the $final$ attribute of the input event;

v) a pointer to the input event;

vi) a pointer to a pending update input event (*pending_event*, initially empty);

vii) a pointer to the version of the pending update (*pending_version*, initially empty);

Note that the created transaction can be either speculative or nonspeculative (i.e., final). If the event was marked as final, the created transaction is also marked as final. Otherwise, the transaction is marked as *speculative*.

The creation of the task descriptor is also shown from lines 1 to 10 of Listing 5.1. Note that deciding if an update is late or not requires special care. An update may be taken from the input queue by a thread that loses the CPU for enough time that another update arrives and enables the transaction to commit. Thus, destroying task descriptors from committed transactions relies on the ordering of the final events and on knowing what the threads are doing. First, channels between two operators are FIFO and updates that make a transaction final are totally ordered by the *txid* field. In other words, the commit of upstream transactions is totally ordered, and thus, the final version of the events are orderly sent. Second, in order to discard old task descriptors, none of the threads should be holding a late update for such descriptor.

```
1  process_update(spec_event_t ev) {
2     task_lock = get_lock(ev.oid, ev.txid);
3     lock(task_lock);
4     task = retrieve_task_descriptor(ev.oid, ev.txid);
5     if (task == NULL) {
6        if (is_late_update(ev.oid, ev.txid))
7           discard_event(ev);
8        else
9           create_task_descriptor(list, ev); // creates also the transaction
10    } else if (ev.version < task.version) {
11       discard_event(ev);
12    } else { // ev.version > task.version
13       // update older event, restarting the transaction if necessary
14       if (ev.version < task.pending_version)
15          discard_event(ev);
16       else {
17          task.pending_version = ev.version;
18          task.pending_event = ev;
19          if (ev.final == TRUE) {
20             task.final = TRUE;
21             set_tx_flags(task, REFRESH | FINAL);
22          } else
23             set_tx_flags(task, REFRESH);
24          }
25       }
26    }
27    unlock(task_lock);
28 }
```

Listing 5.1: Updating task descriptors.

Handling speculative events

As explained above, in the arrival of an event for which there is no task descriptor, a new one is created. Otherwise, if such a descriptor already exists, the descriptor and the associated transaction

must be updated.

The update process has two phases. In the first phase, the system checks if the update is valid (i.e., it is not an old, late update), updates the task descriptor, and notifies the transaction. In the second phase, the original event is actually updated, triggering a reexecution of the transaction if necessary.

The first phase of the updating process is done by the thread that retrieves the update event from the input queue and is also summarized in Listing 5.1. The version of the newly arrived event is checked against the version in the descriptor. If the version of the newly received event is lower or the task descriptor is already marked as final, the newly received event is discarded. This can happen, for example, when two threads retrieve two versions of the same event from the input queue and the thread with the highest version makes progress faster.

Now assume a more recent version for a previously received timestamp arrives, that is, a valid update. The first step is to check if the event is simply a final marker for the earlier event. As we will see later, when using speculation to mask fault-tolerance delays, this is the most common case. In this case, the update will mark both the transaction and the task descriptor as nonspeculative. Note, however, that this does not necessarily mean that the transaction is allowed to commit. It still may be the case that transactions with lower timestamps still have to be committed.

The last possible update case is that the event is an non-final update. Then, the thread handling this update will get the lock to the descriptor and signal that there is a pending update (by setting the *pending_version* and the *pending_event* fields). If there is already a pending update, only the one with the highest version needs to be kept. After that, the transaction associated with that task descriptor is put in a $REFRESH$ state.

The work for the first phase of the update is then finished. After that, the thread checks if there are any paused transactions that should be continued before trying to get further events from the input queues. A paused transaction is to be continued when it is in the wait list (i.e., the list of transactions that are on hold) and all its dependencies have already committed (or finished).

The next step depends on the current state of the transaction. If the transaction is in the wait list, the thread that handled the update (or any other thread) eventually gets the transaction and proceed to the second phase of the update. Otherwise, the transaction must be running with some other thread. In this case, the thread processing the transaction eventually executes an operation through the STM (e.g., read or write, try to commit). At the beginning of such an operation the thread checks the transaction state. In this case, the $REFRESH$ flag will be set and the thread will execute the second part of the update. If there was any relevant change during this update, the transaction is aborted and reexecuted from the beginning.

The second phase of the update process consists in the following steps:

i) acquire the lock for the task descriptor;

ii) compare the update event with the original event;

iii) if a value is different then it is copied; the thread also checks if this position is in the read set of the transaction; if so, the transaction is marked to be reexecuted;

iv) at the end of the compare, the *version* field is updated and the *pending_version* and *pending_event* are reset;

v) the thread checks if the transaction should reexecute, commit, or go back to the wait list (e.g., if the update did not make it final or if there are still other earlier transactions waiting to be committed);

Instead of waiting for the a thread to execute an STM operation and check for pending updates, it is also possible to preemptively force the thread executing the transaction to update. In Linux, this can be done by sending a signal to the thread (*pthread_signal* [BC02]) and having the signal handler to apply the update and if necessary abort the transaction (which is implemented with a *long jmp* [BC02]).

One special case of an update is the empty event ($NULL$). An empty event is sent by the system when a transaction that previously generated an output event aborts and, during the reexecution, does not send any event. When updating a transaction with an empty event, the transaction is aborted and an empty transaction is put on the wait list. When the empty transaction becomes the next-to-commit, the next-to-commit counter is advanced and the empty transaction destroyed. If the aborted transaction had also generated an output event, it too generates a $NULL$ event to be sent downstream.

5.2.2 Speculative accesses

In the speculation mechanisms presented in the previous chapter, when two transactions tx_1 and tx_2 processing events e_1 and e_2, respectively, conflict, the transaction with higher timestamp (say tx_2) will abort and wait until the other (tx_1) commits. Nevertheless, in the current scenario, speculation is mostly about the nonoccurrence of failures. In this case transactions will be executed mostly in correct order, but will only be able to commit much later (e.g., when a checkpoint becomes stable in a remote node). Therefore, if tx_1 already finished processing, it would be more productive to let tx_2 proceed and read (or overwrite) memory positions that were used by tx_1. Later, if tx_1 commits, tx_2 will be able to commit. On the contrary, if tx_1 aborts, tx_2 will also have to abort.

For example, consider two transactions tx_1 and tx_2, with timestamp 1 and 2, respectively. Executing tx_2 after tx_1 has finished processing (even if it cannot commit yet) allows tx_2 to use speculative results from tx_1. However, conflicting out-of-order executions still cause the priority transactions to abort the low-priority ones. Further, if tx_2 tries to read from tx_1 while tx_1 is still active, tx_2 aborts and waits for tx_1 to finish.

Pseudocode for speculative reads and writes is shown in Listings 5.2 and 5.3. Some additional modifications are also necessary for the abort function: when a transaction is aborted and must release its locks, it now only releases locks that still belong to it, ignoring stolen locks.

In a summary, for a reading operation there can be up to two versions of a memory position: the version from the last committed transaction that modified that position (i.e., the version actually in the memory); and, the version from the last non-committed, but finished, transaction that modified that same position (i.e., the version in the write set of the transaction holding the lock for that position). In addition, although transactions only read values from finished transactions

```
1  read(position_t pos) {
2     restart:
3     lock = get_lock(pos);
4     if (lock.locked) { // Some tx wrote to pos
5        if (lock.owner == this) {
6           value = get_value_from_wset(this, addr);
7           return value;
8        }
9        tx = lock.owner; // tx -> conflicting transaction
10       if (tx.timestamp < this.timestamp) {
11          if (!is_precommitted(tx)) abort();
12          // consider the modifications from the other
13          create_dependency(this, tx); // if tx abort, this one aborts too
14          value = read_from_wset(tx, pos);
15          if (get_lock(pos) != lock) goto restart; // something changed
16          update_read_set(this, pos, tx.timestamp);
17          return value;
18       }
19    }
20    // ignore the other (or there is not other), use the timestamp of the last
            committed tx that modified the position
21    version = lock.version;
22    value = read_from_memory(pos);
23    if (get_lock(pos) != lock) goto restart; // something changed
24    update_read_set(this, pos, version);
25    return value;
26 }
```

Listing 5.2: Speculative read.

```
1  write(position_t pos, value_t value) {
2     restart:
3     lock = get_lock(pos);
4     if (lock.locked) {
5        // Some tx wrote to pos
6        if (lock.owner == this) {
7           update_writeset(this, addr, value); // we wrote to it before
8           return;
9        }
10       tx = lock.owner; // tx -> conflicting transaction
11       if (tx.timestamp > this.timestamp) {
12          // this one has priority
13          abort(tx); // abort other, release locks
14          if (steal_lock(pos, tx, this) != OK) goto restart; // try again
15       } else {
16          // the other has priority
17          if (!is_precommitted(tx)) abort();
18          // replace the locks atomically
19          if (steal_lock(pos, tx, this) != OK) goto restart; // try again
20       }
21    } else {
22       lock2 = create_lock(this, lock.version);
23       replace_lock(pos, lock, lock2);
24       add_to_writeset(this, lock2);
25    }
26 }
```

Listing 5.3: Speculative write.

(i.e., intermediary values are not seen), consistent snapshots are not guaranteed anymore as there may still be transactions older than tx_1 that were not finished yet (i.e., a transaction may see a set of values that would never have coexisted in memory as it sees both committed values and values that are not guaranteed to be eventually committed).

5.2.3 Reliable ordered broadcast with optimistic delivery

Active replication requires that replicas process the same input events in the same order (see Section 2.3). This can be guaranteed by a reliable ordered broadcast, also known as atomic broadcast [CASD95, CT96]. However, to benefit from optimism we use an atomic broadcast protocol that benefits from speculation. Such a protocol is called optimistic atomic broadcast [PS98].

Intuitively, the optimistic atomic broadcast may deliver messages multiple times. The so-called optimistic deliveries consider an speculative ordering for the messages. The last delivery is marked as final and consider an ordering of messages that is consistent among all nodes. Further, the final ordering of a message does not need to match its previous optimistic orderings.

In our case, events delivered optimistically are marked as speculative and is the role of the speculation support that, if the ordering of events are different between the optimistic and the final deliveries, all effects visible on the state of the operator are reviewed to match the final ordering.

An atomic broadcast protocol satisfies the following properties:

1. *Validity*: if a correct node broadcasts a message m, then it will eventually deliver m.
2. *(Global) Agreement*: if a correct node delivers a message m, then all correct processes eventually deliver m.
3. *Uniform integrity*: every process delivers a message m at most once and only if m was previously broadcasted by some processes.
4. *Total order*: if two correct processes p and q deliver two messages m and m', then p delivers m before m' if and only if q delivers m before m'.

We then slightly change this ordering property so that it does not apply to optimistically deliveries, but only to final ones, and add a *local agreement* property (similar to [KPA+03]) to ensure that a speculative processing will eventually become final.

4. *Total final order*: if two correct processes p and q do the final delivery of two messages m and m', then p does the final delivery of m before m' if and only if q did the final delivery of m before m'.
5. *Local agreement*: if a processes does an optimistic delivery of m, it also does a final delivery of m.

We support two approaches for implementing atomic broadcast. In the first approach, we assume that messages have unique ids and that nodes execute an *agreement protocol* to order the messages. In the second approach, we assume that the event sources (or the input adapters that receive the messages and convert them to events) have *approximately synchronized clocks*.

Consensus-based atomic broadcast with optimistic delivery

Pseudocode for an optimistic atomic broadcast protocol is shown in Listing 5.4. The atomic broadcast is a service executed in all replicas of an operator. The service receives messages from the sources by retrieving them from the input queue. The messages are appended to a list and speculatively delivered. Then, a consensus protocol with two phases is started. In the first phase it tries to lock a value and in the second phase it decides on that value if no failures occur. Because the locked value has a high change of being the final ordering, it is useful to deliver this order as speculative again. If the current leader of the consensus protocol fails, it may not be possible to decide on the currently locked ordering proposal and another round of consensus starts.

```
1  atomic_broadcast() {
2     while(TRUE) {
3        while (!is_input_buffer_empty() && msg_list.size() < MAX_BATCH) {
4           msg = get_msg_from_input_buffer();
5           append_to_list(msg_list, msg);
6        }

8        optimistically_deliver(msg_list); // deliver the estimated order

10       while(!finished) {
11          decision = consensus_estimate(msg_list); // refine estimate
12          optimistically_deliver(decision);
13          finished = consensus_decide(); // decided on estimate?
14       }
15       finally_deliver(decision);
16       msg_list = remove(msg_list, decision); // remove delivered messages
17    }
18 }
```

Listing 5.4: Optimistic atomic broadcast.

The atomic broadcast protocol above is based on consensus, we use a leader-based consensus protocol that rely on failure detectors that do not wrongly suspect correct processes [VR01]. A node is elected to be leader of the consensus (the node with the lowest id) and the leader and the other nodes run different protocols. For completeness we include the pseudocode for the consensus in the appendixes. Because the atomic broadcast protocol above relies on consensus for all deliveries, we refer to it as *consensus-based atomic broadcast*.

When using the above protocol, we use the order of delivery as the timestamp of the transactions that will process the events. This approach is not optimal because it may cause unnecessary conflicts. For example, if three events e_1, e_2, and e_3 are delivered optimistically in this order, but finally delivered as e_2, e_1, e_3. Both e_1 and e_2 are aborted because e_2 overwrites e_1 and e_1 overwrites e_2. These abort occur even if the two events did not conflict. It is possible for the speculation infrastructure to use the event ids (e.g., the *oid* and *txid* fields) to derive the changes in the orderings and then use these changes to change the timestamps of transactions. In this case, transactions would only abort when changing the timestamp of the transaction is not be possible (e.g., because of a dependency). However, in this work, we do not consider this optimization.

Time-based optimistic atomic broadcast

Our second approach to implement optimistic atomic broadcast protocols is based on research about deterministic merge [AS00] and transparent recovery [HCZ08]. The approach is based on four assumptions: (*i*) the graphs of operators is static; (*ii*) channels between operators are FIFO; (*iii*) sources have approximately synchronized clocks; and (*iv*) events have unique timestamps. Assumptions *i* and *ii* were already part of our model. Assumption *iv* is trivially implementable by forcing timestamps to be unique within a node and them appending the node id in the least significant bits. Finally, assumption *iii* is nontrivial, but is can be implemented in a local (or system) area networks with NTP.

In fact, it is not strictly necessary that the sources themselves have synchronized clocks, but only local clocks with bounded drift rates. The replicas (the nodes receiving the messages) could, through an agreement, compute common *adjustment values* that all replicas apply to make the timestamps of the events from different sources, comparable. For example, assume two sources feed events to two replicas. If the two sources have local clocks that are ten seconds apart, the replicas could agree to subtract ten seconds from the timestamps of the events coming from the source with the most advanced clock. An algorithm for solving this problem was proposed by Aguilera and Strom [AS00]. Here, we assume for simplicity that sources have synchronized clocks.

In this approach, the replicas are completely independent as long as there are no failures from one of the sources. Nodes keep two lists and a timestamp value: a list *msg_list* with the messages that were optimistically delivered, but not finally delivered; a list $upstream\_sources$, which contains the list of upstream nodes that generate events to be consumed by the current node; and a variable $last\_ts$ with the last timestamp that was finally delivered.

The failure free case in each replica runs as follows:

i) Initially, the list *msg_list* is empty and $upstream\_sources$ has all upstream nodes that feed the current replica.

ii) A thread responsible for receiving and delivering messages listens to the connections with the event sources and with the other replicas.

iii) When the thread receives a message *m* from a source *s*, it checks if *s* is in the $upstream\_node$ list. If *s* is not in the list, it discards the message and goes back to step *ii*.

iv) If *s* is in the list, it checks if *m* is already in the list or if its timestamp is lower than $last\_ts$. If one of these conditions is true, it also discards the message and goes back to *ii*.

v) It delivers *m* optimistically and forwards *m* (unchanged) to the other replicas. This forwarding is only useful in case of failures and can be replaced by an on-demand approach. Next, it adds the message to list msg_list.

vi) Being $m.ts$ the timesamp of a message *m*, the thread checks if it has in *msg_list* a message m' from each node in $upstream\_sources$ such that $m'.ts \geq m.ts$. If so, it can do the final delivery of all messages in *msg_list* that have timestamp lower or equal to $m.ts$. The delivery order is the order of the message timestamps.

vii) It then removes all delivered messages from *msg_list* and set *last_ts* to *m.ts*.

In case one of the sources fails, the replicas need to execute a consensus protocol. The consensus determines which is the last message from the failed source that needs to be considered. After this consensus, when a replica receives this last message, it removes the failed source from the *upstream_node* list. The replicas that do not have that message can simply wait (if the messages are also being forwarded by the replicas) or request the message. If there are multiple messages they will also be received (because of the FIFO channels). In addition, if a replica already has the latest message, it checks its *msg_list* to check for messages that it can deliver now that it does need to wait for the failed source.

The above approach provide all the guarantees needed for our optimistic atomic broadcast:

1. *Validity:* if a correct source sends an event e, the reliable FIFO channels (e.g., TCP) guarantees that correct downstream replicas will receive e.

2. *(Global) Agreement:* if a correct downstream replica delivers an event e, then it also forwards e to all other downstream replicas (spontaneously or on request). The *Validity* property guarantees that all other replicas deliver e.

3. *Uniform integrity:* every process optimistically delivers a message m only the first time m is included in the *msg_list*; as a consequence they also finally deliver m only once.

4. *Total final order*: if two correct processes p and q do the final delivery of two messages m and m', the delivery order is the order of the timestamps, which is totally ordered.

5. *Local agreement*: if a replica does an optimistic delivery of e, it will finally deliver e either when (*i*) it receives events from the other upstream sources with timestamps higher than $e.ts$; or (*ii*) it detects the failure of the other sources and learn through the consensus that it does not need to wait anymore.

Because the above protocol relies mostly on timestamps, we refer to this protocol as *time-based atomic broadcast*. When using the time-based protocol, the timestamp of the transaction that process the event is the timestamp of the events themselves. This timestamp does not change even if the delivery order changes. In contrast to the consensus-based atomic broadcast, the time-based approach is naturally robust against unnecessary aborts due to slight changes in orderings. However, as a disadvantage it may suffer from unnecessary delays when the sources do not produce events in the same rate.

5.3 Applications

5.3.1 Passive replication and rollback recovery

Consider again our running example, in Figure 5.1 we zoom into the neighborhood of the `Processor1` operator. As discussed earlier, this operator has a local state that was constructed based on the events received so far and we must be able to reconstruct this state in case of a

failure. The reconstruction of the state is based on events received from the two sources and, because the order of the events received from each source matters, the input events (or at least their ids) are logged as they are received[1]. If speculation is used, the order of the logging is then used to generate timestamps for the transaction that will process the event.

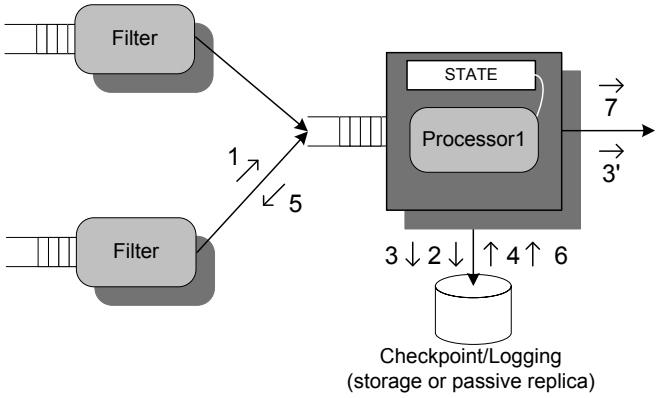


Figure 5.1: Basic protocol for checkpoint-based fault-tolerance.

In addition to the ordering of the events, it is also possible that the computations on the operator depend on some nondeterministic decisions. Some examples of nondeterministic decisions are random numbers, physical time during the processing of the event, and even scheduling of threads in a multithreaded operator (if these decisions may affect internal data structures). These nondeterministic decisions are also logged. Furthermore, to avoid that the log grows indefinitely the operator is also periodically checkpointed. With a checkpoint, the complete state of the operator is saved to a stable storage and all the logs are cleaned.

Alternatively, if the operator logs only the ids of the messages, it restores the checkpoint and requests a replay for messages since the beginning of the log. It then uses the log only to enforce the ordering of messages from the different sources (and if needed, to replay the nondeterministic decisions).

Note that independently if the node logs the complete messages or only the ids and sources, the upstream nodes must still keep a buffer of messages sent but not yet acknowledged to be stable. If whole messages are logged, the acknowledgment from the downstream node will be sent when the log becomes stable. Otherwise, if only the ids and sources are logged, the acknowledgment is only sent after a complete checkpoint of the operator is stable.

Conventional approach

A system implementing a conventional checkpoint-based fault-tolerance approach runs as follows:

[1] As with the time-based atomic broadcast, if events have timestamps, a deterministic merge of the stream could also be applied. Nevertheless, there are scenarios where it is important to bound the maximum time that an event can wait before being processed and, thus, logging is required to keep determinism.

i) when the `Processor1` receives an event from one of the upstream operators (e.g., an event in message 1 in Figure 5.1) it logs the event (e.g., through control message 2 in the figure);

ii) next, during the actual processing of the event, other nondeterministic decisions may be taken and they also need to be logged (e.g., through message 3);

iii) eventually, the logged input event (step *i* above) will be stable in the storage (as notified by message 4) and if the complete events are logged, the operator may notify (through control message 5) that such event will never be requested to be replayed again; the upstream node may then remove that event from its output buffer;

iv) later, the nondeterministic decisions used during the processing will also get stable (as notified by message 6);

v) finally, the results of the processing can be sent downstream (illustrated by the output event in message 7).

The operator must wait until the log is stable on disk before sending outputs, otherwise events could be processed in a different order during a replay (or use different nondeterministic decisions) and the operator may reach a state different from the one expressed by the previous outputs. Thus, although the new state would be also valid, the inconsistency could have critical consequences.

When recovering from a failure, the operator restores its latest checkpoint and replays the messages stored in the log. Then it uses the last sequence number in the log from the lastest event from each input channel to request upstream nodes to replay messages starting at that point. The `Processor1` operator may output duplicate events, events that had already been emitted before the failure. Nevertheless, because any ordering or additional nondeterminism was logged before outputs were sent, the duplicates will have the exact same information as their first instances (including ids) and can be silently dropped by downstream receivers.

Logging events and nondeterministic decisions

The logging of information to the stable storage used in the protocol above works as follows. Logging requests can be issued to log either events (or source-id pairs) or to log nondeterministic decisions taken during the processing of the event. These log requests are asynchronous, they return as soon as the request has been enqueued.

Later, when the processing of the event is finished and the resulting events are ready to be sent, they are blocked until the logging requests have been committed to disk. The storage requests can be handled by a set of threads that can write to different data storages in parallel in order to maximize throughput. Thus, if the user configures N storage points (e.g., local disks, NFS mounted disks), there will be one thread per storage point plus 1 extra thread that collects and groups requests while the others are busy, thus, $N + 1$ threads are used in total. When a thread finishes writing a set of decisions, it releases its associated storage point and hands it to the thread that was collecting requests before taking over the role of waiting for new requests.

Eventually, when all the storage requests associated with some output event are stable, this event may be forwarded.

Speculative approach

In order to illustrate how speculation works and how the fine grained control enabled by the STM can help, consider again our example from Figure 5.1. The approach below is based on speculative events. Recall that computations based on speculative events cannot be committed until a final version of this event is received.

Consider initially a first scenario in which the `Filter` operators do not issue speculative events, only regular events. The system works mostly as the conventional approach detailed previously. However, once the computation is done the outputs are immediately emitted as speculative, without waiting for the confirmation that the logging or checkpoiting is stable (i.e., an speculative output event is in message 3′). Later, when logged messages are stable in the storage, the output events are reemitted as final (in message 7).

Consider now a second scenario where the `Filter` operators may also generate speculative events. This will be the case when they also need to do logging, for example. In addition, operators that are both deterministic and stateless may process and forward all events without distinguishing between speculative and final. Thus, it is also possible that the `Filter` operator simply forwards both speculative and final events from its upstream node.

Assume that the upper filter in Figure 5.1 emits a version of an event e_1, namely e_1^s. Then, the other filter emits a final event e_2. These two events cause transactions to be created in the `Processor1` operator. Initially, assume no conflicts occur, the two events do not access the same part of the operators's state. Event e_1^s will be processed. Nevertheless, because event e_1^s is speculative, *its computations cannot be committed*, even after the logging is stable. Further, if an output event is generated, it is also flagged as speculative. After that, event e_2 is processed and an output event is generated. Although event e_2 is not speculative and did not use any state that had been speculatively modified, it cannot commit and its outputs are marked as speculative. The processing of e_2 cannot be committed because it is ordered after e_1. Nevertheless, as we will see later, if the upper filter fails and its transactions need to be aborted, the processing of e_2, as well as its outputs, will not be affected.

In the third scenario, assume there is a conflict, more specifically event e_2 has read from a part of the state that was modified during the processing of e_1^s. In this case, the output event generated by the processing of e_2 will be marked as speculative and a dependency will be created between the two transactions. In this case, a failure of the upper filter will also cause the processing of e_2 to be aborted and reexecuted.

In a summary, when the `Processor1` both learns that its logging operations finished and receives a confirmation for the speculative input event (turning it into final), it too can immediately forward an indication that the respective previous speculative output is now final. This indication is then logged to stable storage only in order to simplify recovery.

Note that if instead of a final event confirming the speculative event, the `Processor1` operator receives a final update that somehow contradicts the previous version, the new event has to be logged and the computations (potentially) repeated. Only after the new logging and computations are finished, the final events can be forwarded.

Dealing with failures in the speculative case

If the `Processor1` operator fails, the failed operator will recover as in the nonspeculative case: it restores the latest checkpoint and uses the log to locally replay events and reconstruct its state. After that, the operator uses the logged sequence numbers to signal upstream nodes from which point they should replay the stream.

However, it can be the case that the latest events on the log are still marked as speculative. In this case, the `Processor1` reprocesses the (speculative) events during the local replay and during the replay from the upstream nodes, will either receive confirmations or updates for these speculative events. Thus, even if the last log for an event (which confirms the speculation) was not stable, the operator will now receive the same confirmation. As a consequence, even if it does not know that it previously sent an event as final already, it will send the same event with the same content. Therefore, although the downstream node will ignore this events (it knows it has already received the final version for that event), this will cause no inconsistency as the new event has the same content (only the version number will be different as it now considers the new incarnation).

The other possible case is that the event was indeed not final (e.g., the decisions used for the production of an event were never stable). Therefore, it must be that no final version of it was ever sent. In this case, a different event may be sent (i.e., a different event for the same *txid*).

Consider an example to illustrate this recovery. Suppose operator `Processor1` receives an (final) event from one of the upstream filters, processed it in a transaction with timestamp t, and emitted an speculative event while waiting for the logs to get stable. `Processor1` then fails before the log becomes stable and during recovery, it restores the checkpoint and start processing events replayed by the upstream nodes. It then assigns an input event to transaction timestamp t. It is possible that another event (e.g., from another channel) gets that same transaction timestamp t. Alternatively, the `Processor1` operator could also had assigned the same event to the timestamp but during the processing, it could had taken different nondeterministic decisions. In both cases, from the viewpoint of the `Processor2` operator, downstream of `Processor1`, a new event containing a previously seen *txid* will be received. This new event triggers the procedure for updating speculative events.

Because the version of an event incorporates the incarnation of the operator that produced it, the new event will have a higher version. Two conditions are then possible. First, the update event may have the same content as the previous event. This is the case, for example, when the logging contains the ordering information and during recovery, the same ordering was used by coincidence. The update procedure will then trigger no reexecution and as soon as the event in the input of the `Processor2` operator is final, the output event (if any) will also become final. Second, if the content is different and this difference was relevant to the computation, the update proceedure will cause the transaction to be aborted and reexecuted. As a consequence, all transactions that had dependencies with this transaction will also be aborted. However, transactions with no dependencies will not be aborted even if they have higher timestamps.

One final practical aspect to be considered is that after a failure, the recovery of a node may take long. Therefore, instead of having the downstream node wait for the recovery and then compare the events, all transactions from the failed node are aborted. Consequently, the node downstream of the failed node does not block waiting for recovery and can proceed computing events comming from another sources.

5.3.2 Active replication

As with the passive replication, assume that the `Processor1` operator in our running example is to be made fault tolerant. In Figure 5.2, we show the replicated operator. We assume that the operator is deterministic. In addition, we assume that the `Processor1` operator uses an atomic broadcast protocol as detailed earlier to deliver messages.

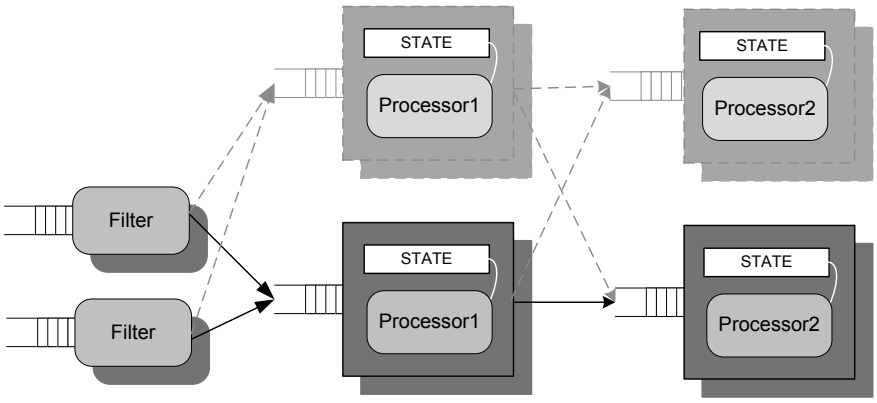


Figure 5.2: Replicated operator.

The most basic requirement for active replication is that replicas process the inputs in the same order. We have previously detailed two approaches for implementing atomic broadcast, a time-based and a consensus-based approach. When implementing active replication by the conventional, non-speculative, approach, optimistic delivery cannot be used and is simply ignored. Moreover, because there is no event updates, there is no need to assign unique ids for the tasks processing the events. Therefore, the conventional approach is oblivious to the atomic broadcast used.

In the speculative approach, events are can be optimistically delivered and the system needs to assign unique ids (or timestamps) to the tasks (i.e., transactions) that process the events. Thus, if the time-based atomic broadcast is used, the physical timestamps of the events will be used for the transaction timestamps. In contrast, the consensus-based atomic broadcast requires neither unique timestamps on the events nor approximately synchronized clocks in the sources. In this case, the delivery order is used as transaction timestamp. If we consider only final deliveries, the both speculative approaches behaves exactly the same as the conventional approach. Nevertheless, with speculation-enabled operators, there are some points where speculation can reduce latency. We discuss each case separately below.

Conventional approach

When an event is finally delivered to a non-speculative operator it can be processed an its outputs will be final. Replicas will accumulate the same state and produce equivalent results (i.e., results may differ only in the metadata, like the id of the replica that produced it). In addition, replicas will generate the events in the same order. Therefore, downstream nodes can trivially eliminate

duplicated events.

In case of failures of one of the replicas or one of the upstream nodes, there is no visible impact. The atomic broadcast tolerates failures and will keep delivering messages to the surviving replicas. Moreover, the downstream nodes will continue to receive the events from the surviving replica and the application will keep running. During recovery, the failed replica can be recovered by copying the state from the surviving replica.

Speculative approach with consensus-based atomic broadcast

Consider initially the case of a single-threaded operator replicated on two nodes, R_1 and R_2. First, recall that the atomic broadcast is based on a leader, say R_1. Thus, when the atomic broadcast protocol running on R_1 receives a set of messages it delivers these messages optimistically and proposes its ordering to the other replicas, in this case R_2. R_1 cannot use this ordering as final yet. It cannot produce final events because if it fails before R_2 commits to use this ordering, R_2 could use a different ordering. As a consequence, inconsistencies could occur: the state of R_2 would not reflect the state that downstream replicas expect it to be considering the previous (final) events from R_1. Nevertheless, if R_1 does not fail, its ordering will eventually become final. Therefore, the speculative computations in R_1 will probably be confirmed.

Second, the node running replica R_2 receives input events and the communication protocol delivers them optimistically. Immediately, R_2 can start processing these events speculatively. There are two scenarios where these speculations have high chance of success: (*i*) if the system is under low load, the variation in the arrival time of the events on different nodes will be lower than the inter-arrival times of the events, thus events will be naturally ordered; (*ii*) if the system is under very high load, for example, during a burst of events where the inter-arrival times is lower than the time needed to process them, the events will accumulate on the input queues; they can then be deterministically picked[2]. Nevertheless, R_2 will sometimes speculate incorrectly, but R_2 learns the final order first (at the end of the first consensus round) and, thus, has more time to rollback and reexecute computations.

The third impact of speculation regards speculative events sent downstream. Consider replica R_1. This replica starts processing events speculatively as soon as they arrive and, unless it fails, this speculation will succeed. As a result of the processing, R_1 may generate speculative events. Once again, if this replica does not fail, these speculative events will have the exact same content of the final events. Thus, if these events are forwarded downstream, the next component can already process them (also speculatively). Later, if no failures occurs, the final events will be identical to previously sent speculative events and will not need to be reprocessed. An additional gain is then achieved by overlapping computation on downstream nodes with the atomic broadcast. Thus, speculative events from replicas other than the one with the lowest id should be either discarded by the communication infrastructure or ignored by the downstream nodes. This optimization reduces both network load and overhead on the downstream nodes (by allowing them to only consider speculative events that are most likely to become final).

[2]Deterministic ordering in this case can be implemented by having one single priority queue for all events from all connections arriving at an operator, and ordering the events by their ids.

Speculative approach with time-based atomic broadcast

In the case of the time-based atomic broadcast, replicas are allowed to start speculative processing as soon as events are delivered optimistically. In addition, the only way in which the optimistic order can be changed is if one or more events arrive late and are inserted among the previous events. Nevertheless, as long as the conflict rate between transactions is small (i.e., there is parallelism in the workload-operator combination), these new events will have a low impact on the speculative computations.

When a replica finishes processing an event speculatively, it can send speculative output events. Assume that this speculation has a high change of succeeding. In this case, events carry the same timestamp as the transactions that processed them and there will be also a high change that these events will be confirmed. Therefore, downstream nodes can profit from preprocessing these speculative events.

Note that because the timestamps of the transactions are deterministic, if conflict rates are small, there is a high chance that two speculative events from different sources that have the same timestamp will also have the same contents.

Multithreaded operators

One considerable limitation of active replication is that it cannot easily support multithreaded operators. Typically, to support multithreading, relevant scheduling decisions (e.g., lock acquisition and release, or atomic operations used by lock-free algorithms) need either to be made deterministic or to be communicated and enforced in the remote replicas. Collecting and enforcing these scheduling decisions is costly.

In the case of our STM-based speculation, the many possible interleavings of concurrent threads need to be neither communicated nor guessed as the STM causes executions to appear atomic. As a consequence, all the potential scheduling interferences will be masked away by the STM and can be uniquely determined by the ordering of the messages in the processing sequence, exactly as in the single-threaded case. Consequently, although STMs are often not as efficient as hand-optimized parallel code, STMs are useful because they greatly simplify the process of developing parallel code for active replication. In addition, once the costs of collecting and enforcing the scheduling decisions need to be considered, the relative overhead of the STM is considerably reduced.

5.4 Extensions

5.4.1 Active replication and software bugs

Both passive and active replication approaches work well for the crash-failure model. Nevertheless, they assume failures are not correlated. This may not be the case if, for example, software bugs need to be considered. An operator may crash because one specific input triggered an existing software bug. Software bugs are common in practice, specially when code needs to be developed under high time or budget pressure.

In the presence of software bugs that cause an operator to crash, conventional active and passive replication approaches could be completely ineffective. Passive replication could handle the recovery of the operator, reconstructing the state before the crash and then reprocessing the same inputs. Nevertheless, the replayed execution is likely to reach the same point where the previous failure occurred, leading to a new failure. Similarly, in active replication, all replicas could be simultaneously affected and crash at the same time.

In this section, we provide an extension to our speculative active replication approach. The goal of this extension is to provide an example of how speculation can also be used with broader failure models. Here, the system provides resilience against wrong results and corruption of the state in the operators. Speculation is then used to reduce both processing latency and expected recovery time when using known software fault tolerance approaches (i.e., enhancing code with runtime checks [RL04, ACCH09, NS05]).

Assumptions

For this extension we consider faults caused by sets of commons software bugs that can be detected by the execution of additional runtime checks. In addition, we exclude incorrect operators and inputs that are maliciously crafted in order to compromise operators. Further, operators are single threaded and *do not* use optimistic delivery or out-of-order processing. Nevertheless, events are still processed inside transactions and these transactions can only be committed when an external commit command is received for that transaction.

With an active replication approach that only needs to tolerate crash failures, all replicas were equal. In contrast, we now consider two different types of replicas: (*i*) checker replicas, running special *strengthened* versions of the operator, tolerate a certain class of common software bugs; different types of checker replicas may detect different software bugs; (*ii*) the original replica, running a non-instrumented version of the operator (i.e., the original user-provided code). In addition, we assume that there are no major bugs in the event-processing framework code (which can be sufficiently tested), but only in the user-provided code for the operations. Because the checker replicas execute additional computations (to detect and tolerate the failures), we consider that they are, on average, slower than the original operators.

Operator are able to do a complete recovery. This recovery can be done by restarting the operator and copying the most recent state from another replica or by restoring a local checkpoint and replaying the missing events (e.g., like the traditional checkpoint-based recovery discussed earlier).

Finally, in this model, we allow that an input event that causes a software fault is discarded without being processed. This is necessary as the algorithm itself is likely to be unable to process such an input and, thus, no other recovery is possible.

Detecting and notifying software faults

A traditional approach for handling the problem of bugs in software is to have a special compiler pass or a runtime environment that inserts checks for common programming bugs. Consider for example out-of-bound accesses, where a memory access directed at an array actually falls into a

position outside the bounds of this array. Although apparently simple to deal with, out-of-bounds accesses are responsible for major problems even in mature software [RL04, ACCH09].

In this extension, we take the code of the original operator and generate a checked version of it. This derived operator, named *checker*, will execute the same algorithm, but it is equipped with runtime checks that detect out-of-bounds accesses and allow an action to be taken before the access is executed.

To track the bounds of buffers, allocation operations (like `malloc()`) are wrapped to associate the returned pointer p with the bounds of the allocated buffer. At runtime, the checker maintains the memory bounds of all buffers in a table. To be able to lookup the buffer bounds of a pointer, each pointer carries the index of its associated buffer in this table. To improve performance, we store the index in the upper 16 bits of the 64-bit pointer value (similar to [ACCH09]). The associated bounds are propagated to new pointers derived from p by copying or pointer arithmetic. Finally, a runtime check is inserted at every pointer dereference, which verifies that the pointer is within its associated bounds.

The notification of an invalid access depends on the position where the invalid access actually falls into. Faults whose effects can be rolled back by simply aborting the transaction are called *minor faults*. Faults whose effects are not undone even if the current transaction rolls back are named *major faults*.

Some examples of minor faults are the following:

i) while an operator is accessing the state, it executes an operation that causes an exception, for example, a division by zero (generating a floating point exception in Linux) or a dereference of a $NULL$ pointer (causing a segmentation fault);

ii) while executing the code in Listing 5.5, an invalid $position$ value in the input event causes the operator to do an out-of-bounds access in line 11 and modify a position within the $last\_update\_time$ array instead of one in the *value* array;

iii) while executing the code in Listing 5.5, the operator reads an uninitialized position (using random values could cause replicas to diverge and create inconsistencies);

iv) an unexpected value in an attribute of an input event causes an infinite loop.

In contrast, examples of major faults are:

i) while writing to the local-scope array $buffer$, an out-of-bounds access causes the operator to corrupt the stack (as we with regular speculative operators, accesses to local variable are not instrumented by the STM);

ii) using a nondeterministic value (e.g., a non-initialized pointer) in a jump instruction (the checker cannot assume that execution of unknown code in a replica did not cause an unrecoverable memory corruption).

Our failure model is, therefore, defined in terms of the checkers that are attached to the system. The system tolerates a certain class of faults as long as one replica executes a checker that: (*i*) detects that class of faults, providing an output to the framework that indicates if the fault was

```
1  typedef struct {
2     int position, update, time;
3  } event_t;

5  static int values[VECTOR_SIZE];
6  static int last_update_times[VECTOR_SIZE];

8  // Process the events
9  process(event_t *ev) {
10    char buffer[BUFFER_SIZE];
11    values[ev->position] += ev->update;
12    last_update_times[ev->position] += ev->time;

14    ... // periodically produce an event with the changes in the values buffer
15 }
```

Listing 5.5: Sample operator with an out-of-bounds bug.

major or minor; and (*ii*), prevents that failure from compromising the replica running the checker. In addition recovery works as follows:

i) when a checker replica detects a minor failure during the processing of event *e*, that replica permanently aborts the transaction that processed *e* and discards the event; all other replicas *abort the transaction* that processed *e*;

ii) similarly, when a replica detects a major failure during the processing of event *e*, that replica permanently aborts the transaction that processed *e* and discards the event; all other replicas *restart the operator* and initiate a complete recovery (e.g., by copying the state of the surviving checker replica or by restoring a local checkpoint and during replay, ignoring the event that caused the fault);

Conventional approach

Assume that the original replica is faster as it does not execute extensive runtime verifications. In a system without speculation all failures are major failures. Consider, for example, a system with two replicas. One replica is the original operator and the second replica is instrumented for out-of-bounds checking [BWS+10]. Both replicas process the input events. When the (slower) checker replica detects that an input event caused a minor fault, this event was already processed by the (faster) original replica and corrupted its state. Therefore, the original operator must do a complete recovery.

In addition, it is possible that the checker replica detected the failure after making some modifications to the state of the operator. Because there is no support for rolling the operator state back, the checker thread also needs to execute a complete recovery.

Speculative approach

The creation and processing of transactions in this extended active replication is similar to the crash-tolerant active replication. When an event e is delivered, a transaction with timestamp t is created. Because only the final ordering of the atomic broadcast is used, the timestamp t for the transaction that processes event e will be the same in all replicas. Moreover, in this case we do not allow speculative reads. Speculative read accesses (i.e., transaction reading from finished, but not committed, transactions) could cause invalid data to corrupt another transaction in a way that is not necessarily the same in all replicas.

The differences between this extension and the crash-tolerant active replication are the following. First, results of the replicas are voted. This voting is done in the downstream replicas. When the downstream replica receives all the results for a timestamp t: (*i*) if at least one of the results indicate a *MAJOR* fault, the voter forwards the message to all upstream replicas; otherwise, (*ii*) if at least one of the results indicate a *MINOR* fault, the message is also forwarded upstream; finally, (*iii*) only if all results do not indicate a failure, the voter sends an OK_t message that allows upstream replicas to commit values.

Second, the checker replica will, as in the non-speculative case, process its input events. It is, however, also enhanced with speculation. Thus, it does not need a checkpoint. If during the processing of the input event with delivery order t, the checker detects a fault, it produces a $MINOR_t$ or a $MAJOR_t$ output event, according to the type of the failure detected, and permanently aborts the transaction. When a transaction is permanently aborted, its computations are rolled back and the input event that caused the transaction to be created is discarded. The $MINOR_t$ or $MAJOR_t$ event is then sent to the downstream neighbors. If, however, no fault was detected, the normal output events generated by the computation (or an empty event if no output is generated) are forwarded to the downstream neighbors and the checker replica waits for response from the downstream voter to decide if it can commit transaction t, if it should abort the transaction, or if it should do a complete recover (e.g., if another checker detected a major fault).

Third, the original, non-checker, replica also processes its input events. However, it delays commit of a transaction t until it learns from an OK_t message that no faults were remotely detected for this timestamp. Meanwhile, the non-checker replica forwards its results speculatively to the downstream operator. Eventually, the non-checker replica will receive either an OK_t, a $MINOR_t$, or a $MAJOR_t$ message. If it receives an OK_t message, it commits transaction t. If it receives an $MINOR_t$ message, it permanently aborts transaction t. Finally, if it receives a $MAJOR_t$ message, it executes a complete recovery, retrieving state from a local checkpoint of from a remote checker.

It is also possible that a replica takes longer to process a message. This would be the case, if reading from a invalid memory position caused a infinite loop, for example. In this case, if the replica receives an indication that a fault was detected by a remote replica, it can immediately abort the current transactions and execute the necessary recovery.

Last, the downstream operator can receive events from either checker or non-checker replicas. Normally, results from non-checker replicas will arrive first as processing in these replicas has less overhead. If this is the case, the downstream component can start processing these events speculatively. Eventually, the downstream operator will receive an regular event, a $MINOR_t$, or a $MAJOR_t$ message from the checker replicas. If it receives an regular event (i.e., no faults detected) from all upstream checkers, it may commit any speculative processing that was based

on the speculative input events. Otherwise, if it receives at least a $MINOR_t$ or $MAJOR_t$ message, it aborts the computations permanently. If it had produced speculative events based on this computations, it also emit empty events as the final versions for these events, causing the rollback of any speculative computation downstream.

5.4.2 Enabling operators to output multiple events

While presenting our approaches we assumed operators would produce either zero or one output. This assumption simplified the explanation of our approaches and does not limit our system. Consider for example that a transaction in an operator may issue several output events for an input event. Because the system is statically configured, all downstream nodes will receive the same set of events. Therefore, the only effect of combining all output events in a single one is that events generated earlier in the transaction had to wait until the end of the transaction before been sent.

Nevertheless, the approaches discussed earlier can also be extended for transactions that generate multiple events. With passive replication, the timestamp of transaction that will process an input event is created based on the *oid* and *txid* parameters. With multiple events, the creation of the timestamp also needs to consider the *id* field to distinguish between updates and new events. Later, if transaction *txid* in the upstream node aborts and, in a second execution, generates less events. Some of the transactions in the current operator would be updated, but others not. Then, during commit all transactions that were updated would be enabled to commit and all transactions that were not updated are permanently aborted (as if they had been updated with an empty event). However, if during the second execution, the transaction had issued more events. The exceeding events would cause the creation of new transactions.

In active replication with time-based atomic broadcast, the extra events could be handled by creating secondary attributes that would work as subdivisions of time. Similarly to the passive replication, events that were not updated by the latest execution of a transactions are implicitly updated with an empty event and extra events generate extra transactions. Finally, the active replication with consensus-based broadcast works exactly as the passive replication. It also needs to consider the *id* in addition to the *oid* and *txid* fields when creating, updating, and committing transactions.

5.5 Evaluation

In this section, we first evaluate the benefits of speculation in passive replication. After that, we consider active replication and, then, our extension that enables the use of strengthened replicas to check computations.

Unless otherwise noted, the experiments in this section were executed in machines equipped with two Intel Xeon processors (8 cores total) and 4 GBytes of RAM. In some cases, we used a Sun T1000 machine with an UltraSPARC T1 processor (8 cores with 4 hardware threads each), 16 GBytes of RAM, and a machine with four AMD Opteron processors (16 cores total), 16 GBytes of RAM.

5.5.1 Passive replication

To evaluate the cost of the logging operations, we have first experimented with a simple system composed of just two operators. For each event processed, the operators logs a 64-bit value, which could represent an (*oid*, *txid*) tuple or simply an arbitrary nondeterministic decision. In Figure 5.3, we show the impact of the logging in a nonspeculative system for different configurations. We use three logging configurations in which the system is equipped with one, two, and three local hard drives (referred to as *1 disk*, *2 disks*, and *3 disks*, respectively). In addition, we have evaluated two logging configurations where storage is simulated, in these two cases we assume that a set of event logs can be stored in either 10 *ms* or 5 *ms* (referred to as *Sim 10* and *Sim 5* in the figure).

The first simulated value, 10 *ms*, serves the purpose of simulating multiple disks in a machine that has enough cores for executing multiple operators but has a single disk, which would represent a bottleneck in the experiments. The second simulated value, 5 *ms* is used to simulate the cost of logging to the memory of a node in a remote location (e.g., in a different data center).

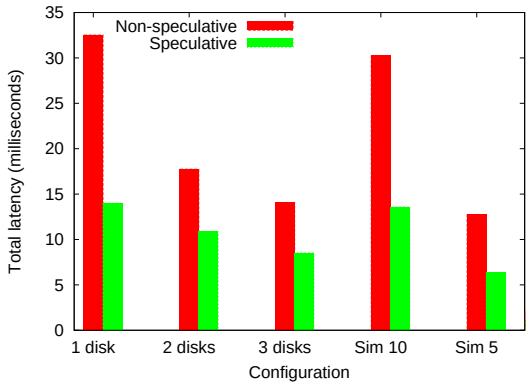


Figure 5.3: End-to-end latency with two operators for different logging configurations.

To evaluate the impact of speculation in a more complex application, we analyze the impact of the checkpoint in a graph with 2 to 7 operators that need to log their decisions. This experiment was executed in a single machine, the Sun T1000, with each operator running as a separate process connected to its neighbors through a TCP connection. The machine has enough hardware threads for not having contention on the processor and, to avoid bottlenecks on disk accesses, we used simulated disk accesses with 10 *ms* access time.

As shown in Figure 5.4, the latency for the speculative executions is nearly constant independently of the number of operators. In a real distributed scenario, for each remote TCP connection there would be a latency increase from a few hundreds of microseconds in LAN settings to up to several milliseconds in WAN settings. Nevertheless, these added latencies would still be shorter then the delays produced by the logging and, thus, the graph would have a similar shape.

5.5.2 Active replication

In this section we evaluate the three main advantages of speculation, namely: (*i*) optimistic delivery and preprocessing; (*ii*) multithreaded operators; (*iii*) early forwarding of speculative events.

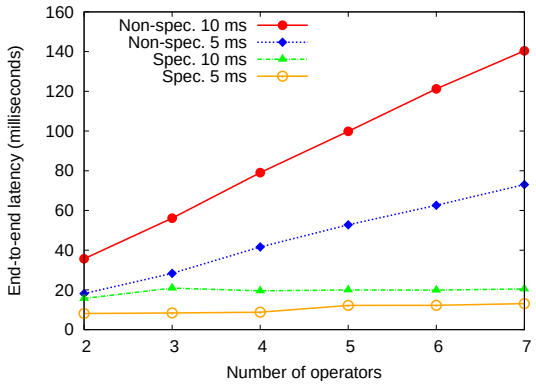


Figure 5.4: End-to-end latency with different number of operators and logging times.

Optimistic delivery in atomic broadcast

We evaluate next our assumption that messages sent by two different sources will often arrive at the replicas of operator in the same order. In this experiment, we consider two sources and two replicas for the stateful operator, all in the same local area network. Each source emits events at increasing rates from 500 to 15,000 events/second. Furthermore, we consider a cheap and non-parallelizable operator, which is the worst case for speculation. In this case, when a misspeculation occurs and an event must be aborted and reprocessed, all the events that were speculatively processed after it will also need to be aborted and reprocessed.

We then measure the rate of aborts, rollbacks, and successful commits. The commits reflect the number of finished events. The rollbacks are the number of times that a final delivery did not confirmed a previous optimistic delivery. Finally, when there is a rollback, if events had been already processed according to an optimistic delivery, several computations may have to be aborted. These measurements are shown, respectively, in the upper, middle and lower part of Figure 5.5.

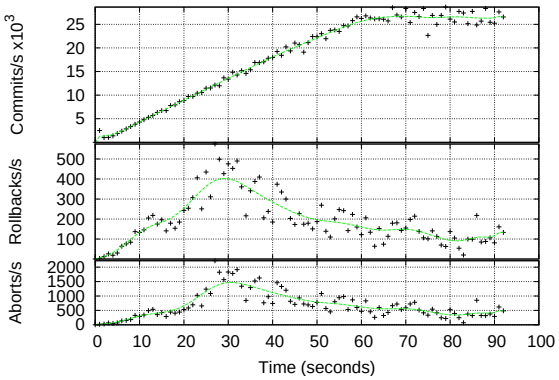


Figure 5.5: Commit, rollback, and abort rates for different workloads (the lines are an approximations for the data).

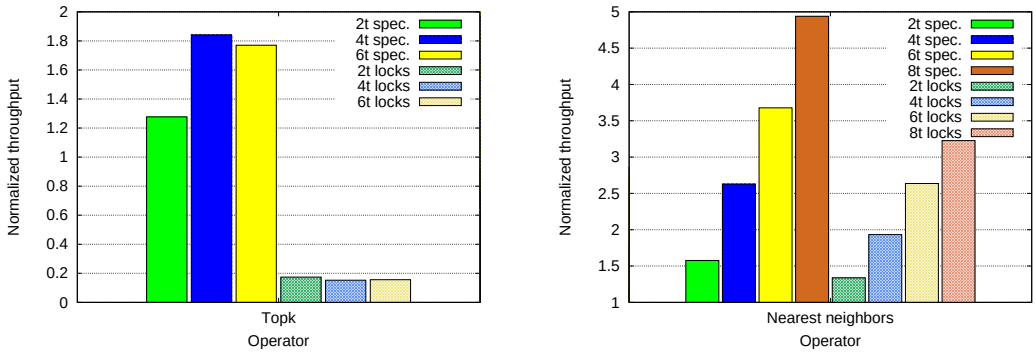
As shown in the figure, the number of aborts and rollbacks is higher in the middle area

around time 30. This is expected because, on the one hand, with lower input rates the variation in communication delays is lower than the interarrival times and events arrive naturally in order. On the other hand, with high loads the events accumulate in the input queues and allow deterministic ordering.

Multithreading-enabled active replication

As discussed earlier, in active replication, multithreaded operators must be forced to execute operations in a way that is repeatable in all replicas. We followed the approach proposed by Basile et al. [BKI06] and modified our operators from the previous chapter. In the next experiment, these operators are changed to acquire locks deterministically: (*i*) when a thread tries to acquire a lock, it blocks until all threads also try to acquire a lock; (*ii*) when all threads are blocked, locks are assigned deterministically. As long as there are no other sources of nondeterminism, this algorithm is repeatable in all replicas. Our implementation of these deterministic locks uses a `pthread_barrier` to block threads until all of them reach the lock acquisition function.

The speedups achieved for the top-k and nearest neighbors operators with speculation and deterministic locks are shown in Figure 5.6. For the top-k, the overhead of synchronizing the threads is prohibitive: the parallel versions are slower than the sequential. This is the case because threads have to synchronize too often and the computations between synchronization points is too small. In addition, because the threads execute in lockstep, all threads will always be on the same line of the top-k matrix and are more likely to contend for the same locks.



(a) Top-k (Intel) (b) Nearest Neighbors (Sun)

Figure 5.6: Effectiveness of parallelization when determinism must be enforced (base: sequential nonspeculative execution).

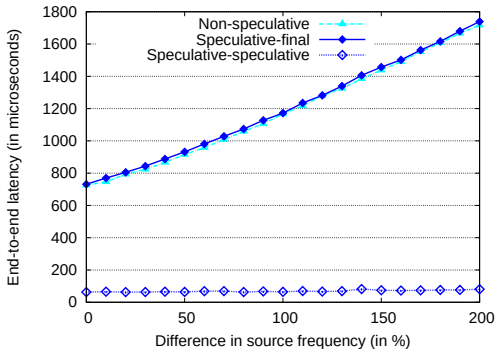
For the nearest neighbors there is much less synchronization and more computation between these synchronization points. Therefore, greater speedups are achieved. Nevertheless, also in this case, our approach performs much better than deterministic locking.

Advantages of speculative events

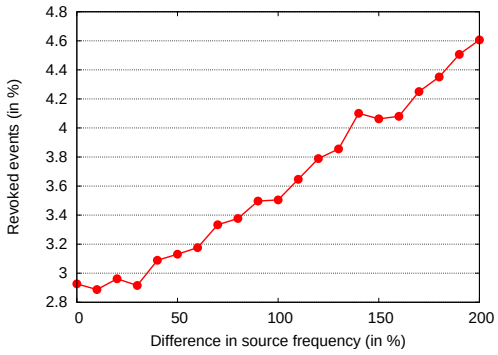
To evaluate the benefit of speculative events, we consider two scenarios. First, we revisit the out-of-order processing from last chapter, where out-of-order processing for short computations

did not pay off. We show that by combining out-of-order processing, time-based ordered delivery, and speculative events, we can see considerable decrease in latencies. Second, we evaluate the benefit of speculative events when failures occur. We show that even in the cases that atomic broadcast does not impose high latencies, speculative events can be still be useful.

Consider now Figure 5.7(a). This figure depicts the average latency of the events from their sources until they reach the input of the operator downstream to an operator that uses time to deterministically order events (i.e., a time-based approach for ordered delivery). Further, in order to factor out synchronization of clocks, we put all operators in the same node and evaluate only the effects of different event generation rates. This experiment considers the top-k operator and one source produces events at a mean rate of 1000 events per second. The other sources produces events in a decreasing rate. For example, when the difference in the generation frequency between the replicas is 0% (extreme left of the *X* axis), both produce around 1000 events per second each. For a difference of 100%, the slower source produces 500 and the faster 1000 events per second.



(a) Latency with time-based event ordering



(b) Percentage of revoked speculative events

Figure 5.7: Benefit of speculative events when using time to order events deterministically.

Three curves are shown in Figure 5.7(a): (*i*) the processing latency when using a nonspeculative operator (referred to as *Non-speculative*); (*ii*) the processing latency for final events when using a speculative operator (referred to as *Speculative-final*); and (*iii*) the processing latency for speculative events (referred to as *Speculative-speculative*). As discussed in the previous chapter when a computation is relatively short in comparison to the commit phase of speculation, speculative out-of-order processing offers no benefit. However, now, speculative computations are allowed to output speculative results while still waiting for the final ordering. Finally, the number of revoked speculative events is proportional to the abort rate in the speculative operator. The consequence is that the downstream operator is able to see events much earlier and, at the same time, there is only a slight chance that computations based on these events will be aborted, as shown in Figure 5.7(b).

Active replication is usually chosen because it permits "virtually" immediate recovery from failures. This is possible because the replacement operator is already actively producing results. Nevertheless, failures impact the protocol indirectly, through the atomic broadcast. For example, consider two replicas running a consensus-based atomic broadcast. If one replica fails, the surviving replica is only allowed to unilaterally decide on a message ordering if it is sure that

the other replica has failed. However, in practical distributed systems, the communication delays in a network may vary and peak values are much larger than average values. For this reason, time-outs for failure detection must be conservative (e.g., to avoid costly consequences of frequent erroneous suspicions).

Therefore, in the second scenario we consider the case of failures. We consider an application like our running example, where only the `Processor1` operator receives messages from multiple sources. This operator is the only operator that needs to execute protocols for ordered delivery. Moreover, we place the operators in different nodes and use the consensus-based agreement for ordered delivery in the replicated operator.

In such a situation, the advantage of the optimistic delivery in a single component is small in comparison to the end-to-end delay. Nevertheless, if failures occur, speculation allows that replicas keep producing events. The results of this experiment are shown in Figure 5.8. When a node fails, the atomic broadcast cannot deliver final events until the failure is reliably detected. Nevertheless, the optimistic broadcast is still able to deliver events optimistically and these events can be processed speculatively.

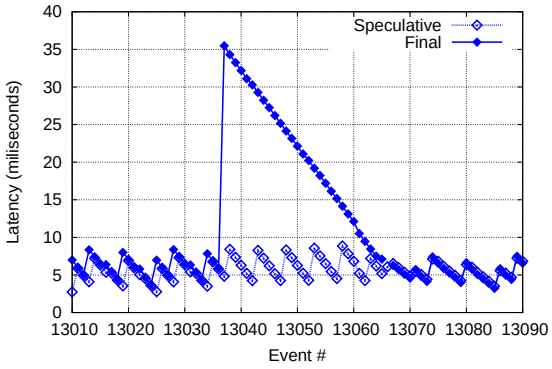


Figure 5.8: Effects of a failure in the generation of final events.

Note that for some applications, early results can be very useful event if these results are not guaranteed. Similarly, if a the application above included an operator that was longer than the failure detection delay, this operator would have been able to use the speculative events and the failure would be completely masked.

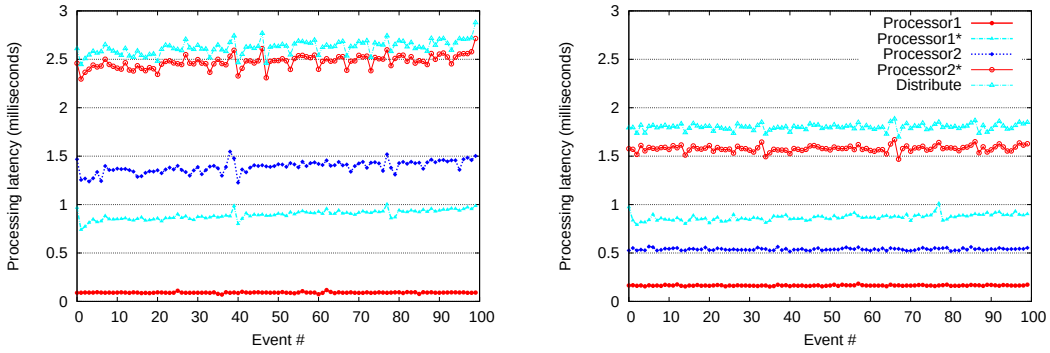
Active replication with software faults

In our last set of experiments, we evaluate a scenario with our extension that enables active replication to tolerate failures other than crash. In this case, the `Processor1` operator implements a simple neural network that classifies events according to a preconfigured model. In addition, we consider a stateful `Processor2` operator that executes a top-k operation. Finally, the `Distributor` operator serves only as a gateway that filters out speculative events before they reach the consumers.

In this example, both `Processor1` and `Processor2` operators will be replicated and all operators are placed in the same machine, which is equipped with four AMD Opteron processors

and 16 cores in total. We use a checker that is able to detect out-of-bound accesses [RL04, BWS+10].

The results of this experiment are shown in Figure 5.9. Because the operators are in the same machine, we can measure the latency of an event from generation up to when operators generate outputs triggered by this event. For example, the curve named *Processor1**, represents the latency of the events from their generation until they reach the output of the checked replica for the `Processor1` operator.



(a) Without speculation (b) With speculation

Figure 5.9: Performance comparison between a nonspeculative and a speculative execution when using active replication in combination with checkers.

The benefit of speculation can be seen in Figure 5.9(b), an event reaches the output of *Processor2*, the non-checked replica of the `Processor2` operator, much earlier than in Figure 5.9(a). This is possible because events can be first processed by the fast version of the `Processor1` operator and then by the fast version of the `Processor2` operator.

In Figure 5.9(b), the time between curves *Processor2* and *Distribute* consists mostly in waiting for the checker versions of the operators to finish. Note that the second checker starts executing based on the speculative result of the first non-checker replica. This is possible because either the first checker will later confirm this result and the second checker can commit, or the first checker detects a failure and the second checker aborts the event permanently.

5.6 Summary

In this chapter we discussed the usage of speculation between operators. We extended the speculative support presented in the previous chapter with ability to send and process speculative events. We used this functionality to hide the costs of making operators fault tolerant.

Our evaluation shows that when using passive replication and rollback recovery, our approach enables that many operators are able to save their nondeterministic decisions or checkpoints almost in parallel. In contrast, with conventional passive replication each node that saves some information adds the complete duration of the store operation to the processing delay.

With active replication, our system is able to overlap the ordering phase of the required atomic broadcast protocol with useful computations. Our experiments showed that if time is used to order

events in a replica, out-of-order processing and speculative events can be very useful, even with short operations, like top-k. In addition, speculative events also allow replicas to keep working in failure scenarios when the system would be otherwise blocked until the failure is detected.

Another enhancement to active replication was the ability to use multithreaded operators. In the previous chapter, because of the overhead, speculative parallelization achieved lower performances than lock-based approaches. Nevertheless, once we consider active replication and the need for deterministic scheduling, the relative performance of our approach increases. Our evaluation results showed that our approach performs considerably better than a previously proposed deterministic locking scheme. Moreover, when multiple replicas run manually-parallelized code, even benign bugs that provide otherwise acceptable results could create inconsistency among replicas.

Finally, we presented an extension that enables active replication to tolerate failures other than crash failures. The basic idea is to have two types of replicas. First, replicas that are fast and can provide results that are speculatively used in downstream nodes. Second, replicas that are robust and perform checked executions that detect when inputs trigger software bugs. Speculation is then used to perform recovery when only minor problems occur and to hide part of the added latency by enabling operators to start checks use speculative results.

Chapter 6

Related work

In this chapter we review related work and highlight the main similarities and differences to our work.

6.1 Event stream processing engines

Event processing systems (ESP) are considered a spin-off from the active database research [Day94], but they were also greatly influenced by discrete event simulation [CM79] and signal processing (e.g., projects like Ptolemy [UC 10]). A list of early database systems that incorporated the concept of triggers, which enabled actions to be executed when certain events occurred, has been compiled by Chakravarthy and Adaikkalavan [CA08].

There are many recent projects that focus on processing continuous streams of events, some examples of research projects are: Borealis [AAB+05], CEDR [BGAH07], Gigascope [CJSS03], GSDM [IR05a], Streamflex [SPGV07], SASE [WDR06], STREAM [ABB+04], StreamIt [AGK+05], System S [GAW+08], TART [SDFW09]. There are also many commercial products that address ESP, some examples are: Esper [Esp10] (open source), Progress Apama [Pro10], StreamBase [Str10], IBM Infosphere [GAW+08] (offshoot of System S).

These projects can be classified by the types of languages used to express applications and by their goals. Regarding the languages, there are three basic approaches: extending SQL, extending a traditional programming language (like C or Java), and using a (custom or not) script language. SQL-based approaches can be found in Esper, Gigascope, GSDM, SASE, STREAM, and StreamBase. Extensions to Java can be found in StreamFlex and TART. And, finally, script languages (or custom languages with their own compilers) are used in Borealis, CEDR, Progress Apama, StreamIt, and System S.

Regarding the main goal, systems can be focused on processing the stream or on detecting event patterns. On the one hand, processing the stream, or *stream mining*, aim at executing data mining operators that are suitable for streams (e.g., StreamIt, GSDM, Streamflex, TART). In order for an operator to be suitable for streams it must be consider that not all events can be kept for later consulting (preferably each event is looked at only once) and the needed memory and processing time are bounded [Agg07]. One example is the top-k used in our experiments [CCFC04]. It uses an approximation of the event stream to compute the (approximate) frequency of events. Other

examples of synopsis can be found in the work from Cormode and Muthukrishnan [CM05] and in the survey of Aggarwal and Yu [AY07]. More general surveys on the issues in stream processing had been done by Babock et al. [BBD+02] and Golab and Tamer [GO03]. In addition, some works assume a data model that is closer to signal processing, the data is, for example, single decimal numbers (e.g., StreamIt and the work from Muthukrishnan [Mut05]). Nevertheless, most implemented systems assume a model where events are more sophisticated entities with a rich set of user-defined (e.g., customer-id) and system-defined (e.g., source, occurrence time) attributes (e.g., Borealis, Esper, Streamflex, TART).

On the other hand, detecting known patterns of events has a large commercial appeal and has been commonly referred to as *Business Intelligence* (BI) [Luc01]. This is the focus of the CEDR, SASE, STREAM. Detecting patterns is normally precisely computed and it bounds computation requirements by considering only a window of events (e.g., either the last n events or the events in the last t time units).

The prototype described in this dissertation uses C programming language augmented with library calls and we address the general case where no specialization on the operators are assumed. The exact operations are then implemented by the user with the help of library functions to handle event-related (e.g., generation, discard) and fault-tolerance related tasks (e.g., logging, checkpointing). The key distinguising features of our prototype are the support for parallelization of stateful operators, speculative events, and out-of-order events. Parallelization of stateful operators and support for speculative events are discussed in the next sections. Tolerating out-of-order events has also been considered by Barga et. al [BGAH07] (for CEDR), by Li et al. [LLD+07] (for SASE), by Hwang et al. [HCZ08] (for Borealis), and by Li et. al. [LTS+08] (for Gigascope).

In CEDR, the authors address the case of pattern matching and assume that output events can be emitted by an operator and later corrected (see discussion below on speculation in event processing systems). In addition, they consider the tradeoff between blocking of an operator, memory usage, and consistency. In order to tolerate out-of-order events and still keep high consistency, either the operator blocks or it outputs a result, but keeps enough state that enables it to emit a correction later. When an operator neither blocks nor keeps enough state, results become inconsistent (i.e., if an event arrives later it is not possible to detect that an older pattern match should have considered this event). The users have then to configure the system considering the consistency level, state size, and the amount of blocking in a way that suits their applications needs.

Li et al. [LLD+07] also consider pattern matching. They propose an extension for SASE that allows the usage of a slack parameter that defines a maximum waiting time for the out-of-order events to arrive. Finally, Hwang et al. [HCZ08] and Li et al. [LTS+08] consider general operators and use punctuations to determine until when an order-sensitive operator should wait. A punctuation is an special event that marks the end of a portion of the stream (i.e., a punctionation p_t states that no events with timestamp less than t will arrive in the future). After receiving a punctuation, a windows of events is defined and the events in that window are sorted. The operator can then process the ordered events.

In comparison to the systems above, we assume that the lack of ordering was caused by, for example, a temporary silence in one of two input channels or by the optimistic delivery of the

atomic broadcast protocol. Therefore, we assume the final ordering will soon be known. Our main goal is then to use speculation to preprocess events before the final ordering is known. Our system could be used to reduce processing latency in work of Hwang et al. [HCZ08] and Gigascope [LTS+08] as both need to wait for the punctuation before processing. Our system could also support the same out-of-order processing as done for SASE [LLD+07] or CEDR [BGAH07], but because it is not optimized for pattern matching, keeping the transactions open will require more memory. Finally, we do not address tradeoffs between consistency, blocking time, and memory costs.

6.2 Parallelization and optimistic computing

Parallelizing programs by hand is known to be a difficult task. On the one hand, coarse-grained locking does not provide much parallelism. On the other hand, fine-grained locking and lock-free implementationsis are difficult and error prone. In addition, debugging, testing, and composing parallel code is difficult [GN08, ABD+09, JAAS09].

Automated parallelization can be done statically or during runtime. Static approaches are normally based on compilers. Parallelizing compilers have been studied for a long time, but have limited applicability, specially for general purpose applications [SCZM05]. Basically, parallelizing compilers have problems to deal with complex memory access patterns and are pessimistic: in order to provide parallelization they must statically prove that threads are independent. Steffan et al. [SCZM05] proposes an approach that combines thread-level speculation (TLS) and compilers. This approach enable compilers to generate parallel code when there is a *high enough chance* that threads are parallel and uses a TLS-based approach as a safety net for when problems occur. Nevertheless, modifying a compiler to include a new optimization is a very difficult process. Because compilers are too complex and such a change affects many internal data structures, it is estimated that it takes a decade for a new compiler optimization to reach a production compiler [ABD+09]. Therefore, runtime approaches for parallelism are needed both to address problems not suitable to parallelizing compilers and to allow rapid innovation.

In the next sections, we discuss speculation and optimistic parallelization approaches.

6.2.1 Speculation

Speculation in computer systems have being used in many different contexts. For example, hardware speculation for branch prediction [Smi81] is crucial in modern processors. It speculatively executes instructions after a branch, if it speculation is later proven wrong it discards the results. Out-of-order execution has also been around for long time, since the IBM 360/91 [Tom67].

Closest to our work are the seminal works from Jefferson [Jef85] and Strom and Yemini [SY85]. Jefferson's *virtual time* considers the general case of speculation in a distributed system. In this work, the author considers the case where nodes process messages in the order of their timestamps. Moreover, when a node receives a message, instead of waiting until it is sure no other messages with lower timestamps will arrive, it simply continues computing. The node is then speculating that no message with a timestamp in the past will arrive. However, if a node later receives such a message, it rollbacks to a state before that message, process the new message,

and then reprocesses the other messages it has received. Strom and Yemini apply an approach similar to virtual time to fault tolerance (see discussion about fault tolerance below).

A problem with virtual time is that a rollback in a node will cause messages from the past to be sent again and potentially trigger further rollbacks in other nodes. Thus, in a general distributed system it is crucial to coordinate globally which checkpoints must be kept to avoid cascading rollbacks. In our case, we use a software transactional memory (STM) to efficiently keep fine-grained checkpoints, but in contrast to virtual time we do not face the problem of cascading rollbacks as ESP applications are acyclic directed graphs.

In this work, we also consider the usage of operators that are strenghtened against common software bugs. We then use the original versions to provide faster, but less trustwhorthy, results that are speculatively used in downstream nodes (see the discussion about fault tolerance and active replication below). This kind of speculation is also the goal of Fast Track [KBDZ09] and the works from Nightingale et al. [NPCF08] and Süßkraut et al. [SWK+10].

6.2.2 Optimistic parallelization

Steffan and Mowry proposed Thread Level Data Speculation (TLDS) [SM98, SCZM05] as a way to benefit from multiprocessor computers when the programs are not designed for explicit parallelism. They suggested that compiler support could enable activities in a sequential program to be executed simultaneously and then committed from the less speculative activity to the most speculative one. If some dependencies are detected the speculative activity is terminated and restarted. More recently, Oancea et al. [OMH09] proposed SpLIP, a software thread-level speculation algorithm based on recent STMs. Their implementation aims to be more scalable by avoiding serial-commits and to have a lower overhead by trading in false conflicts for less locking and memory ordering instructions.

In IPOT (Implict Parallelism with Ordered Transactions) [vPCC07], Praun et al. propose a programming model offering constructs that allow the specification of sections of code that can be parallelized optimistically. IPOT's implementation is also inspired in transactional memories and their support for isolation and atomicity. As our work, they also used ordered commits from the less speculative to the more speculative tasks, detecting conflicts and reexecuting tasks when needed.

Transactional memory was popularized by Herlihy and Moss [HM93] and the idea of an all-software version was proposed by Shavit and Touitou [ST95]. Our speculative support is based on TinySTM [FFM+07, FFR08]. Other STM implementations are McRT-STM [SATH+06], TL2 [DSS06]. Herlihy et al. proposed DSTM2 [HLM06], a framework for implementing STMs in Java. Saha et al. [SATH+06] also provide a quantitative analysis of various STM design tradeoffs (e.g., write through versus write back, read version versus read locking).

Performance of STMs can be greatly improved by hardware support. Because of the inherent difficulty to exploit multicore processors, manufacturers are expected to provide increasing support for the development of tools for concurrent programming. One such ongoing project is AMD's Advanced Synchronization Facility (ASF) [DH08]. Recent results have shown that the performance of TinySTM can be greatly improved with ASF [CCD+10].

The main difference of our speculation support in comparison to conventional STMs is the usage of the timestamp of the event as the transaction timestamp and to enforce this ordering

during commit. Conventional time-based STMs, like TinySTM, typically use a timestamp that is computed when the transaction tries to commit. Because of this difference, we have to provide a set of additional capabilities, like enabling transactions to be paused, put aside, and later continued. In addition, the enforced ordering of transactions also increases the importance of optimism control. For this end, we have proposed an abstraction named conflict predictor.

6.2.3 Parallelization in event processing

The parallelization of costly operations in an ESP system is addressed by Flux [SHCF03] using a partition-compute-combine approach. Ivanova and Risch [IR05b] also rely on this approach. The main limitation of these approaches is that it only works with operators that can be processed according to this divide and conquer pattern. Further, even in such cases, the merging phase can be complex and limit the total speedup according Amgdahl's law.

In some systems (e.g., Streamflex [SPGV07]) operators are assumed to be stateless and, thus, can be parallelized by simple replication. Other systems (e.g., Borealis [TcZ07]) turn to load shedding, instead of parallelization, to address bottleneck operators

Finally, based on the same motivations as ours, Sturzrehm et al. [SFF09] proposed modifications to DSTM2 to enable optimistic parallelization for ESP operators written in Java.

6.2.4 Speculation in event processing

The idea of using events that can later be corrected has been studied in CEDR [BGAH07] and Borealis DPC [BBMS08]. CEDR uses a temporal model that deals with out-of-order events by allowing results to be output and later be retracted and revised in case a relevant, late event arrives. Nevertheless, in CEDR any event can be later corrected and the authors do not consider how downstream nodes will support the correction events. In our system a pattern matching operator could execute computations speculatively and emit speculative output events. However, we require an explicit deadline for the waiting of the late events. This deadline is what allows the commit of the pending transactions.

In Borealis DPC, the authors address consistency and responsiveness tradeoffs. For example, assume a network failure occurs and one of two inputs of an operator becomes disconnected. After some time (which defines the maximum tolerated delay, specified by the user), the operator is allowed to proceed (after checkpointing its state) and produce *tentative events* (similar to our speculative events). When a downstream node sees the first tentative event it executes a checkpoint and enters in a tentative mode. In the tentative mode, operators produce only tentative output events. Later, when the network recovers and the missing inputs arrive, the nodes restore the checkpoints and start issuing final events for the previous tentative events.

In contrast to Borealis, our speculations bets that computations will not be revoked. For example, we speculate that an asynchronous checkpoint in a node will complete (i.e., failures are rare) and the speculative events will soon become final. In Borealis, the system forks into a state that is known to be wrong, but does so in the hope that the early incorrect results can still be useful. We then focus on fine-grained rollbacks instead of checkpoints of the whole graph. In addition, although Borealis uses active replication of the operators, it does not consider optimistic deliveries or addresses multithreaded operators. Finally, we provide some support for executing

external actions, like writing to persistent storage. Borealis consider such operations to not be fundamental to ESP.

6.3 Fault tolerance

In this work we consider passive and active replication. Both approaches have being extensively studied in the context of general distributed systems. Nevertheless, the problem of highly available distributed ESP systems is a topic that only recently has been addressed. In the next sections we first discuss the most related work in the general case of distributed systems and then discuss the work specific to event processing.

6.3.1 Passive replication and rollback recovery

The passive replication approach we consider is a primary-backup [BMST93] approach, as only one node does the computation and this node checkpoints its state to stable storage. When we assume the stable storage is local to the node, the approach is usually referred to as rollback recovery. Elnozahy et al. [EAWJ02] present a survey on rollback recovery. The work most closely related to ours is optimistic logging [SY85], where the authors also consider that nodes forward results for which logs and checkpoints are not yet stable on disk. Nevertheless, for the case of general distributed systems, optimistic logging is less attractive for three main reasons that greatly increase its complexity.

First, garbage collection of older checkpoints (and truncating the log) requires coordination between nodes. Garbage collection must ensure that nodes keeps enough checkpoints to be able to recover to a consistent state. For example, consider that a node receives a message from a remote node. If the decisions used to generate that message are still not stable on the remote node's disk (i.e., the equivalent of our speculative events), the current node must keep also an older checkpoint that has the state previous to the reception of that message. In our case, a node can always keep only the lastest checkpoint of the operator state. State modifications that were triggered by speculative messages are not visible outside the STM.

Second, for similar reasons as for garbage collection, if a node fails, during recovery it must negotiate which checkpoint it will restore. The node must make sure to restore a checkpoint that keeps the system consistent (e.g., no node received a message that was never sent). Then, when it restores a checkpoint, other nodes also may need to rollback and restore checkpoints that preserve consistency. In our case, when a node recovers, it restores the latest checkpoint. Then, because in ESP system events only flow from the upstream nodes to downstream nodes, as long as upstream nodes keep a buffer of messages recently sent, upstream nodes do not have to restore older checkpoints. After restoring the checkpoint, the recovering node starts reprocessing the messages and producing new results. Once these new results reach the nodes downstream of the recovering node, the STM checks if the changes between the new events and the older ones, produced before the failure, are relevant (i.e., the changes indeed affect previous computation). Only if the changes are relevant the pending open transactions need to be aborted.

The third and final reason is that when using optimistic logging in a general distributed system, nodes have to coordinate before outputting results. This is necessary in order to guarantee that

no results that leave the system will be later revoked because of a failure. Again, coordination requires multiple message rounds and is, therefore, an expensive procedure. In our case, also because of the limited communication pattern of ESP systems (which are acyclic graphs), stability is uniquely determined by the upstream nodes that sent the speculative messages. Thus, no global agreement is necessary.

Also related is the work from Nightingale et al. [NVCF06]. This system is intended to allow applications that rely on synchronous disk writes to proceed with computation before the writes are stable. In this case, the overlapping between writing and processing is enabled by having the operating system to track the dependencies and hold any externalization that is causally dependent on the not yet stable writes.

6.3.2 Active replication

Active replication [CPR+92] and the state-machine replication approach [Sch90] have been studied for general distributed systems for a long time. To address some of its limitations like the inability to handle nondeterminism (multithreading being one type of nondeterminism), variations like semi-passive [DSS98] and semi-active replication [DTT99, CPR+92] were proposed. In semi-active replication, one replica (the leader) takes the nondeterministic decisions and forwards these to the other replicas (the followers). Semi-active replication can be used to solve the problem of multithreaded executions, but it requires collecting and communicating scheduling decisions and may cause the followers to lag behind the leader. Similarly, Basile et at. [BKI06] evaluate a mechanism to transmit all lock-acquisition decisions from the leader node to the follower nodes. In semi-passive replication, all the relevant computations, together with any potentially nondeterministic scheduling decisions, are done by the primary. The primary then forwards state changes to the replicas.

Another approach is to make multithreaded replicas deterministic. Jiménez-Peris et al. [JPPnMA00] present a deterministic scheduler that guarantees that multithreaded replicas will execute deterministically. Nevertheless, only one thread can be active at a time. Thus, no parallelism is indeed exploited. Basile et al. [BKI06] propose an approach that is able to preserve some of the parallelism by dividing the execution in rounds so that the order of the lock acquisitions is decided deterministically at the beginning of each round. In this case, threads that try to acquire a lock will block until all other threads try to acquire any lock. Then, when all threads are blocked waiting for locks a scheduler grants them in a deterministic order. Although this approach enables more parallelism to be exploited, there are still considerable limitations, specially if the processing of the different threads is unbalanced. In our case, because transactions appear to be executed atomically and are committed in the delivery order of the events they process, scheduling is deterministic. In addition, for workloads with low conflict rates the overhead of speculation is small, enabling good exploitation of parallelism. Note, however, that previous approaches do not benefit from these low conflict rates: even when the probabilily of conflicts is small, as shown for our example operators, locks still need to be acquired for safety.

From the data point of view, optimistic replication is a very well studied concept (see the survey from Saito and Shapuri [SS05]). Pessimistic approaches control the access to the data in a conservative way, for example, by locking. In contrast, optimistic approaches enable updates to proceed speculatively and eventually conciliate them in a total order. Using optimistic deliveries

from an atomic broadcast protocol in order to advance computations was proposed by Pedone and Schiper [PS98]. This idea was then extended by Kemme et al. [KPA+03].

Kemme et al. consider a system running in a local area network and exploit the fact that in such a network messages often arrive in the same order in all nodes. In this situation, the order of the messages as given by the optimistic delivery of an optimistic atomic broadcast (e.g., [PS03]) would often be the same as the final delivery after the distributed aggreement that does the ordering. They then use this fact to speculative start database transactions based on the optimistic delivery. This work differs from ours in several ways. First, their protocol requires that transactions atomically acquire all needed locks before starting. Although this is feasible for stored procedures in a database, it is not the common case for event processing operators, where the locks will likely depend on intermediate results of the computation. In our case, the locks are indirectly acquired during the execution (the STM handles the locking).

Moreover, in the system from Kemme et al., if a transaction is optimistically ordered after another conflicting transaction, the second transaction can only be tried after the first commits (i.e., after the final delivery). In our system, we allow transactions to speculatively read from the write sets of conflicting transactions that were ordered before as long as these already finished processing. Finally, because the cost of databases transactions, which require disk accesses, can be much higher than the aggrement phase of the atomic broadcast in a local network, relative gains can be small. In contrast, in our case, the agreement phase and the durations of (memory) transactions are in the same order of magnitude. Furthermore, our approach allows speculation to speedup computations not only in the current operator, but also in downstream operators.

Sousa et al. [SPMO02] discusses some issues that may cause optimistic delivery to not reflect optimistic delivery, specially in wide area networks. One of the major problems, which also afects systems in local area networks, is that messages broadcasted by a node tend to be (optimistically) locally delivered much earlier than messages from remote nodes. This problem does not occur in ESP, where the operator itself is not among the recipients of its events (the operator graph is acyclic). This problem can also be solved using the bias algorithm [AS00], which was mentioned in Chapter 5.2.3 as an optimization to the time-based atomic broadcast.

Lastly, traditional active replication is based on the assumption that nodes fail independently and, thus, replicating nodes increases reliability. As software bugs become the main source of failures, this assumption may not hold. To address these scenarios, we discussed the usage of automatically generated versions of the original operators to provide diversity among the replicas. Our work is based on the idea of n-version programming [CA95], where multiple independent copies of the same software component are developed and executed as replicas. Further, the usage of automatically generated versions of a component is also considered by Schneider and Zhou [SZ05] and Cox et al. [CEF+06]. Cox et al. propose a framework to use variants and monitor their executions to detect deviation before the system can be compromised. In the same work, the authors describe how to generate variants that can be used with the framework. Schneider and Zhou describe how active replication can tolerate Byzantine faults. They also consider the usage of variants of components to reduce susceptibility to specific software faults.

Note that if n-versioning is used to provide diverse replicas, replicas may use very different algorithms. If operators are then multithread, communicating scheduling decisison between replicas does not solve the nondeterminism caused by multithreading. This problem does not

affect our approach.

6.3.3 Fault tolerance in event stream processing systems

There are examples of both active and passive replication in event processing systems. The works from Shah et al. [SHB04], Hwang et al. [HBR+05, HCZ08], and Balazinska et al. [BBMS08] are examples of active replication approaches. Hwang et al. [HBR+05] and Strom et al. [SDFW09] consider passive replication.

Flux, proposed by Shah et al. [SHB04], is a pseudo-operator to be inserted between regular operators in a event processing dataflow. It handles coordination of process pairs, for example, managing recovery and communication for the original operators. Flux requires the original operators to provide interface functions for retrieving and restoring the internal state and guarantees that no events will be lost or duplicated due to failures and recoveries. Nevertheless, their operator model is restricted to deterministic single-threaded operators and the authors consider that both replicas receive events from a single source.

Hwang et al. [HBR+05] classify operator graphs in four classes (repeatable, convergent-capable, deterministic, and arbitrary) and present four types of recovery protocols (amnesia, passive standby, upstream backup, and active standby). They advocate that many applications do not require precise recovery and they classify how precise each of the recovery protocols is for the different operator graph types. Finally, they discuss how three of the protocols could be adapted to provide precise recovery for arbitrary operators (which is the only class that includes nondeterminism). In the passive standby approach, based on checkpoints, the operator can only forward checkpointed tuples downstream. The active standby approach, based (like Flux) on the process-pair model, requires that primaries send the nondeterministic decisions to the secondaries and then wait for the acknowledgment before sending events downstream.

In Borealis R [HCZ08], Hwang et al. acknowledge the problem of the high performance costs of implementing active replication and propose modifications to the systems and operators to allow them to execute independently. In essence, they add well synchronized clocks to the sources, punctuation to the streams, and sorting of events (between punctuations) on the operators. In fact, this time-based approach can be seen as an implementation of atomic broadcast and, as expected, latency costs will be added as events cannot be processed as soon as they are received. The operator must wait for the punctuation and then sort the events so that the same ordering of events can be guaranteed among the replicas. Further, this approach does not address the use of multithreaded operators. In our work, we also consider the usage of time to deterministically merge streams of events, as proposed by Aguilera and Strom [AS00]. In comparison to Borealis R, we extend its approach in order to minimize latency costs. We allow operators to speculate while waiting for the final ordering and we provide support for multithreaded operators.

In Borealis DPC [BBMS08], Balazinska et al. consider active replication, but do not consider multithreaded operators or optimistic delivery. Nevertheless, they address tradeoffs between availability and consistency (see discussion above on speculation in event processing), which we do not address.

Finally, in TART [SDFW09], Strom et al. consider passive replication. Their system consider that operators do periodic checkpoints. Nevertheless, to avoid logging, the authors make operators deterministic. In the case of operators with multiple inputs, they use timestamps in the events

to force a deterministic ordering. Moreover, they propose techniques to minimize the waiting time of events in input queues. These techniques try to deterministically estimate the processing time of an event through the stream application so that an event arrives approximately when it is expected and, thus, does not need to wait long on the queue. On the one hand, their techniques resemble work on deterministic merge by Aguilera and Strom [AS00] and could in some cases improve our time-based atomic broadcast. On the other hand, our deterministic parallelization and our speculative preprocessing of out-of-order events could be used to improve their system.

Chapter 7

Conclusions

In this chapter we summarize our achievements and outline interesting directions for future work.

7.1 Summary of contributions

Using stateful operators in event stream processing (ESP) applications poses a series of nontrivial challenges. In this dissertation we address several of them.

Out-of-order processing. Events have often to be processed *in order*, either because the timestamp of the events is relevant to the operation or because repeatability is necessary. Nevertheless, enforcing ordering typically leads to waiting. Our speculation support permits events to be preprocessed out-of-order while still reaching the same results as in an in-order processing engine. Our experiments showed that in a local operator, out-of-order processing can reduce processing latency as long as there is enough computation. We then showed that in a distributed scenario, benefits can be seen even for smaller computations. Compared to related work, our approach provides efficient fine-grained rollbacks and addresses general operators.

Parallelization. Manual parallelization of operators is a complex task, even for experts. Our speculative parallelization enable users to obtain a parallel versions of operators with no additional effort. Moreover, because the software transactional memory (STM) can provide feedback on where the conflicts are, expert programmers can incrementally address bottlenecks and, consequently, improve performance. This process is considerably facilitated because the speculation preserves the sequential semantics. Finally, experiments have shown that for highly parallel operators (e.g., the nearest neighbors) the speedups obtained by speculation are close to the results obtained with fine-grained locking. Parallelization is currently a hot reaseach area and many techniques have been proposed to facilitate parallelization, nevertheless, before this work, optimistic techniques had not yet been applied in ESP.

Cost of passive replication. When operators need to be made fault tolerant, passive replication is a common approach, it only requires that operators save all nondeterministic decisions in stable

storage. For application that need low-latency processing, like ESP, having multiple operators logging their decisions impose high latency costs. Our approach allows these logging operations to be carried in parallel, practically reducing the cost of a sequence of log operations to the cost of a single one. In addition, our techniques for event processing do not suffer from the complications of previous techniques applied in general distributed systems (e.g., need for agreement before recovery, garbage collection, or output).

Cost of active replication. For cases where downtime must be minimized, active replication is more applicable than passive replication. Active replication also imposes considerable performance costs as the delivery of messages and the scheduling of threads in operators must be synchronized. Our approach enables speculative processing to overlap the high latency process of delivering the messages. In addition, the same mechanism exempts replicas from needing to communicate scheduling decisions or to use additional techniques to provide deterministic scheduling. Our approach combines elements from previous works and extend them to exploit features of ESP applications. For example, we allow speculation to speedup computations not only in a single operator, but through the whole graph of operators.

7.2 Challenges and future work

During the development of this work, we identified many interesting challenges that are worthy of further investigation.

STMs greatly facilitate development of concurrent code. It is therefore expected, that some hardware support for transaction memory will be implemented in production processors. Recent results with AMD's Advanced Synchronization Facility (ASF) [CCD+10] show that the overhead of an STM can be considerably reduced. Using such a support would increase the amount of parallelism as well as broaden the spectrum of operators that can benefits from the out-of-order processing techniques proposed in this work. It is also viable to produce easy-to-use software tools that uses feedback from the STM to guide a nonexpert operator programmer through the process of leveraging the parallelism achievable by speculative parallelization. External actions from within a transaction are also an open and difficult to handle problem.

Using events timestamps to make processing deterministic greatly simplifies the development of ESP applications. Deterministic systems can be better understood by simulation and postmortem analysis. In addition, determinism simplifies both passive and active replications. Nevertheless, time-based approaches have their problems too. When different sources of events emit events in different frequencies or the events from different sources suffer different variations in communication delays, operators have to wait. Adaptive techniques that could regulate speculation, silence propagation (when one of the sources do not emit events), and unbalanced communication delays could considerably reduce latency when using time-based determinism.

Finally, using plugable checkers to increase the domain of faults that can be tolerated by active replication is an interesting concept. It is still an open issue to determine how much of the overhead of checking can be masked by speculation, especially if out-of-order processing and multithreading are allowed.

Publications

[BWS+10] Andrey Brito, Stefan Weigert, Martin Süßkraut, Christof Fetzer, and Pascal Felber. Handling crash and software faults efficiently in distributed event stream processing. Accepted for publication in the Third International Conference on Dependability, 2010.

[FBFJ10] Christof Fetzer, Andrey Brito, Robert Fach, and Zbigniew Jerzak. Streammine. In Annika Hinze and Alex Buchmann (editors), Principles and Applications of Distributed Event-based Systems. Information Science Publishing, ISBN 1605666971, Hershey, PA, US, 2010.

[BFF09b] Andrey Brito, Christof Fetzer, and Pascal Felber. Multithreading-Enabled Active Replication for Event Stream Processing Operators. In Proceedings of the 28th International Symposium on Reliable Distributed Systems (SRDS 2009), 2009.

[BFF09a] Andrey Brito, Christof Fetzer, and Pascal Felber. Minimizing Latency in Fault-Tolerant Distributed Stream Processing Systems. In Proceedings of the 29th International Conference on Distributed Computing Systems (ICDCS 2009), 2009.

[Bri08] Andrey Brito. Optimistic parallelization support for event stream processing systems. In Proceedings of the 5th Middleware doctoral symposium (MDS 2008), 2008.

[BFSF08] Andrey Brito, Christof Fetzer, Heiko Sturzrehm, and Pascal Felber. Speculative Out-Of-Order Event Processing with Software Transactional Memory. In Proceedings of the 2nd Conference on Distributed Event-Based Systems (DEBS 2008), 2008.

[BF07] Andrey Brito and Christof Fetzer. Improved event processing performance through parallel event transformation. In Proceedings of the 2nd International Workshop on Event-driven Architecture, Processing and Systems (EDA-PS'07) at the 33rd International Conference on Very Large Data Bases (VLDB 2007), Vienna, Austria, 2007. (Short paper.)

Pseudocode for the consensus protocol

Active replication requires ordered delivery of messages in all nodes. In Chapter 5, we discuss two approach to achieve such ordered delivery. One of these approaches is based on a consensus protocol. In this appendix, we detail how such a consensus protocol can be implemented.

The pseudocode for the interface of the consensus-related functions is shown in Listing 1. It is based on the leader-based consensus protocol for failure detectors that do not wrongly suspects correct processes by Verissimo and Rodrigues [VR01]. In function *consensus_decide*, the leader can already decide in the locked value, but the other nodes must wait for the final acknowledgement from the leader.

```
1  integer my_id, leader_id;
2  integer consensus_id;

4  consensus_estimate(list_t l) {
5     proposal_t p = {my_id, l}; // proposal_t = {id, value}
6     if (my_id == leader_id) return leader_consensus(p);
7     else return nonleader_consensus_estimate(p).;
8  }

10 consensus_decide() {
11    if (my_id == leader_id) return TRUE;
12    else return nonleader_consensus_decide();
13 }

15 /* Retrieves a message (from the network buffer) with the given type (e.g.,
       PROPOSE) from the given node (id) for the given consensus instance (
       consensus_id), blocking until such message arrives or the failure of the
        given node is detected (in which case it returns NULL). */
16 wait_consensus_msg(id, consensus_id, type);
```

Listing 1: Consensus helpers.

The consensus algorithm for the leader node is shown in Listing 2. The leader proposes a value and wait for the acknowledgement from all correct nodes. If all correct nodes acknowledge (*ACK* message is received), the value is locked and the leader broadcasts a decision. If some nodes reply with a non-acknowledgment (*NACK*), it means that this node knows from a value that was proposed from a previous leader that crashed. Because we want optimistic deliveries to be correct as often as possible, we favor older proposals. The leader then restarts the consensus

and proposes the oldest known value.

```
1  leader_consensus(proposal_t p) {
2     boolean restart = FALSE;
3
4     start:
5     // propose
6     broadcast({leader_id, PROPOSE, p, consensus_id});
7
8     // wait replies or failure notification for all nodes
9     for (id = 0; id < N; id++) {
10       // reply = {type, p}
11       reply = wait_consensus_msg(id, consensus_id, ACK | NACK);
12       if (reply == NULL)
13          continue; // this one is failed
14       if (reply.type == NACK && reply.p.id < p.id) {
15          // Some node has an better decision
16          p = reply.p;
17          restart = TRUE;
18       }
19    }
20
21    if (restart)
22       goto start; // restart with the better decision
23
24    // value locked, no opposition
25    broadcast({leader_id, DECISION, p, consensus_id);
26    consensus_id++;
27
28    return p;
29 }
```

Listing 2: Leader consensus.

Finally, the pseudocode for the consensus protocol running on the nodes other than the leader is shown in Listing 3. In the first phase the nodes wait from a proposal from the leader and check if this proposal is better than the one they have. As discussed above, proposals originated from a previous leader that failed are better than newer ones. If the proposal received is good, nodes acknowledge. Otherwise, they send non-acknowledgements ($NACK$) together with the better proposal. Then, in the second phase, the algorithm waits for a decision and indicates the end of consensus (by returning $TRUE$) or, if the leader failed, indicates the need for restart (by returning $FALSE$).

```
1  nonleader_consensus_estimate(proposal_t p) {
2     proposal_t p_l;

4     start:
5     // Phase 1: wait for proposal from leader
6     p_l = wait_consensus_msg(leader_id, consensus_id, PROPOSE);
7     if (p_l == NULL) {
8        // Nothing received (leader failure detected), change leader
9        leader_id = (leader_id + 1) % NUM_NODES;
10       if (leader_id == my_id) {
11          return p;
12       }
13       goto start; // try again with new leader
14    }

16    // Proposal received from the leader
17    if (p_l.id > p.id) {
18       // Node knows a proposal that may be already in optimistic use
19       send(leader_id, {NACK, p, consensus_id});
20    } else {
21       p = p_l; // adopt proposal from leader
22       send(leader_id, {ACK, p_l, consensus_id});
23    }

25    return p;
26 }

28 nonleader_consensus_decide() {
29    // Phase 2: wait for decision
30    p_l = wait_consensus_msg(leader_id, consensus_id, DECISION);

32    // nothing received (leader failed)
33    if (p_l == NULL) {
34       return FALSE;
35    }
36    return TRUE;
37 }
```

Listing 3: Non-leader consensus.

Bibliography

[AAB+05] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong H. Hwang, Wolfgang Lindner, Anurag S. Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the borealis stream processing engine. In *2nd Biennial Conference on Innovative Data Systems Research (CIDR'05)*, pages 277–289, 2005.

[ABB+04] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. STREAM: The Stanford Data Stream Management System, 2004.

[ABD+09] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009.

[ACCH09] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th USENIX Security Symposium*, Montreal, Canada, August 2009.

[Agg07] Charu Aggarwal, editor. *Data Streams: Models and Algorithms*. Springer, New York, NY, USA, 1 edition, 2007.

[AGK+05] Saman Amarasinghe, Michael I. Gordon, Michal Karczmarek, Jasper Lin, David Maze, Rodric M. Rabbah, and William Thies. Language and compiler design for streaming applications. *International Journal of Parallel Programming*, 33(2):261—-278, 2005.

[Ary95] Sunil Arya. *Nearest neighbor searching and applications*. Phd thesis, University of Maryland at College Park, 1995.

[AS00] Marcos Kawazoe Aguilera and Robert E. Strom. Efficient atomic broadcast using deterministic merge. In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 209–218, New York, NY, USA, 2000. ACM.

[AY07] Charu C. Aggarwal and Philip S. Yu. A survey of synopsis construction in data streams. In Charu Aggarwal, editor, *Data Streams: Models and Algorithms*, Advances in Database Systems, chapter 9, pages 169–208. Springer, 2007.

[BAAS09] R. Bocchino, V. Adve, S. Adve, and M. Snir. Parallel programming must be deterministic by default. In *First USENIX Workshop on Hot Topics in Parallelism*, 2009.

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Symposium on Principles of Database Systems*, pages 1—-16, New York, NY, USA, 2002. ACM.

[BBMS08] Magdalena Balazinska, Hari Balakrishnan, Samuel R. Madden, and Michael Stonebraker. Fault-tolerance in the borealis distributed stream processing system. *ACM Trans. Database Syst.*, 33(1):1–44, 2008.

[BC02] Daniel Bovet and Marco Cesati. *Understanding the linux kernel*. O'Reilly \& Associates, Inc., Sebastopol, CA, USA, 2 edition, 2002.

[BF07] Andrey Brito and Christof Fetzer. Improved event processing performance through parallel event transformation. In *Proceedings of The Second International Workshop on Event-driven Architecture, Processing and Systems (EDA-PS'07) at the 33rd International Conference on Very Large Data Bases (VLDB 2007)*, Vienna, Austria, 2007. (Short paper).

[BFF09a] Andrey Brito, Christof Fetzer, and Pascal Felber. Minimizing latency in fault-tolerant distributed stream processing systems. In *The 29th Int'l Conference on Distributed Computing Systems (ICDCS 2009)*, Los Alamitos, CA, USA, June 2009. IEEE Computer Society.

[BFF09b] Andrey Brito, Christof Fetzer, and Pascal Felber. Multithreading-enabled active replication for event stream processing operators. In *The 28th International Symposium on Reliable Distributed Systems*, pages 22–31, Los Alamitos, CA, USA, September 2009. IEEE Computer Society.

[BFSF08] Andrey Brito, Christof Fetzer, Heiko Sturzrehm, and Pascal Felber. Speculative out-of-order event processing with software transaction memory. In *DEBS '08: Proceedings of the second international conference on Distributed event-based systems*, pages 265–275, New York, NY, USA, 2008. ACM.

[BGAH07] R. Barga, J. Goldstein, M. Ali, and M. Hong. Consistent streaming through time: a vision for event stream processing. In *Proceedings of the third biennial conference on Innovative data systems research (CIDR'07)*, Asilomar, USA, January 2007.

[BKI06] Claudio Basile, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Active replication of multithreaded applications. *IEEE Trans. Parallel Distrib. Syst.*, 17(5):448–465, 2006.

[BMST93] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The primary-backup approach. In *Distributed systems (2nd Ed.)*, pages 199–216. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[Bri08] Andrey Brito. Optimistic parallelization support for event stream processing systems. In *MDS '08: Proceedings of the 5th Middleware doctoral symposium*, pages 7–12, New York, NY, USA, 2008. ACM.

[BWS+10] Andrey Brito, Stefan Weigert, Martin Süßkraut, Christof Fetzer, and Pascal Felber. Handling crash and software faults efficiently in distributed event stream processing. Submitted to the Third International Conference on Dependability (DEPEND'10), 2010.

[CA95] Liming Chen and A. Avizienis. N-version programming: a fault-tolerance approach to reliabiliy of software operation. *Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'., Twenty-Fifth International Symposium on*, pages 113–119, Jun 1995.

[CA08] Sharma Chakravarthy and Raman Adaikkalavan. Events and streams: harnessing and unleashing their synergy! In *DEBS '08: Proceedings of the second international conference on Distributed event-based systems*, pages 1—-12, New York, NY, USA, 2008. ACM.

[CASD95] Flaviu Cristian, Houtan Aghili, Ray Strong, and Danny Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. *Information and Computation*, 118(1):158 – 179, 1995.

[CCC07] K. Mani Chandy, Michel. Charpentier, and Agostino Capponi. Towards a theory of events. In *ACM International Conference Proceeding Series; Vol. 233*, pages 180—-187, New York, NY, USA, 2007. ACM.

[CCD+10] 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 *EuroSys '10: Proceedings of the 5th ACM European conference on Computer systems*, 2010.

[CCFC04] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theoretical Computer Science*, 312(1):3–15, 2004.

[CEF+06] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: a secretless framework for security through diversity. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.

[CJSS03] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: a stream database for network applications. In *International Conference on Management of Data*, pages 647—-651, New York, NY, USA, 2003. ACM.

[CM79] K. Mani Chandy and Jayadev Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, 5(5):440—-452, 1979.

[CM05] Graham Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55:58–75, 2005.

[CPR+92] M. Chereque, D. Powell, P. Reynier, J.-L. Richier, and J. Voiron. Active replication in delta-4. In *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, pages 28–37, Jul 1992.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[Day94] Umeshwar Dayal. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.

[DBB+88] Umeshwar Dayal, Barbara Blaustein, Alejandro Buchmann, Upen Chakravarthy, Meichun Hsu, R. Ledin., Dennis McCarthy, Arnon Rosenthal, Sunil K. Sarin, Michael J. Carey, Miron Livny, and Rajiv Jauhari. The HiPAC project: combining active databases and timing constraints. *ACM SIGMOD Record*, 17(1), 1988.

[DH08] Stephan Diestelhorst and Michael Hohmuth. Hardware acceleration for lock-free data structures and software-transactional memory. In *Proceedings of the 2008 Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods*, April 2008.

[DSS98] X. Défago, A. Schiper, and N. Sergent. Semi-passive replication. In *SRDS '98: Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, page 43, Los Alamitos, CA, USA, 1998. IEEE Computer Society.

[DSS06] Dave Dice, Ori Shalev, and Nir Shavit. Distributed Computing. In *International Symposium on Distributed Computing*, volume 4167 of *Lecture Notes in Computer Science*, pages 194–208, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[DTT99] A. M. Déplanche, P. Y. Théaudière, and Y. Trinquet. Implementing a semi-active replication strategy in chorus/classix, a distributed real-time executive. In *SRDS '99: Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, page 90, Washington, DC, USA, 1999. IEEE Computer Society.

[EAWJ02] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114—-131, 2003.

[Esp10] EsperTech Inc. Esper complex event processing website. `http://esper.codehaus.org/`, March 2010.

[FBFJ10] Christof Fetzer, Andrey Brito, Robert Fach, and Zbigniew Jerzak. Streammine. In Annika Hinze and Alex Buchmann, editors, *Principles and Applications of Distributed Event-based Systems*. Information Science Publishing, Hershey, PA, US, 2010.

[Fet03] Christof Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Transactions of Computers*, 52:99–112, Feb 2003.

[FFM+07] Pascal Felber, Christof Fetzer, Ulrich Müller, Torvald Riegel, Martin Süßkraut, and Heiko Sturzrehm. Transactifying applications using an open compiler framework. In *TRANSACT*, August 2007.

[FFR08] 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.

[GAW+08] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. SPADE: the system s declarative stream processing engine. In *International Conference on Management of Data*, pages 1123—-1134, New York, NY, USA, 2008. ACM.

[GN08] P. Godefroind and N. Nagappan. Concurrency at microsoft - an exploratory survey, 2008. EC2 - Exploiting Concurrency Efficiently and Correctly (in conjunction with the 8th International Conference on Computer Aided Verification), 2008.

[GO03] Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *ACM SIGMOD Record*, 32(2):5—-14, 2003.

[HBR+05] Jeong-Hyon Hwang, Magdalena Balazinska, Alexander Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. High-availability algorithms for distributed stream processing. In *ICDE 2005: Proceedings of the 21st International Conference on Data Engineering*, pages 779–790, Washington, DC, USA, 2005. IEEE Computer Society.

[HCZ08] Jeong-Hyon Hwang, U. Cetintemel, and S. Zdonik. Fast and highly-available stream processing over wide area networks. In *ICDE 2008: Proceedings of the 24th International Conference on Data Engineering*, pages 804–813, Washington, DC, USA, April 2008. IEEE Computer Society.

[HK08] 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, NY, USA, 2008. ACM.

[HLM06] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. *ACM SIGPLAN Notices*, 41(10):253—-262, 2006.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *ACM SIGARCH Computer Architecture News*, 21(2):289 – 300, 1993.

[IR05a] Milena Ivanova and Tore Risch. Customizable parallel execution of scientific stream queries. In *Very Large Data Bases*, pages 157—-168. VLDB Endowment, 2005.

[IR05b] Milena Ivanova and Tore Risch. Customizable parallel execution of scientific stream queries. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 157–168. VLDB Endowment, 2005.

[JAAS09] Robert L. Bocchino Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX Workshop on Hot Topics in Parallelism (HotPar'09)*, Berkeley, CA, USA, March 2009.

[Jef85] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(3):404–425, 1985.

[JPPnMA00] R. Jiménez-Peris, M. Patiño Martínez, and S. Arévalo. Deterministic scheduling for transactional multithreaded replicas. In *Reliable Distributed Systems, 2000. SRDS-2000. Proceedings The 19th IEEE Symposium on*, pages 164–173, 2000.

[KBDZ09] Kirk Kelsey, Tongxin Bai, Chen Ding, and Chengliang Zhang. Fast track: A software system for speculative program optimization. In *CGO '09: Proceedings of the seventh annual IEEE/ACM international symposium on Code generation and optimization*, New York, NY, USA, 2009. ACM.

[KPA+03] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann. Using optimistic atomic broadcast in transaction processing systems. *Knowledge and Data Engineering, IEEE Transactions on*, 15(4):1018–1032, July-Aug. 2003.

[LLD+07] Ming Li, Mo Liu, Luping Ding, Elke A. Rundensteiner, and Murali Mani. Event stream processing with out-of-order data arrival. In *ICDCSW '07: Proceedings of the 27th International Conference on Distributed Computing Systems Workshops*, page 67, Washington, DC, USA, 2007. IEEE Computer Society.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[LTS+08] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. Out-of-order processing: a new architecture for high-performance stream systems. In *Proceedings of the VLDB Endowment*, volume 1, pages 274—-288. VLDB Endowment, 2008.

[Luc01] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Mut05] S. Muthukrishnan. *Data Streams: Algorithms and Applications*. now Publishers Inc., Hanover, Ma, USA, 1 edition, 2005.

[NPCF08] Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. *SIGARCH Comput. Archit. News*, 36(1):308–318, 2008.

[NS05] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2005)*, 2005.

[NVCF06] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. In *Operating Systems Design and Implementation*, Berkeley, CA, USA, 2006. USENIX Association.

[OMH09] Cosmin E. Oancea, Alan Mycroft, and Tim Harris. A lightweight in-place implementation for software thread-level speculation. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 223–232, New York, NY, USA, 2009. ACM.

[PHW07] Donald E. Porter, Owen S. Hofmann, and Emmett Witchel. Is the optimism in optimistic concurrency warranted? In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pages 1–6, Berkeley, CA, USA, 2007. USENIX Association.

[Pro10] Progress Software Corporation. Website. `http://www.apama.com/`, March 2010.

[PS98] Fernando Pedone and André Schiper. Optimistic atomic broadcast. In *DISC '98: Proceedings of the 12th International Symposium on Distributed Computing*, pages 318–332, London, UK, 1998. Springer-Verlag.

[PS03] F. Pedone and A. Schiper. Optimistic atomic broadcast: a pragmatic viewpoint. *Theoretical Computer Science (Elsevier)*, 291(1):79–101, 2003.

[PS06] Kostas Patroumpas and Timos Sellis. *Window Specification over Data Streams*, volume 4254 of *Lecture Notes in Computer Science*, pages 445–464. Springer Berlin Heidelberg, 2006.

[RL04] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *In Proceedings of the 11th Annual Network and Distributed System Security Symposium*, pages 159–169, 2004.

[SATH+06] 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 *Principles and Practice of Parallel Programming*, pages 187—-197, New York, NY, USA, 2006. ACM.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22:299–319, 1990.

[ScZ05] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34(4):42–47, 2005.

[SCZM05] J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry. The STAM-Pede approach to thread-level speculation. *ACM Transactions on Computer Systems (TOCS)*, 23(3):253—-300, 2005.

[SDFW09] R. Strom, C. Dorai, T.H. Feng, and Zheng Wei. Deterministic Replay for Transparent Recovery in Component-Oriented Middleware. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 615–622, Washington, DC, USA, 2009. IEEE Computer Society.

[SFF09] Heiko Sturzrehm, Pascal Felber, and Christof Fetzer. TM-Stream: An STM framework for distributed event stream processing. In *Proceedings of the 2009 IEEE International Symposium on Parallel \& Distributed Processing*, pages 1—-8, Washington, DC, USA, 2009. IEEE Computer Society.

[SHB04] Mehul A. Shah, Joseph M. Hellerstein, and Eric Brewer. Highly available, fault-tolerant, parallel dataflows. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 827–838, New York, NY, USA, 2004. ACM.

[SHCF03] Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proceeding of the 19th Internationsal Conference on Data Engineering*, pages 25–36, 2003.

[SM98] J. Steffan and T Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, page 2, Washington, DC, USA, 1998. IEEE Computer Society.

[Smi81] James E. Smith. A study of branch prediction strategies. In *ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture*, pages 135–148, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[SPGV07] Jesper H. Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. Streamflex: high-throughput stream programming in java. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 211–228. ACM Press, New York, NY, October 2007.

[SPMO02] A. Sousa, J. Pereira, F. Moura, and R. Oliveira. Optimistic total order in wide area networks. In *21st IEEE Symposium on Reliable Distributed Systems, 2002. Proceedings.*, pages 190–199. IEEE Comput. Soc, 2002.

[SS05] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

[SS06] Ori Shalev and Nir Shavit. Predictive log-synchronization. *ACM SIGOPS Operating Systems Review*, 40(4), 2006.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[Str10] StreamBase Systems Inc. Website. `http://www.streambase.com/`, March 2010.

[SWK+10] Martin Süßkraut, Stefan Weigert, Thomas Knauth, Ute Schiffel, Martin Meinhold, and Christof Fetzer. Prospect: A compiler framework for speculative parallelization. In *Proceedings of The Eighth International Symposium on Code Generation and Optimization (CGO)*, April 2010.

[SY85] Rob Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, 1985.

[SZ05] Fred B. Schneider and Lidong Zhou. Implementing trustworthy services using replicated state machines. *IEEE Security and Privacy*, 3(5):34–43, 2005.

[TcZ07] Nesime Tatbul, Uğur Çetintemel, and Stan Zdonik. Staying fit: efficient load shedding techniques for distributed stream processing. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 159–170. VLDB Endowment, 2007.

[Tom67] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.

[UC 10] UC Berkeley EECS Dept. The ptolemy project website. `http://ptolemy.eecs.berkeley.edu/`, March 2010.

[Vit85] Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37—-57, 1985.

[vPCC07] Christoph von Praun, Luis Ceze, and Calin Caşcaval. Implicit parallelism with ordered transactions. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 79–89, New York, NY, USA, 2007. ACM.

[VR01] Paulo Verissimo and Luis Rodrigues. *Distributed Systems for System Architects*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[Wam08] Jons-Tobias Wamhoff. Gibraltar binary transactification tool. Master's thesis, Dresden University of Technology, 2008.

[WDR06] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *International Conference on Management of Data*, pages 407—-418, New York, NY, USA, 2006. ACM.

[Wel71] T. A. Welch. Bounds on information retrieval efficiency in static file structures. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1971.

[WF07] Ute Wappler and Christof Fetzer. Software encoded processing: Building dependable systems with commodity hardware. In *Lecture Notes in Computer Science on Computer Safety, Reliability and Security (SafeComp 2007)*, 2007.

[WH95] David A. Wood and Mark D. Hill. Cost-effective parallel computing. *Computer*, 28(2):69–72, 1995.

[WV01] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.