TECHNISCHE UNIVERSITÄT DRESDEN

Data Replication in Hybrid Memory Database Systems

Dissertation

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

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von
Mikhail Zarubin, M.Sc.
geboren am 2. November 1988 in Rostow am Don, UdSSR (Russland)

Gutachter:	**Prof. Dr.-Ing. Wolfgang Lehner** Technische Universität Dresden Fakultät Informatik Institut für Systemarchitektur Lehrstuhl für Datenbanken 01062 Dresden
Fachreferent:	**Prof. Dr.-Ing. Dirk Habich** Technische Universität Dresden Fakultät Informatik Institut für Systemarchitektur Lehrstuhl für Datenbanken 01062 Dresden
Tag der Verteidigung:	7. März 2022

Dresden, im Januar 2022

ABSTRACT

The recent advances in hardware technologies – i.e. highly scalable multi-core NUMA architectures and non-volatile random-access memory (NVRAM) – lead to significant changes in the architecture of in-memory database systems. The novel memory type allows persistent writes while featuring DRAM-like characteristics – byte addressability, high bandwidth, and low access latencies. It is likely to complement or replace the block-based secondary storage (e.g., HDDs or SSDs) for storing the primary data of the DBMS. Therefore, the next generation of highly-performant scalable database systems will rely on single-level hybrid memory (e.g., compound exclusively of DRAM and NVRAM) NUMA architectures and is expected to keep the primary data solely persistent in NVRAM, while query processing could be executed on both mediums. Unfortunately, NVRAM faces certain drawbacks such as a lower write endurance, lower bandwidth, higher latencies, and - most importantly - an increased error-proneness compared to DRAM. Thus, efficient minimal-overhead data protection mechanisms have to be deployed in the underlined architectures to avoid primary data losses.

This thesis provides an analytical overview of such envisioned hybrid memory database systems, gives a survey of reliability techniques that are generally deployed in computing systems, identifies their strengths and weaknesses when used in hybrid memory databases. As a result, this work proposes effective adoption and optimization primitives for the software-managed data replication as the most applicable resilience approach. In particular, research focus is given to runtime and space (and, therefore, NVRAM wear-out) reduction of the replication overheads, while preserving strong resilience guaranties and instant recovery opportunities. Subsequently, this thesis proposes a rich set of techniques that leverage data replication for query processing needs to achieve high performance, allocation flexibility and effective hardware utilization in modern scale-up systems. The usefulness of the suggested improvements is thoroughly evaluated and analysed using a variety of low-level and end-to-end benchmarks and proofs-of-concepts.

CONTENTS

AKNOWLEDGMENTS

I would like to thank all my supervisors and colleagues for all their help and advice with this PhD. I would also like to thank all members of my family for all the support I received during the period of doctoral studies.

Mikhail Zarubin
January 3, 2022

INTRODUCTION

1.1 MOTIVATION

In modern digitalized world, data is generated in large amounts and is said to be the new oil of the 21st century. Thus, in numerous application areas ranging from science to industry, the importance of efficient and scalable data processing increases constantly, whereby traditional *ACID (atomicity, consistency, isolation, durability)* properties of database systems must be taken into account [CDN11]. To speed up the processing of increasing data amounts, the general trend of the last decade is shift towards scalable *in-memory* database architectures [KZZL17, ALR+17, Kim15]. Such in-memory databases achieve higher performance compared to those based on traditional disk storage (HDD or SSD). This is due to the fact that they rely on algorithms optimized for main memory, which are normally simpler and faster [KZZL17, KWJP16]. Moreover, processing information that has been stored in byte-addressable memory eliminates the seek time costs when searching and accessing data, compared to block-addressable disk storage.

While being fairly advantageous in many fields that require high-performance or real-time data processing, the in-memory database systems used to suffer two major drawbacks. The first one is the high costs of main memory volumes (e.g., DRAM or SRAM) which are much more expensive than those of disks. The second drawback is a lack of *durability* compliance. This refers to the loss of information in case of power-off. However, over the last years, the cost of DRAM has begun to decline, making the in-memory approaches more affordable and, thus, largely neglecting the first drawback. Furthermore, the recent advance in memory technologies – non-volatile random-access memory (NVRAM) – allows to mitigate the second drawback and provide the full durability compliance within in-memory paradigm.

This novel memory type combines storage-like persistent writes with features of traditional main memory – byte addressability, low latencies and high bandwidth. Due to such advantageous properties, it is likely to complement or replace the block-based secondary storage for storing the primary data of the DBMS. Thus, it is expected that the next generation of highly-performant in-memory database systems will rely on single-level hybrid memory (e.g., consisting exclusively of DRAM and NVRAM) scale-up architectures. Besides hybrid memory, scale-up hardware consists of several processing units or sockets interconnected within a single-box compute node [PALG19]. Therefore, targeted applications are potentially able to keep the primary data solely persistent in NVRAM, while query processing could be executed on both mediums. A number of respective changes and adoptions in the architecture of in-memory database systems are already proposed by science and industry [ALR+17, Kim15]. Such changes result in a modified in-memory approach, which is further referred by this thesis as a *hybrid memory database architecture*.

On the one hand, NVRAM allows to greatly extend available byte-addressable capacities on affordable prices while still facing certain drawbacks. Those are higher latencies, lower bandwidth, lower write endurance and - most importantly - an increased error-proneness, compared to DRAM or SRAM. On the other hand, the data durability and consistency require to deploy efficient minimal-overhead data protection mechanisms in the underlined architectures.

Thus, the goal of this thesis is to adopt or propose reasonable data protection mechanisms for hybrid memory databases. As at this point in time, respective safety measures (with regard to NVRAM) are not yet well researched. So, the thesis starts with a survey of resilience techniques that are generally used in information processing systems, explores their potential strengths and weaknesses when applied to hybrid memory database systems. And as an outcome, this work proposes effective deployment and optimization

primitives for the software-coordinated physical data replication as the most applicable reliability approach. With regard to the data storage model the investigations rely on a columnar data organization [BMK99, BZN05, HDU+19, SAB+05] as the most efficient state-of-the-art format in analytical in-memory processing systems.

The emphasis is given to the compute node-local (e.g., addressed within single-box scale-up machine) minimization of runtime and space overheads of the data replication process, while preserving strong resilience guaranties and instant recovery opportunities. Subsequently, this thesis suggests a rich set of techniques that leverage data replication (particularly in compressed data formats [AMF06, GRS98]) for query processing needs to achieve high performance, allocation flexibility and effective hardware utilization in modern scale-up systems (e.g., via deployment of multiple sockets). Additional attention is paid to the single instruction multiple data vectorization as performance optimization opportunity for physical replication and other in-memory database operations [PRR15, ZR02, DUP+20]. The applicability of the proposed improvements is thoroughly evaluated using a variety of benchmarks and proofs-of-concepts.

1.2 SUMMARY OF CONTRIBUTIONS

This thesis targets concepts and low-overhead mechanisms for provision of data reliability guaranties in hybrid memory databases, being run on scale-up hardware architectures. Furthermore, the deployment opportunities of redundant data copies for query processing needs are precisely researched. In addition, several benchmarks and proof-of-concept prototypes are used to evaluate proposed solutions. The contributions can be summarized as follows:

1. First, the thesis details on the foundations of both envisioned hybrid memory database systems and targeted scale-up hardware architectures. Because of the novelty of NVRAM memory within modern hardware landscape – particular attention is given to the low-level evaluation of its performance characteristics to make a foundation for the efficient scale-up system level integration.

2. Based on the initial explorations, the principles of how the data reliability could be ensured for such novel data processing applications are revised, compared to traditional architectures. This is done via the discussion and qualitative analysis of resilience approaches that are generally deployed in computing systems. As a result, that survey concludes that the software-managed data replication is the most applicable reliability technique. Further, a number of important research challenges are drawn when attempting to integrate respective protection mechanism in hybrid memory database systems.

3. The ultimate goal is to minimize the *runtime* overhead of the data replication process – as thorough evaluation of the state-of-the-art implementation demonstrates prohibitively low performance for the targeted domain of high-speed hybrid memory databases. Thus, the detailed insights on sources of the runtime replication overhead are provided. Furthermore, this thesis suggests a rich set of optimization primitives to reduce the performance penalty of replication for typical database workloads. These optimizations are subsequently automated using a template-based approach to adapt for current workload conditions.

4. To reduce the *space* overhead of the replication (which normally implies at least 100% of additional storage per replica), this thesis investigates the opportunities of data compression in NVRAM-centric data stores focusing on lightweight integer compression techniques. Such algorithms are widely used in DRAM-backed columnar processing systems. Subsequently, a *polymorphic compressed replication* concept is suggested to store replicas using a variety of compressed data formats and, therefore, to average possible space reductions.

5. The scale-up hybrid memory database systems are able to persistently store and to efficiently process data exclusively in main memory. These specific properties enable the *immediate* on-demand usage of replicas for other than reliability purposes. Thus, this thesis derives a conceptual vision on how compute node-local physical replication, while efficiently facilitating the primary data protection, could be leveraged for query processing needs as well. In particular, it suggests to allow data processing operations on (possibly compressed) replicas flexibly allocated over the sockets of scale-up server. As evaluated, that allows to significantly speed up concurrent workloads and improve hardware utilization.

6. As all the data (including replicas) processed by the database operators in targeted systems is stored on byte-addressable mediums – it is directly exposed to the available CPU instructions. Thus, the data level parallelism or *single instruction multiple data (SIMD)* vectorization could be deployed for optimization of NVRAM-centric operations similarly to DRAM-backed in-memory databases. The usefulness of SIMD instructions is researched by this thesis not only for physical data replication, but also for query processing on persistent replicas. Furthermore, a novel concurrent SIMD-heterogeneous query execution model (SIMD-MIMD cocktail) is proposed to outpace the traditional SIMD-homogeneous performance.

7. The usefulness of the suggested improvements is thoroughly evaluated and analyzed using a variety of low-level and end-to-end benchmarks and proofs-of-concepts. In particular, the suggested *polymorphic compressed replication* and *SIMD-MIMD cocktail* approaches are evaluated within the MorphStore[1] – a prototype of an in-memory query processing system for columnar data developed at the Chair of Databases of the TU Dresden.

1.3 OUTLINE

Figure 1.1 visualizes an abstract outline of this thesis. The corresponding structure matches in part with the previously presented summary of contributions. Introduction aside, the remaining part of the thesis is compound of the following blocks. Chapter 2 provides the necessary background on the key technologies and architectures addressed in the thesis. In particular, it presents the non-volatile random access memory technology and its influence on modern hardware architectures. Further, it discusses the deployment challenges from the user software point of view and expected integration advantages in the database domain. As a conclusion, this chapter stresses the need for effective protection of NVRAM-resident data and draws the respective research challenges tackled further in the thesis.

Chapter 3 focuses on a compute node-local synchronous physical replication as the most reasonable for hybrid memory systems resilience approach. Namely, it evaluates the

[1] https://github.com/MorphStore

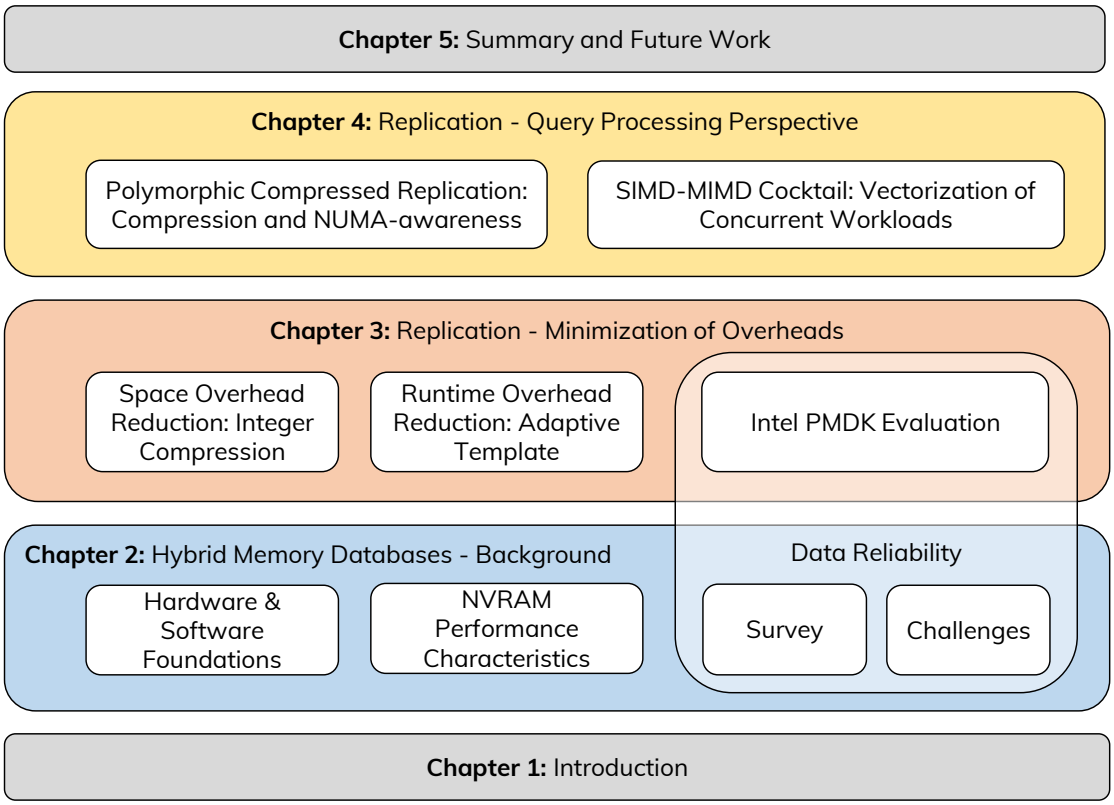


Figure 1.1: Thesis structure and outline.

state-of-the-art implementation and presents the compute node-local mechanisms to provide the foundation for an efficient NVRAM replication with a low latency and throughput penalty. Further, the evaluation and adoption of the lightweight integer compression techniques are provided to ensure the space footprint-minimized replication. Finally, this chapter proposes flexible NUMA-aware allocation for stronger protection of persistent base data against all possible NVRAM failures. The evaluation of the proposed in this chapter ideas is embedded in respective sections for better integrity and consistency.

Chapter 4 describes a conceptual vision on how the compute node-local physical replication, while efficiently providing the base data protection, could be deployed for query processing purposes as well. The research starts with the discussion of the underlying system model, and then proposes and evaluates two options for the replication-related query processing optimizations. First option (polymorphic compressed replication) leverages the *storage* component (via diversity in data layouts), while second (SIMD-MIMD cocktail) proposes to rely on *compute* counterpart (via diversity in available vectorized instruction set extensions). The evaluation of the suggested in this chapter ideas is integrated in respective sections as well.

Chapter 5 concludes the thesis with a summary of this manuscript. Finally, an outlook for the most promising future research directions, that could complement or expand the contents of this thesis is provided.

BACKGROUND AND CHALLENGES

This chapter[1] gives the necessary introduction to the key technologies and system architectures addressed in the thesis. Particularly, it presents the novel persistent memory technology, its impact on modern hardware architectures, deployment challenges from user software point of view, and expected integration advantages in the database domain. As a conclusion, this chapter stresses the motivation and research challenges tackled further in the thesis.

2.1 HYBRID MEMORY SYSTEMS

This part provides the necessary background regarding the non-volatile random access memory as a key enabler for the hybrid memory systems, which are presented subsequently from both hardware and software perspectives.

2.1.1 Non-volatile random access memory

This section is devoted to the description of the recent [2] storage/memory hardware advance – non-volatile random access memory (also known as persistent memory, storage class memory or non-volatile main memory). Further in the thesis, the term *non-volatile random access memory* is used to address a set of underlined hardware technologies that provide byte-addressable access to the persistent devices attached directly to CPU memory bus.

A general trend, currently observable in many memory-based data intensive applications like in-memory databases, key-value stores, data warehousing, and analytical processing tools is the growing demand for hardware platforms with increasingly larger memory volumes [MMNLM20]. This need reflects the general processing style of such systems, as they tend to constantly have most of the data in main memory, in contrast to disk-centric architectures. On the other hand, the technological process used to scale the frequency or capacity of modern volatile main memory devices is facing its physical limitations [Loh08, QSR09]. To tackle this challenge, industry is intensively researching and integrating new memory technologies. One of the most notable recent developments is non-volatile random access memory (NVRAM).

In essence, NVRAM is a random-access memory that retains data without applied power. Such durability is a novel feature compared to dynamic random-access memory (DRAM) and static random-access memory (SRAM), which have been dominating volatile random access memory technologies over the last decades, as both maintain data only for as long as power is applied. The term *random access* means that the access latency is equal for all internally stored bytes. To complement such volatile memories and provide persistency property the computing systems conventionally used block-addressable (accessed over the slow I/O bus) storage devices such as solid state drives (SSDs), hard disk drives (HDDs) or magnetic tapes. Therefore, nowadays, NVRAM is able to close the gap between persistent and volatile memory classes and transforms the traditional memory hierarchy pyramid of Von Neumann architecture [GH93] as illustrated in Figure 2.1. Namely, NVRAM combines features of both – byte-addressability, low access latencies,

[1] Parts of the material in this chapter have been developed jointly with Thomas Kissinger, Dirk Habich, Thomas Willhalm, and Wolfgang Lehner. Namely, Section 2.2 is based on [ZKH+19]. The copyright of [ZKH+19] is held by Springer-Verlag GmbH Germany, part of Springer Nature; the electronic version of the article is available at https://link.springer.com/article/10.1007%2Fs00778-019-00549-w.

[2] as of year 2021

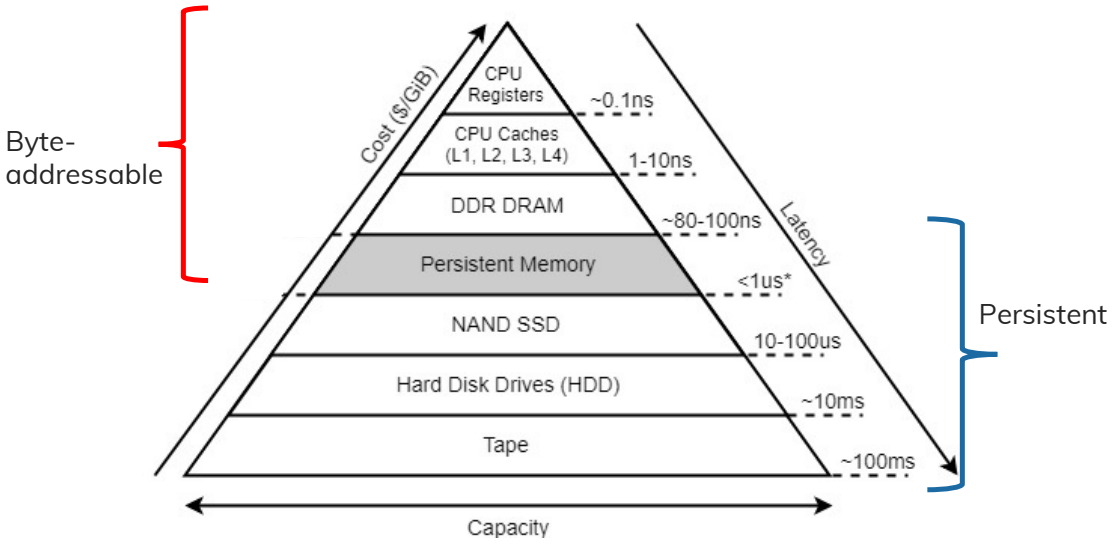


Figure 2.1: Memory hierarchy pyramid indicating the place of persistent memory (adopted from [Sca20]).

durability, and high capacities. Such combination is particularly interesting and promising in several domains of science and industry (e.g., data processing applications, high performance computing). The goal of this thesis is to examine it from database systems point of view taking into account not only its advantageous features, but also drawbacks such as limited endurance and increased, compared to DRAM, error-proneness.

There is a variety of actively developed hardware technologies which goal is to provide NVRAM-required properties. Those include Ferroelectric RAM [BAKS16], Magnetoresistive RAM [Hei14], FeFET [PLH21], Spin-transfer Torque [FKV+16] and others. However, the most stable and already applied in real hardware technology is Phase Change Memory (PCM) [WRK+10]. PCM is an NVRAM, mainly based on chalcogenide glass, and is sometimes referred to as CRAM. Such chemical substances fills up the memory cells and are able to change the phase state. This state switching in PCM is based on the presence of two different solid-state phases, i.e., crystalline and amorphous with different electrical resistivity. The information-storing ability in phase change cells is provided by the transition between the low resistive crystalline phases to high resistive amorphous phase [GS20]. The transition is induced by the high temperature, and can hold several intermediate states representing up to four bits per memory cell.

2.1.2 Scale-up hybrid memory architecture on hardware level

In the next sections this work relies on the Phase Change RAM devices from Intel named *Optane DC Persistent Memory* as a physical ready-to-use implementation of NVRAM hardware. These devices are currently shipped using the *non-volatile dual in-line memory module (NVDIMM)* packaging also known as *persistent memory module (PMM)*. The NVDIMMs mostly follow the traditional DRAM DIMMs hardware interface [Dim21]. Therefore, they are directly attached to the memory buses similarly to DRAM as illustrated by Figure 2.2. Precisely, this example denotes the current state-of-the-art (though, not the only possible) attachment scheme of persistent memory modules to the integrated memory controllers (IMCs) while using the shared connection to memory channels. Thus, each channel serves a pair of DRAM and NVRAM units. In such a machine

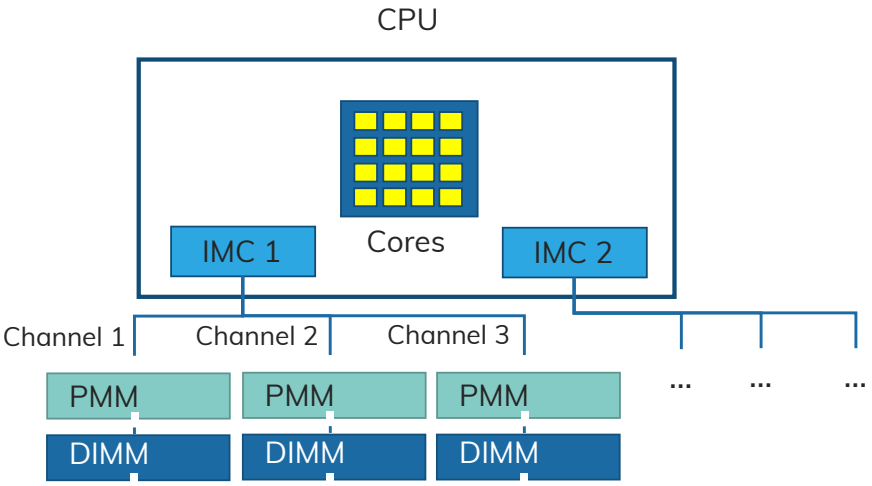


Figure 2.2: Typical attachment scheme of NVDIMMs in a single socket machine.

the CPU is equipped with 2 identical IMCs enabling 6 memory channels in total. The co-existence of the two byte-addressable memory types within a single architecture makes it actually *a hybrid memory system*. And in the considered example NVRAM is placed next to DRAM (or at a same level of memory hierarchy) that reflects a *single-level* hybrid memory approach.

One of the fundamental requirements for the modern high performance server systems is *scalability*. The scalability is the property of a system to handle a growing amount of work by adding resources to the system [Bon00]. The state-of-the-art method applied in practice for scaling up hardware in terms of compute cores and memory resources is known as as non-uniform memory access architecture (NUMA). This approach (also referred as *scale-up multiprocessor system*) physically partitions compute cores and assigns local hardware memory resources to each partition or socket [PALG19]. Thus, the NUMA architectures implicitly assume that higher performance is enabled in combination with a certain *parallelism* paradigm that make use of such partitions [PRR15]. There are three sources of parallelism provided by modern scale-up hardware: thread parallelism [Pac11], instruction-level parallelism [PM13], and data parallelism (discussed further in Section 2.3).

The NUMA processors or sockets are connected using an interconnect network (e.g., Unified Pair Interface [Int09]) that typically increases the response time when accessing memory of a remote socket. Such increases are referred to as a non-uniform memory access behavior. Several previous works prove that considering the NUMA effect is crucial for competitive data processing performance [KZZL17, PJHA, KKS+14b]. In principle, NUMA-oriented scale-up systems behave like distributed systems, but feature a faster communication due to cache coherency facilities and the close proximity of processors. Thus, the near-memory processing paradigm (NMP) is state-of-the-art on such platforms [KZZL17, PJHA, KKS+14b]. That means, NMP restricts data access to memory-local compute resources and only the unavoidable communication is performed via interconnects.

In such scale-up systems, which are now becoming a standard in the field of high performance server architectures, each local memory domain consists of DRAM and NVRAM at the same hierarchy level [DKK+14] making them actually *hybrid memory* systems. As a conclusion, both mediums are directly exposed to the multi-core CPUs, whereby every

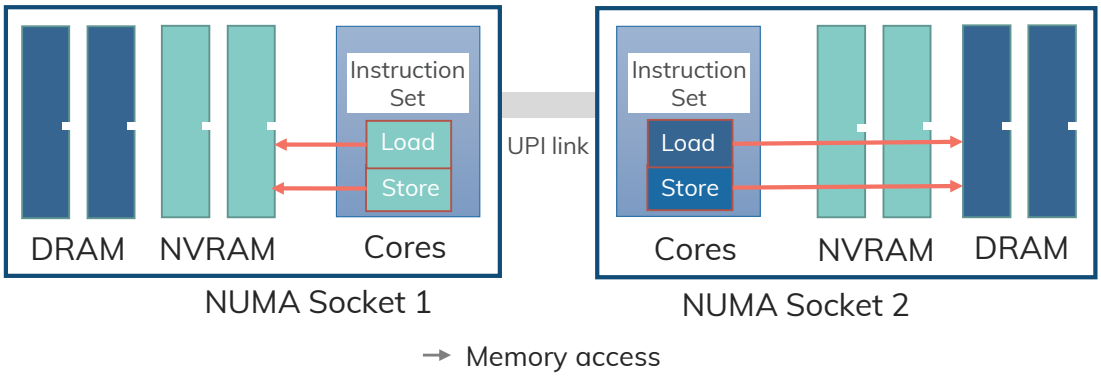


Figure 2.3: Schematic view of 2-socket hybrid memory system.

core may feature a full set of available instructions (e.g., load, store, vectorized extensions, etc..). Such placement enables a unified memory access style on both mediums as shown in Figure 2.3.

For completeness it is important to notice that, in principle, NVRAM could be mounted as a slow extension of DRAM capacity [Nvr20] (violating its persistency property), intermediate cache level between DRAM and I/O storage [CB18], or just as a fast block-addressable device similarly to flash memory devices (sacrificing its byte-addressability). However, these are not mainstream options and obviously may hurt the advantages of using NVRAM in the field of highly-performant database management systems. Therefore, in the following this thesis relies on the *single-level hybrid memory approach*.

2.1.3 Scale-up hybrid memory architecture on software level

To enable the effective exploitation of the described above hardware architectures in a variety of application domains, the software counterpart needs certain support or modifications compared to traditional (DRAM for processing, disk for persistency) approach. That applies to both operating system and user program levels. The basic requirements for such conversion, as well as main adoption principles are provided by Storage and Network Interfaces Association (SNIA) NVRAM programming model [Sni17]. The most important and relevant concepts are discussed below, while the actual focus is given to NVRAM-related issues within hybrid memory systems, as DRAM counterpart basically mimics the in-memory processing paradigm.

Access Methods. As already mentioned above, there is a number of alternatives in a way how NVRAM can be architectured in the system. The most important software-defined options are shown in Figure 2.4. Under Linux, the NVRAM resources are provided in the system through specialized *NVDIMM driver*, while OS-level administration is done via the *ndctl* command line utility [ndc21]. Further, the deployment of persistent memory as a storage would require an application to use either standard raw device access [ndc21] or standard file-based API via conventional or persistent memory-aware file system [XZM+17]. However, to efficiently access physical NVRAM as a main memory in byte-addressable way (using *load/store* semantics), current Linux kernels (as well as Windows and MacOS) implement the *Direct Access (DAX)* functionality. This feature maps physical NVRAM regions into the virtual address space of an application while bypassing kernel page and block-level caches. The following three specific ways of accessing NVRAM in byte-addressable mode are available in Linux-like OS:

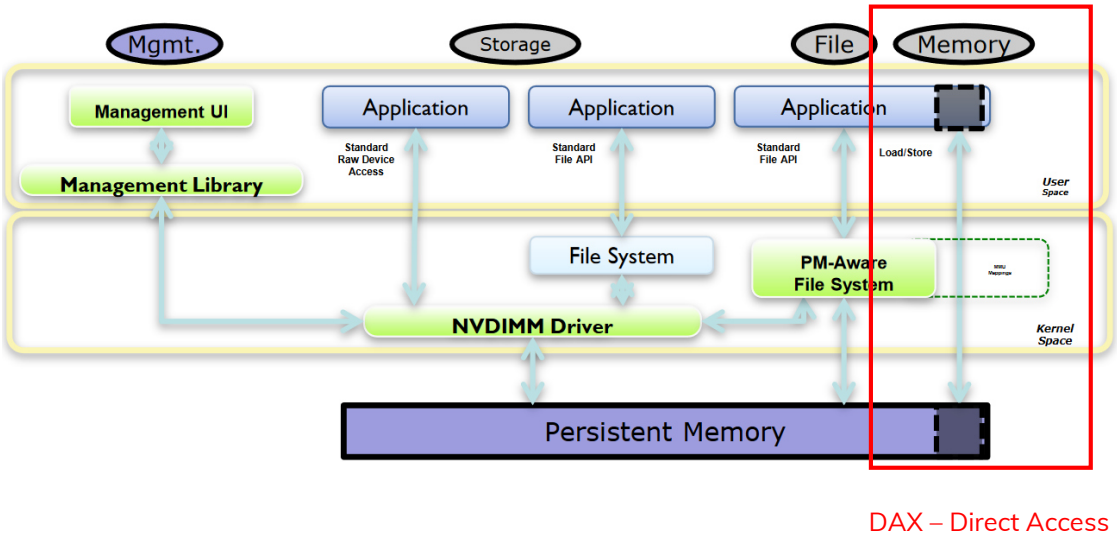


Figure 2.4: SNIA-recommended NVRAM deployment methods (adopted from [Sca20]).

DAXDEV. A DAX device is similar to a raw disk access. In this case, the complete region of the device is designated to one persistent pool. The pages of the NVRAM region are contiguously mapped into the address space of the application without any intervening file system.

DAXFS. The DAXFS access method requires the region of NVRAM to be formatted with a DAX-enabled file system (e.g., EXT4 or XFS) and individual files are mapped into the virtual address space of the application without intervening kernel page caches. Similar to the DAX device, this access method requires the NVRAM to be configured in *AppDirect* mode [Nvr20] whereby it is made explicitly visible as main memory to the operating system.

SHM. While the previous access methods require a specific physical memory region to be defined as NVRAM at boot time, the *shared memory (SHM)* access method uses arbitrary DRAM memory pages (via tmpfs file system) and is thus solely meant for DRAM-based NVRAM emulation.

Persistency Mechanisms. Because of the *DAX* feature, virtual memory pages are directly mapped to physical NVRAM pages and thus, no system calls are required to persist modified data to the NVRAM. However, the presence of volatile CPU caches requires explicit cache line write backs to ensure the persistency of a modification. Moreover, to implement such persistent flush mechanism in a consistent way, the cache line eviction has to be strongly ordered via fence commands to prevent possible out-of-order execution of store instructions. There are several alternatives to ensure consistent persistent flushing on current x86 processors:

CLFLUSH. This instruction writes back a specific cache line and evicts it from the cache hierarchy in the coherency domain.

CLFLUSHOPT. This instruction was introduced to support weakly-ordered higher performance flushes as optimized version of `CLFLUSH`, which is synchronized by previously issued `CLFLUSH` commands. The advantage is that multiple cache lines can be flushed concurrently, thus, exploiting instruction-level parallelism. Nevertheless,

this command requires an additional SFENCE or MFENCE instruction to ensure the completion of all previously issued CLFLUSHOPT instructions, and, therefore, prevent write reorderings. However, this requirement plays a protective role and does not specify an exact number of CLFLUSHOPT operations served by a single fence.

CLWB. This instruction asynchronously writes back a specific cache line *without* explicit eviction. CLWB also requires an SFENCE or MFENCE to ensure its completion.

WBNOINVD. The instructions of this family are only executable in kernel mode (or privileged OS mode) and write back all modified cache lines either with or without eviction. The WBNOINVD instruction is currently available on the Icelake server CPU generation.

MOVNT. The non-temporal MOV instruction bypasses the caches and stores data directly to the memory. MOVNT is a single instruction multiple data extension (SSE2, AVX, AVX2 or AVX-512) that is also executed out-of-order and thus requires an SFENCE or MFENCE to ensure its completion.

PAT. *Page Attribute Tables* (PAT) are used to control the caching behavior of memory pages and have been originally implemented for writing to device memory (e.g., frame buffers). PATs can be leveraged to automatically write back certain memory locations without the need of explicit cache line flush instructions. However, this mechanism exhibits a poor performance according to the measurements using a patched Linux kernel supporting PAT configuration for NVRAM mappings.

Persistent Memory Allocation. The volatile memory allocation is a well-understood and thoroughly-studied research topic. The variety of general purpose industrial and tuned-up scientific allocators are available. However, there are several reasons making DRAM allocators invalid for straight use in combination with NVRAM. Those are the following:

Recoverability. Unlike to the data stored in the volatile memory, where the virtual pointers addresses are preserved only while the program is running, NVRAM-resident objects are supposed to survive the application termination or crashes. That calls for a recoverable addressing space when providing persistent memory volumes. Such a recoverability could be provided via specialized allocation schemes that extends traditional pointers (e.g., returned by *malloc()* call) with durable fixed anchors (e.g., pool starting address) via memory mapped files (attached using *mmap()* call).

Leakage. As mentioned above, NVRAM-allocated chunks has to survive all termination scenarios. That implies to the fact that if memory leaks (independently of the actual reason) – it will retain durably inaccessible, as there is no OS garbage collection mechanism for persistent volume. Therefore, additional measures have to be implemented on both allocator and application sides to monitor actual state of the allocated chunks and release leaked pointers.

Atomic Updates. The persistent memory leakage could result from the incorrect programming but also from the power loss or OS-induced crashes. These reasons could also leave the user content of persistent memory in an inconsistent state if an update operation was terminated. On Intel hardware, the atomic persistent store is 8 bytes. That means if the program or system crashes while an aligned 8-byte store to persistent memory is in operation, on recovery those 8 bytes will either contain the old contents or the new contents. Thus, another important issue within NVRAM-related programming is to ensure the atomicity of persistent updates. That is normally solved applying transactional style in modifications (e.g., complete or nothing). Such transactions could be supported on allocator level deploying 8-byte atomic instructions.

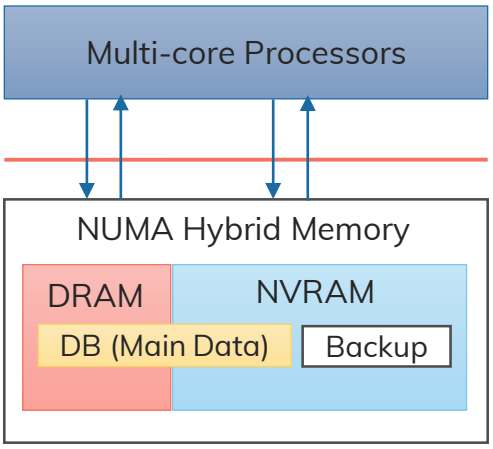


Figure 2.5: Schematic view of a hybrid memory database management system on scale-up architectures.

The discussed above challenges of persistent memory allocation, access and management are addressed in specialized allocators [MAK+13, SBF+15, YXD+17, YXD+15, BCB16, AvR18, DBY+19], tool-kits [Rud15], as well as on the data structure level [OLN+16, OL17a, VTRC11, Vig14, CGN11, CJ15, YWW+16, KSKN18].

Scale-Up Deployment. Previous paragraph described the NVRAM-relevant issues in software adaptation and development. To fully exploit NUMA resources of *scale-up* hybrid memory systems it is also required to support usage of multiple sockets in terms of both memories and multi-core CPUs. These challenges are already addressed in DRAM-baked NUMA-aware applications [KKS+14a]. Thus, due to the byte-addressability property of persistent memory, it is natural to extend the existing concepts and solutions (e.g., NMP and *libnuma*-like NUMA-aware memory management [lib20]) for NVRAM domain.

2.1.4 Hybrid memory database system

After presentation of the necessary background on hybrid memory systems, it is finally possible to introduce the respective DBMS applications that are expected to be accommodated by such systems. The high-level vision is given by Figure 2.5. Here, the main conceptual difference from classical DB architecture is that there is assumed no block-addressable persistent storage device. So, all the primary data including back-ups is placed in byte-addressable NVRAM, that replaces the traditional storage elements. Further, similarly to DRAM-backed in-memory DBMS applications [KKS+14a], the data processing and query execution can be done in fast volatile memory (that would require moving or copying of certain amount of persistent primary content to DRAM pool) or, novelly, directly in persistent domain as it is now byte-addressable [Kim15, ALR+17]. The latter approach, however, calls for more careful data processing procedures as there could be no further safe copy of base data. Therefore, not only medium failures but also errors during query execution (independently of their origins) may lead to the corruption of primary data. Nevertheless, such innovative "completely in-memory" architectures offer a great potential for highly-performant data accesses and instant recovery opportunities (especially if the DBMS state variables are also accommodated by NVRAM) and, thus, are the subject for further investigations in this thesis.

Data and Processing Model. With regard to the data organization model it is envisioned that such hybrid memory database systems, similarly to most of the modern in-memory applications, would favor columnar format [ABH+13, AMF06, DKB+19, LMF+16, RAB+13, ZHNB06, KD18]. Here, the relational data is maintained using the decomposition storage model (DSM) [CK85], where each column of a table is stored separately as a fixed-width dense array [ABH+13]. To allow easy reconstruction of the tuples of a relational table, each column record is stored in the same (array) position across all columns of a table [ABH+13]. Column-stores typically support a fixed set of basic data types, including integers, fixed-, or floating-point numbers, and strings. For fixed-width data types (e.g., integer, fixed-, and floating-point), column stores utilize basic arrays of the respective type for the values of a column. However, floating-point numbers are usually mapped to integers [ABH+13] as well. Variable-width data types like strings are generally dictionary encoded and represented as integers, which enables their storage into fixed-width columns, too [ABH+13, BHF09]. In the simplest case, a dictionary consists of the distinct values of a column, sorted by frequency, and each value is represented as its integer position on the dictionary [ABH+13]. Consequently, all base columns consist of a sequence of fixed-width integers.

For an efficient processing of these integer sequences, the column-at-a-time model is heavily applied in such systems [ABH+13, BKM08]. Here, an SQL query is translated into a query execution plan (QEP) consisting of multiple operators. Typical query operators include `select`, `project`, `aggregate`, `join`, `group-by`, and `set`-operations featuring a mixture of sequential and random memory accesses for reads and writes. Each operator consumes one or two input columns and produces an output column called intermediate. The column-at-a-time processing model explicitly materializes these intermediates, because each operator within a QEP is evaluated to completion over its entire input, before subsequent data-dependent operators are invoked. Such intermediates are volatile columns and thrown away during or right after the query execution. Thus, the application distinguishes between persistent *base data* and ephemeral *intermediates* in this system model. However, the DBMS still has the flexibility to place any column either to DRAM or NVRAM arbitrary, thanks to hybrid memory concept. Nevertheless, on scale-up platforms such placement is expected to follow the NMP paradigm (cf. Section 2.1.2).

2.2 LOW LEVEL PERFORMANCE EVALUATION OF NVRAM

Because of the novelty of NVRAM devices within modern hardware landscape – it is important to acquire a good understanding of its performance characteristics to make a foundation for the efficient scale-up system level integration. This section, therefore, thoroughly examines basic socket-local and -remote behavior of persistent memory and compares it with the volatile counterpart for clarity.

Hardware Setup. All experiments in this section are executed on a dual-socket system featuring Intel Xeon Scalable (codenamed Cascade Lake) processors clocked at 2600 MHz, 384 GiB DDR4 DRAM memory and 1.5 TiB Intel Optane DC Persistent Memory. Each processor has 24 physical cores (48 w/ HyperThreading). The PMMs are plugged into the system in accordance to the ”2-2-2” scheme meaning that each of the three channels of an integrated memory controller (IMC) is attached to a DRAM and NVRAM memory module as shown in Figure 2.2. Each processor features two IMCs and thus all 6 channels can be operated in an interleaved mode to achieve the maximum NVRAM and DRAM bandwidth. As operating system this experimental setup runs a Fedora 27 with kernel version 4.15. While the binaries are compiled using g++ 7.1.3 with enabled "-O3" optimization flag.

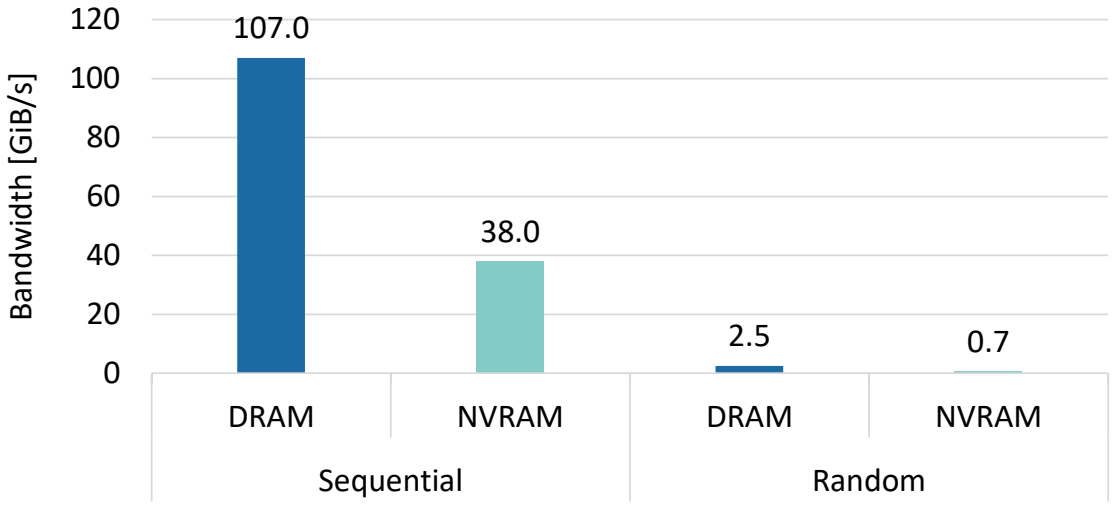


Figure 2.6: DRAM and NVRAM read bandwidth for a sequential and random (8 Bytes) access pattern. The results show the maximum bandwidth measured by multiple threads in parallel.

2.2.1 Socket-local parameters

Local Read Bandwidth. Figure 2.6 shows the maximum read bandwidth achieved by multiple threads (count of 48) on the local socket using a sequential and a random (8 Bytes accesses) memory access pattern each for DRAM and NVRAM. It is observed that the sequential as well as random access DRAM bandwidth is about $3x$ higher compared to the NVRAM bandwidth. Moreover, workloads using a random access pattern with an 8 Byte access granularity (amount of data accessed in a single read) are only able to achieve 2 % of the sequential access bandwidth in the NVRAM as well as DRAM case. The cause of this huge difference is that (i) hardware threads face the full memory latency between the accesses, because the next cache line can not be prefetched, (ii) the internal block-size for data transfers between CPU and memory is an entire cache line (64 Bytes) for DRAM as well as NVRAM, and (iii) the *adjacent cache line prefetch* feature automatically loads the subsequent cache line resulting in a 128 Bytes (two cache lines) data transfer for an 8 Bytes access.

Those effects are further demonstrated in the measurements shown in Figure 2.7. This experiment varies the access granularity for cache line-aligned random accesses (48 threads) on DRAM and NVRAM. The measurements reveal that the effective bandwidth scales up with an increasing access granularity until the cache line size of 64 Bytes is reached. Afterwards, a certain scalability is still given due to the adjacent cache line prefetching until the bandwidth of both DRAM and NVRAM start to converge to their maximal values. Figure 2.8 shows the same experiment for larger access granularities and reveals that hardware prefetchers recognize a sequential access pattern at a 2 KiB access granularity allowing the bandwidth to scale up to its maximum for both memory technologies.

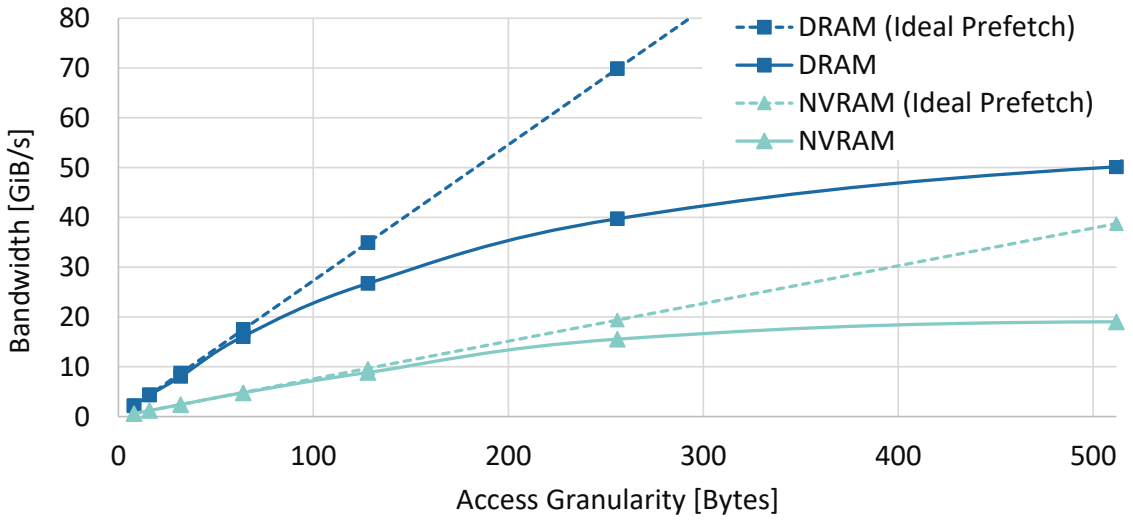


Figure 2.7: DRAM and NVRAM bandwidth for cache line-aligned random memory accesses using access granularities from 8 Bytes to 512 Bytes.

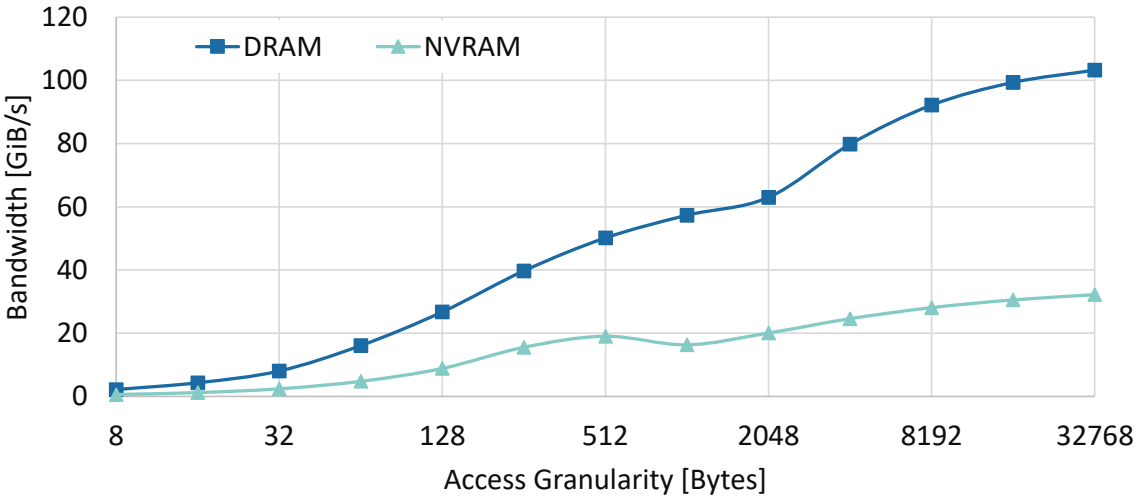


Figure 2.8: DRAM and NVRAM bandwidth for cache line-aligned random memory accesses using access granularities from 8 Bytes to 32 KiB.

Local Write Bandwidth. Figure 2.9 shows the write bandwidth for a sequential and random (8 Bytes) access pattern for DRAM and NVRAM measurements. Similar to the read workload, the peak bandwidth for both memory types is reached using a multi-threaded sequential access pattern (with counts of 48 and 24 threads for DRAM and NVRAM workloads, respectively). While the DRAM write throughput is about 75 % of the peak read throughput, this asymmetry mounts up to a $5x$ lower write bandwidth on NVRAM compared to the read measurements. Interestingly, this asymmetry stays the same for random access patterns on DRAM, but improves from 20 % to 42 % on NVRAM. Nevertheless, the writing random access workload only reaches 2.5 % (DRAM) and 5 % (NVRAM) of the sequential writing bandwidth, respectively. With regard to single-threaded performance, only a few percent difference is measured between DRAM and NVRAM.

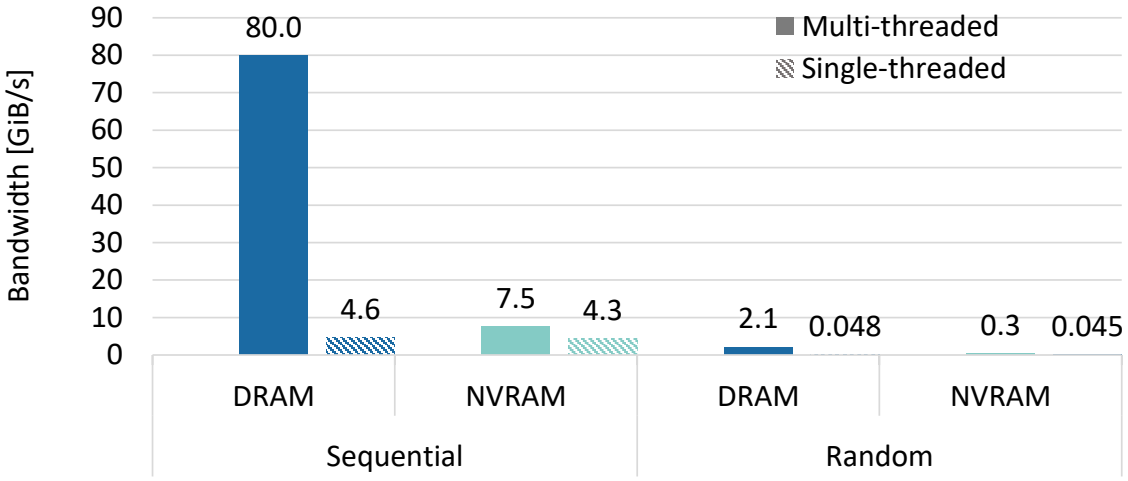


Figure 2.9: DRAM and NVRAM write bandwidth for a sequential and random (8 Bytes) access pattern. The measurements are given for single-threaded and multi-threaded (best number of threads) executions on the local socket.

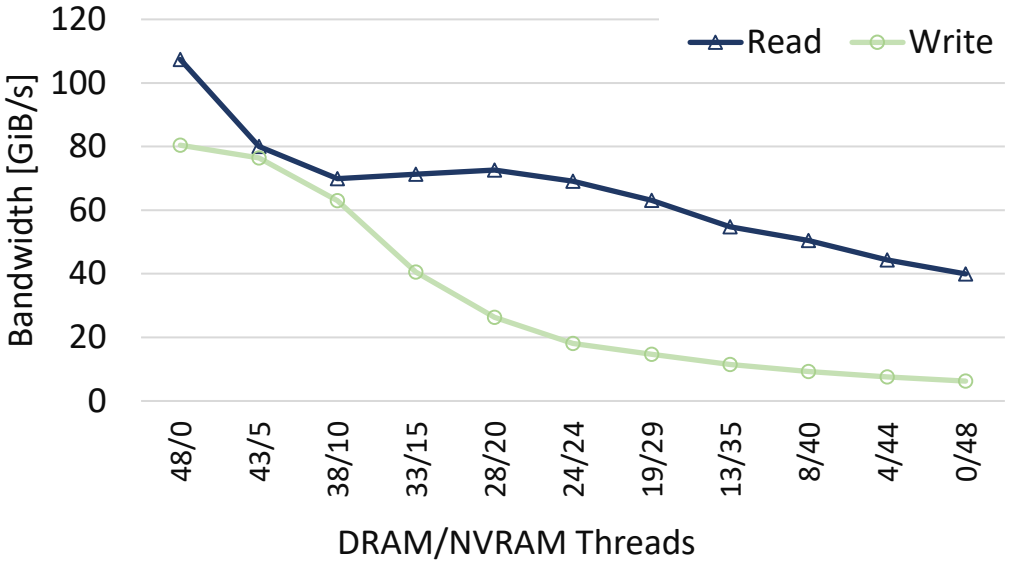


Figure 2.10: Shared DRAM and NVRAM read and write bandwidth for a sequential access pattern.

Concurrent DRAM-NVRAM Access. To figure out whether the two memory types can be used simultaneously with the same maximum performance, the measurements reflected for the 48-threaded case by Figure 2.10 are performed. This experiment revealed that concurrent accesses to both memories are not able to exceed the maximum reachable bandwidth (independently of the NVRAM thread count) of stand-alone DRAM workloads and actually hurt the performance of both involved components compared to separate results. However, this is expected since both medium types in hybrid platform share the same memory controllers and channels (cf. Figure 2.2).

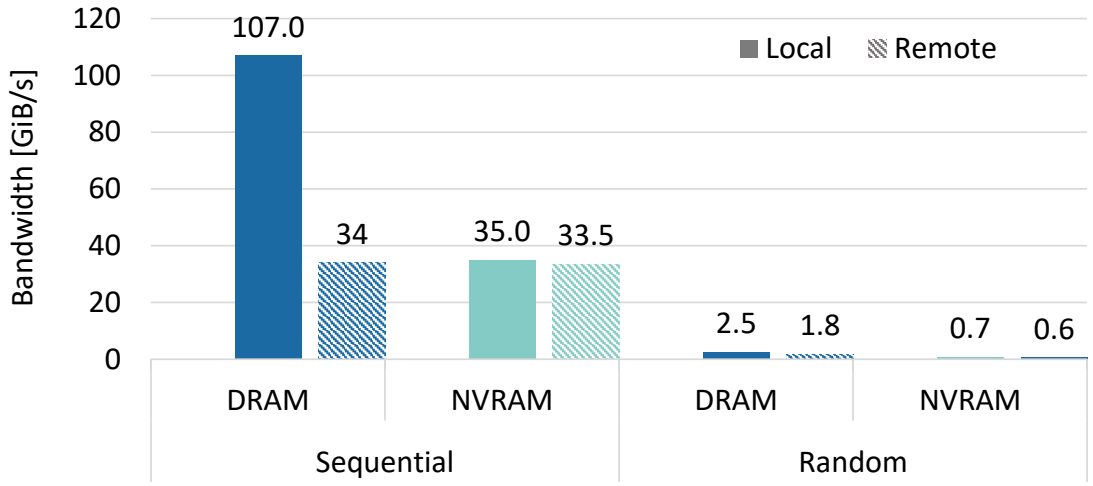


Figure 2.11: Local and remote read bandwidth for sequential and random (8 Bytes) DRAM and NVRAM accesses. The results show the maximum bandwidth measured by executing multiple threads in parallel.

Conclusions. From the read and write bandwidth experiments it could be concluded that NVRAM is $3x$ (read) to $10x$ (write) slower compared to DRAM and exhibits a high read-write asymmetry of up to $5x$. Moreover, NVRAM is accessed at the granularity of a cache line – or two cache lines with activated adjacent cache line prefetching – similar to DRAM and prefers a sequential access pattern. Interestingly, single-threaded measurements show only a small difference in case of writes. From the concurrent DRAM-NVRAM access experiments it could be inferred that intensive simultaneous usage of both memories is not favored by the shared IMCs.

2.2.2 Socket-remote parameters

As already discussed in Section 2.1.2 – non-uniform memory access is the common case in current scale-up server systems usually consisting of 2, 4, 8, or (in certain cases) even more sockets. The NUMA sockets within such hardware architectures are usually connect via an interconnect network (e.g., Intel Ultra Path Interconnect (UPI) [Int09]) that allows access to remote DRAM and NVRAM and ensures cache coherency. However, accessing remote memory induces certain costs in terms of bandwidth limitation and increased latency, which has already been heavily researched within the context of in-memory database systems [KKS+14a, Kim15]. Hence, this section extends the DRAM-NVRAM comparison to NUMA aspects, since UPI performance impacts the data access behavior in case of socket-remote memory allocation.

Remote Bandwidth and Latency. Figure 2.11 shows the read bandwidth for local and remote memory accesses on the evaluation platform. The measurements are conducted for sequential and random access patterns as well as for DRAM and NVRAM memories. The sequential bandwidth is mainly bound by the UPI interconnect bandwidth of the system. While the sequential remote DRAM accesses face a high throughput penalty, the NVRAM and UPI bandwidth is balanced and almost no throughput loss is experienced here. Moreover, the UPI link is full-duplex such that the full NVRAM bandwidth

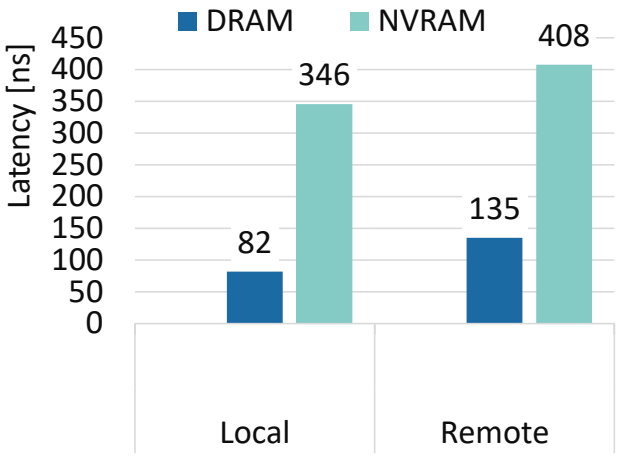


Figure 2.12: Local and remote random read latencies for DRAM and NVRAM. Results are obtained using the `Intel Memory Latency Checker`.

can be achieved for both sockets accessing remote memory simultaneously. Those results suggest that it makes no difference on which socket data is placed, which is not the case because (i) local memory controllers need to be evenly utilized to achieve the full bandwidth of the system and (ii) occupying the UPI link significantly increases the latency of reading or writing random remote accesses. While sequential remote accesses are bandwidth-bound, random accesses are bound by the latency of the UPI link. The latency overhead is visualized in Figure 2.12 showing the local and remote latencies for accessing DRAM and NVRAM, respectively. The measurements reveal that the relative latency penalty is higher for DRAM (about 60 %) than for NVRAM (about 20 %) as it is reflected in the random memory access bandwidth results.

2.2.3 NVRAM access methods

Since there is a number of alternatives in a way how NVRAM can be accessed by the software (cf. Figure 2.4), this part gives an overview of their respective performance. For completeness the DRAM-backed emulation (additionally considering *SHM* or shared memory access method) of persistent memory is also included. Furthermore, the three major options of persistent flushes are considered (cf. Section 2.1.3). As shown by Figure 2.13, the performance numbers (in terms of throughput) of the access methods do not vary significantly in case of a challenging random column store update workload on DRAM-emulated persistent memory. Here, the column store is represented by a contiguous 1 GiB-sized NVRAM chunk filled with 4 B integers. The real NVRAM hardware features slightly larger throughput gap between *DAXFS* and *DAXDEV*. Nevertheless, further in the thesis the *DAXFS* provider is used as the most flexible and convenient method from the maintenance point of view. The relatively small performance difference between DRAM emulation and real NVRAM of about 24 % is explained by the single-threaded execution of this particular experiment. For that reason the workload is not able to reach the full bandwidth of the NVRAM. Moreover, the executed updates do not depend on each other and thus the DRAM and NVRAM latency respectively, is not the limiting factor here.

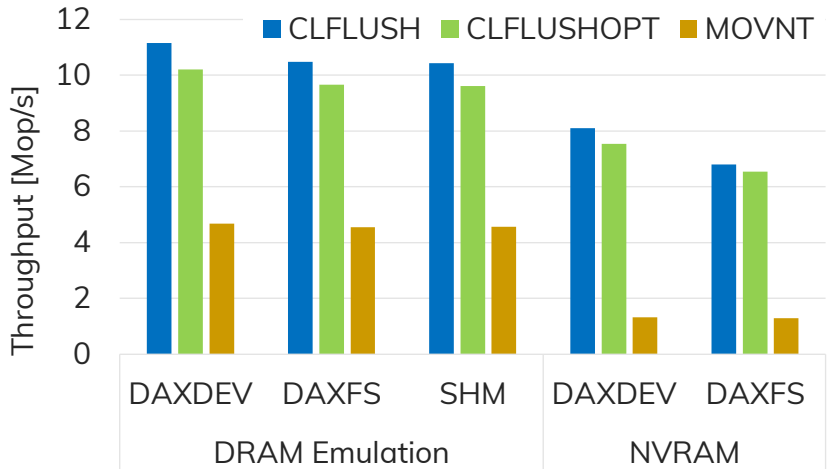


Figure 2.13: Column store update throughput w/o replication for different write back options and access methods each for DRAM-emulated NVRAM and Intel Optane DC NVRAM.

Persistent Flushing. As shown further in Section 3.1, the cache flushing component can heavily affect the data modification behavior. Therefore, this paragraph also compares performance of write back instructions that are available to user mode applications on the test system (Figure 2.13). As mentioned, the random column store update workload is employed for the experiments on both emulated and real persistent memory. Note that `CLWB` is not considered since it behaves the same as `CLFLUSHOPT` on the test platform. As it is observed, `CLFLUSH` and `CLFLUSHOPT` performance numbers are close to each other with a slight advantage for `CLFLUSH`, because it does not require an additional `SFENCE`. The performance of the *MOVNT* is significantly lower, with considerably larger gap in the case of NVRAM accesses.

2.3 VECTORIZED MEMORY ACCESSES IN HYBRID MEMORY SYSTEMS

The optimized CPU instructions designed to directly operate on byte-addressable memory regions are of a particular interest in the field of hybrid memory systems. This is due to the fact that they determine the respective data processing performance and application capabilities. In particular, the advanced vectorization extensions are addressed by this section.

Since decade, computing performance that can be achieved by increasing the clock frequency of a microprocessor is reaching its physical limits thus making the architectural solutions more promising [PRR15]. In accordance to this trend important data parallelism features of Flynn's taxonomy [Fly72] have been added to commodity CPUs – single instruction multiple data (SIMD) extensions. Essentially, these are sets of instructions that can increase an application performance by allowing the basic processing operation to be executed over multiple data elements (i.e., vector) in parallel. Typical SIMD instruction set provides two extensions to the basic instruction set: (i) vector registers which are larger than traditional scalar registers being currently 32 or 64 bits wide, and (ii) tuned vector instructions working on such registers. The vector here is an instruction operand constituted of a set of data elements grouped into a one-dimensional array, while the elements can be normally integer or floating-point values originally stored on medium (e.g., DRAM or NVRAM). Thus, such data-level parallelism is able to transform a vector of elements within single instruction of the processing unit.

2.3.1 Overview of common instructions sets

As already mentioned, in the past years, hardware vendors of common CPU architectures (e.g, x86, ARM) have regularly introduced new SIMD instruction set extensions operating on increasingly wider registers. For instance, Intel's Advanced Vector Extensions (AVX2) operates on 256-bit vector registers and Intel's AVX-512 uses even 512-bit vector registers. Wider vector registers allow processing of more data elements at the same time. For example, an Intel SSE 128-bit vector register can store two 64-bit data elements, an AVX 256-bit vector can store four ($2x$) and AVX-512 512-bit vector can store eight ($4x$) of such data elements. Besides wider vector registers, hardware vendors are regularly introducing more complex vector instructions (tuned for specific operations or even application areas) as well.

Recently, Ungethüm et al. [UPD+20] introduced a specific SIMD abstraction layer called *Template Vector Library (TVL)* for column-stores to tackle the SIMD diversity in a unified way. On the one hand, the *TVL* offers hardware-oblivious vector primitives. On the other hand, the *TVL* also provides an extensible set of hardware-conscious implementations for the hardware-oblivious primitives. Thus, this approach could be used to realize hardware-oblivious vectorized algorithms (e.g., database query operators) based on the provided vector primitives which can be easily mapped to specific hardware-conscious implementations.

2.3.2 Deployment in database scenarios

SIMD (also called vectorization) is a state-of-the-art optimization technique in in-memory databases that is most heavily deployed for columnar data organization and typically applied to isolated query operators [AMF06, PRR15, ZR02, DUP+20, RAB+13, MPM17, LMF+16]. Many vectorized implementations for joins [BTAÖ15, BLP11] and sorting [PR14] have been proposed. Moreover, linear access operators such as scans [WPB+09] and integer compression techniques [AMF06, LB15] are well-investigated. Thus, some query engines are being even fully vectorized [PR20]. Normally, these systems focus on a single SIMD extension (usually AVX-512), i.e., and do not compare efectiveness of alternatives and do not decide at runtime which extension to use.

Since thread-concurrent execution is a natural feature of the hybrid memory system equipped with multi-core computational units – it is important to understand how SIMD vectorized behavoir of database queries is affected by such *multiple instruction multiple data (MIMD)* parallel setting. There is a known issue called *downclocking* [GBB20] that appears in case when large SIMD registers are used by concurrent threads and may lead to the overall performance degradation. To mitigate this problem some works propose separating threads employing AVX-512 and from those executing only scalar instructions by scheduling them on different physical cores to limit the slow-down incurred by AVX-512 on concurrent scalar code. Kumar et al. [KMG14] propose to de-vectorize short vectorized code sections using JIT compilation techniques to avoid the negative impact on scalar code.

From hybrid memory databases point of view SIMD vectorization offers a great opportunity to optimize not only query processing on DRAM-resident data, but also operations involving persistent memory (e.g., replication of primary data) as both mediums are directly exposed to the variety of available CPU instructions as shown in Figure 2.3. Thus, further in the thesis particular attention is given to the opportunities of beneficial SIMD deployment for such purposes.

2.4 RELIABILITY IN HYBRID MEMORY SYSTEMS

As already mentioned in the previous sections, the goal of this chapter is to examine NVRAM and respective hybrid memory architectures from database systems point of view. Since the data reliability is one of the key requirements for most data processing applications, this section starts with an overview of failure processing in envisioned hybrid memory database systems (where NVRAM replaces the storage medium). Subsequently, next section gives a survey of reliability techniques that are generally deployed in computing systems, identifies their strengths and weaknesses when used in hybrid memory databases.

2.4.1 Impact on general database failure processing

Traditionally, database management systems classify failures into three main categories depending on where the problem has occurred [GM09]:

1. **Transaction Failure.** The transaction failure appears when it fails to execute or when it reaches a position from where it can not proceed any further. This class covers also failures of multiple transactions. Generally, the reasons for a transaction failure could be logical (occur if a transaction is not able to complete due to some code error or an internal error condition) or system (occur if the DBMS itself terminates an active transaction because the database system is not able to execute it – e.g., because of the transaction abort, deadlock or resource unavailability) errors.
2. **System Crash.** The system failure may happen due to the power loss or other hardware or software failure (e.g., operating system error) which lead to the loss of volatile content or DBMS state. In such crash scenarios the non-volatile storage and, therefore, accommodated base data are assumed not to be corrupted.
3. **Medium Failure.** The medium failure occurs when storage devices are not able to provide requested data or return it in inconsistent state. For instance, if the HDD is used as persistent medium – this failure may occur due to the formation of bad sectors, disk head crash or any other failure, which destroy all or part of disk storage volume.

The failures belonging to these three classes could be tolerated using a set of well-known error prevention and correction techniques when running the "early days" DBMS architecture, illustrated by Figure 2.14-(a). Here, logging and recovery manager[GM09] of DBMS are used to handle the transaction failures, OS means and recovery manager address system crashes, while medium failures are mostly handled using hardware level reliability approaches. However, with the advent of previously discussed hybrid memory database architectures (depicted by Figure 2.14-(b)) the failure processing approaches need a refinement due to specific innovative properties of such systems. Namely, such architectures offer a great opportunity to persistently store and to efficiently process huge amounts of data exclusively in byte-addressable memory without touching any slow block-accessible non-volatile medium. Thus, the recovery log and important DBMS state information could be now placed entirely in persistent memory [ZLL+15]. Then, the problem of system crashes (e.g., due to the power loss) could be largely mitigated on such systems using consistency-aware persistent programming techniques (cf. Section 2.1.3) and logging data structures [ZLL+15]. Since the transactional failures are not affected by the storage medium – traditional solutions still could be employed for hybrid memory architectures to tolerate them as well. Hence, the only failure class that requires further refinement and detailed investigations is the medium failures. As NVRAM devices are byte-addressable and could be physically distributed over several NUMA sockets on targeted platforms – the possible failure scenarios may have unique features (compared to conventional HDD/SSD devices) described in the next section.

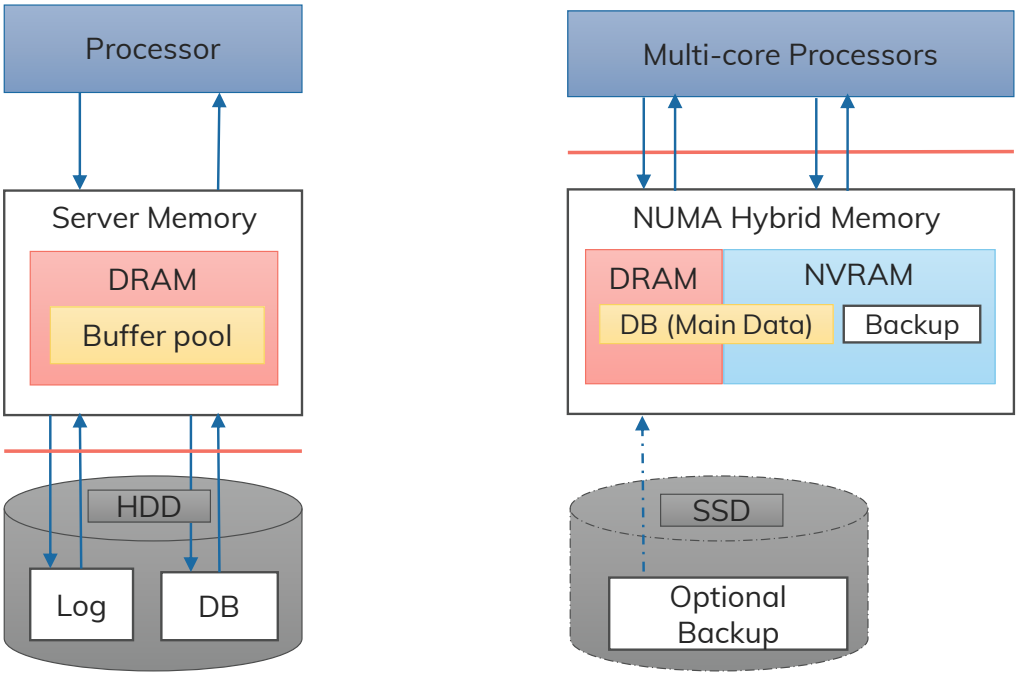


Figure 2.14: Evolution of database architectures.

2.4.2 NVRAM failure scenarios and consequences for the primary data

Actively developed hybrid memory database systems are likely to keep the primary data solely persistent in the NVRAM [ALR+17]. Therefore, efficient data protection mechanisms need to be considered to prevent data losses and to guarantee high availability in case of PMM failures. Such PMM failures may range from single data cell failures to region, chip, or entire PMM failures and can logically be divided in two classes.

Partial PMM Failure. A partial PMM failure occurs if a soft (temporal) or hard (static) error at cell, region, or chip level happens. In such a case, the remaining PMM is still functional and data in this healthy part of the PMM is able to survive the failure.

Full PMM or Socket Failure. A full PMM failure causes the whole NVRAM of the local socket to become inoperative if channel interleaving is configured as it is common to reach the full bandwidth of the NVRAM. In case of multiple PMMs per channel, portions of the local NVRAM are still usable and the error is covered by the previous failure class. If the entire local NVRAM or even the full CPU/IMC fails, this failure still allows to use data placed on a remote socket on the same machine. That may require migration of the execution threads to the cores of remote socket as well. Then system can still work resiliently, while one of the sockets fails.

Although, at this point in time the statistical data on PMM failures is not yet publicly available, the corresponding studies on DRAM DIMMs reliability [SDB+15] and principles of underlying NVRAM technologies [ZLL+15] justify the assumption that NVRAM suffers both from analogous to DRAM issues and non-volatile memory specific problems (e.g., due to high temperatures applied to phase-change memory cells).

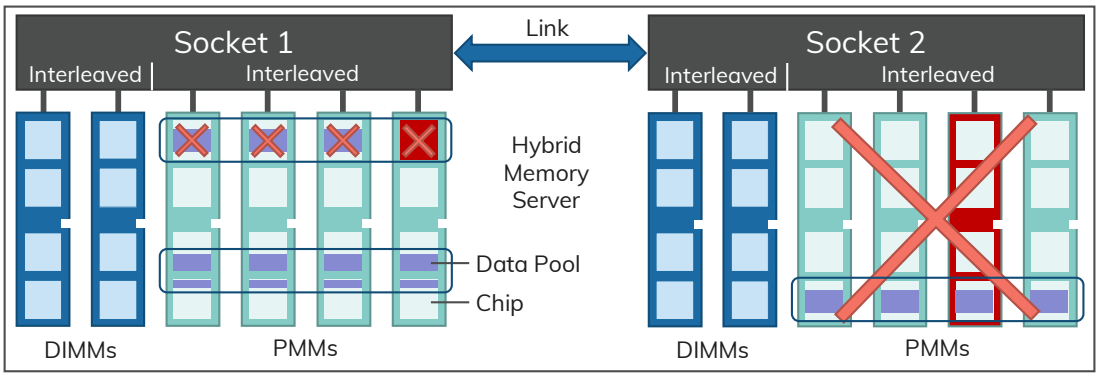


Figure 2.15: Schematic view of a 2-socket hybrid DRAM-NVRAM system. Exemplary visualization of a chip (Socket 1) and full PMM failure (Socket 2) showing the respective consequences for the DBMS.

Figure 2.15 schematically visualizes a hybrid DRAM-NVRAM platform consisting of two sockets, each running the DRAM DIMMs and PMMs, while channel interleaving mode is used for best performance. Hence, the logical non-volatile memory regions are physically stored in an interleaved way across the local PMMs of a single socket. As discussed above, in case of a *partial* PMM failure, e.g., a chip (shown for socket 1), the data of all horizontally adjacent chips becomes unavailable too and in case of a *full* PMM failure (shown for socket 2), the entire NVRAM of the affected socket becomes unavailable.

2.5 SURVEY ON EXISTING TECHNIQUES

The problems of reliability and high availability in computing systems are well-understood and thoroughly studied for a variety of software and hardware architectures. However, the specific properties of hybrid memory systems and discussed above possible NVRAM-induced failure scenarios necessitate a complete rethink of data protection approaches being applied in traditional database architectures. For instance, disk-backed DBMS use block-optimized data replication techniques [Fuj18] that would sacrifice the performance advantage of NVRAM-backed systems. Thus, this section surveys the reliability techniques that are generally used in computing systems and identifies their strengths and weaknesses when used in hybrid memory databases. The related research [ALRL04] suggests the following classification of the reliability means (schematically reflected in Figure 2.16):

Fault Prevention. This approach aims to prevent the occurrence or introduction of failures. The fault prevention is a part of general engineering and is mostly used during the development process of both software (e.g., information hiding, modularization, use of strongly-typed programming languages) and hardware (e.g., design rules). The prevention of development faults is a goal for design methodologies and could be designated to hardware manufacturing side when applied in the area of hybrid memory systems. However, the data reliability issues have to be addressed also during the system functioning and data processing. Thus, means of this approach could not be selected to fully provide the data reliability within challenging field of hybrid memory databases.

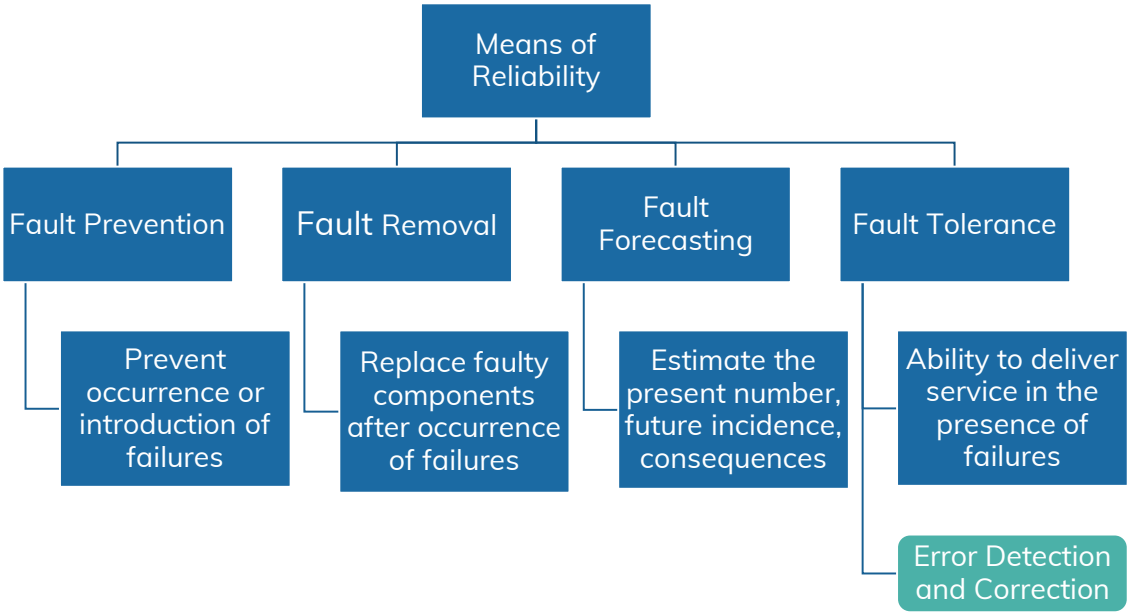


Figure 2.16: Means of reliability deployed in computing systems.

Fault Removal. This mean not only targets the problems that can appear during the development, but also during the use of a system – via corrective or preventive maintenance. The first type of maintenance aims to remove the faulty components that have been detected, while the goal of the preventive maintenance is to remove faults before they might cause errors during the normal operation. Since the interruption of the service delivery even for short time may be unacceptable for many types of databases – both techniques are not applicable when addressing NVRAM failures. Precisely, any type of NVRAM failure (introduced in the previous section) would require a complete replacement of the affected memory module that can lead to the significant downtime of the service (e.g., several hours). Moreover, such replacement of the PMM will not recover the corrupted data making the whole approach oblivious to the primary data losses.

Fault Forecasting. The goal of this technique is to estimate the present number, future incidence and likely consequences of failures. The estimation methods could be subdivided into qualitative (e.g., failure mode and effect analysis), quantitative (e.g., Markov chains and stochastic Petri nets [Zar14]) and mixed (e.g., reliability block diagrams and fault-trees). The stochastic nature of NVRAM failures (e.g., due to the device wear-out or cosmic rays influence) makes the precise fault forecasting almost impossible for hybrid memory systems. And, therefore, such forecasting could only be used to recommend the data placement or reshuffling within NVRAM volume but not actually to fully protect the stored data.

Fault Tolerance. Fault tolerance is an ability of a system to deliver service in the presence of failures. Precisely, this technique is aimed at the failure avoidance and normally is carried out via the error detection and system recovery. The error detection component performs internal checks and maintenance during the system operation. Whilst the recovery counterpart is triggered reactively to the revealed failures with the goal to compensate or remove their impact. Thus, the fault tolerance approach is a most promising candidate to ensure the data reliability in highly-performant hybrid memory database systems, as it is expected to deliver strong data protection guaranties in presence of NVRAM failures without significant system downtimes.

Further, this section gives its focus to the *fault tolerance* reliability approach that is based on online error correction and detection. Essentially, its functioning relies on the redundant system components either compute or storage to ensure protection against respective failures. Since NVRAM replaces the storage in hybrid memory systems - the overview in the following is limited to two traditional storage redundancy techniques: *coding* and *replication*. Both are distinguished based on the underlined principles – former normally transforms, reshuffles or extends the protected data, while the latter is supposed to maintain a certain number of full copies of the same data set. The state-of-the-art implementations of these two flavors applicable for hybrid memory database systems are discussed below.

2.5.1 Hardware coding

The hardware coding is a reliability mean applied in *error correction code memory* (ECC memory). This is a kind of volatile digital data storage that deploys an error correction codes (ECC) to detect and correct n-bit data corruption which may occur in the memory [YMC+11]. Most of the modern DRAM DIMMs belong to ECC memory class that is capable to detect double error and correct single error (abbreviated as SECDEC) [CLX+13]. Thus, such devices are mounted in most machines where the data corruption cannot be tolerated under any circumstances, including critical in-memory databases. However, it is important to concern the respective correction limits regarding the amount of damaged memory cells. As mentioned above, the basic ECC-enabled DRAM DIMMs guarantee the single-bit error tolerance. This means that the data retrieved from each memory word is always the same as data that had been stored, even in case of a single cell corruption (e.g., stuck at 0 or 1 values) or a bit-flip [HKHL18]. There also exists non-ECC memory that uses bit parity checks to support the error detection but not correction, making it necessary to deploy additional reliability mechanisms to actually protect the stored information. Both, the ECC and parity approaches rely on extra memory bits and memory controllers that maintain those bits. Therefore, the redundant bits are used to store the error-correcting code or parity. The parity enables guaranteed detection of all single-cell errors and potentially can discover up to any odd number of bit-flips. While the most frequently used error correcting scheme, the SECDEC Hamming code [CLX+13], allows a single-bit error to be corrected and double-bit errors to be detected. To protect the memory modules against more severe failures (e.g., region or chip) some vendors provide uncommon *Chipkill ECC* – typically more reliable approach that is capable of correcting multiple bit errors, including the crash of an entire chip. However, that approach sacrifices one of the chips of the module for redundancy purposes.

Discussed in this section coding mechanisms are the valid protection techniques for both volatile and persistent memory devices. In fact, the current Intel Optane DC PMMs already deploy ECC approach, dedicating about 25% of the internal memory for error detection and correction purposes only, which is twice as much as for ECC DRAM DIMMs [Nvr20]. Nonetheless, while ECC memory approach has an advantage of low (or moderate in case of periodic error detection scrubbing) performance overhead due to the hardware-integrated coding operations, it is still not applicable as stand-alone technique in the area of hybrid-memory databases. Regardless of the ECC algorithm deployed by the hardware, the field investigations of DRAM and SSDs [ZS19] have revealed that detectable but uncorrectable media errors appear frequently enough to justify additional protection. Furthermore, due to its nature such coding can only protect the data against the small partial corruptions (in most schemes not more then several bits) making the whole method weak against the large region and full PMM/IMC failure scenarios. Moreover, it lacks the desired for databases flexibility to decide which data or memory region actually needs such a protection. Thus, hardware ECC should not be chosen as an exclusive data reliability mean on the targeted architectures.

2.5.2 Software coding

In contrast to the previously discussed ECC memory this section presents the *software-managed* coding means. Thus, within this approach the respective data transformations are to be maintained by the software side and in most cases do not involve any hardware implemented mechanisms. The software-based ECC approach has the flexibility as its main advantage as on the need any type of error correcting codes could be incorporated. While an ability to freely specify the data regions that have to be protected is preserved [HKHL18, YMC+11]. In the following, this section focuses on the most relevant for hybrid memory systems library called Pangolin [ZS19].

The Pangolin is a recently developed fault-tolerant persistent memory programming supplement. It has the following objectives: crash consistency, protection against medium errors, low storage and performance overhead, fast online detection and recovery from medium errors (of size up to 4KiB). The Pangolin library uses a combination of checksums, micro-buffering, and parity to strengthen the user objects against both NVRAM errors and software-induced corruptions. It applies these reliability primitives to persistent objects of any size and supports automatic online detection of data corruption and recovery while keeping the required storage overhead as small as 1% for gigabyte-sized memory pools of NVRAM. The library is closely integrated with the persistent memory allocator of Intel PMDK [Rud15]. This allocator is pool-based (where the pool corresponds to the memory mapped file into NVRAM region). For each pool it reserves a metadata region with important information (e.g., pool identifier and the offset to a *root object* from which all other objects are reachable) and an area for the update logs. The rest of the pool is a persistent heap which is divided into several *zones*. A zone contains its metadata and a sequence of actual memory *chunks* to store user objects.

Pangolin uses replication for pool and zone metadata records as well as RAID-style parity [YJC+16] for user objects to provide redundancy for corruption recovery. To add such parity codes Pangolin arranges zone chunks in a two-dimensional matrix where each chunk row contains multiple, contiguous chunks. The last chunk row is reserved for parity. While the error detection in user objects is based on a 32-bit checksum codes [KDH15] inserted into the object header. Since calculating object checksums and chunk parities make the consistent update procedure more complex (as all records – object data, checksum, and parity – have to be written at once), library deploys a micro-buffering that creates a shadow copy of a persistent object in DRAM. Only when micro-buffer is updated – the changes are transactionally propagated to persistent store.

As weak sides of this approach the following could be mentioned: (i) Pangolin only guarantees protection of up to 4KiB; (ii) the micro-buffers are not protected during updates; (iii) no control of metadata replica allocation is supported (e.g., it is not possible to place it arbitrary far from each other for protection against large region failures). Therefore, based on the example of Pangolin library, it can be concluded that the state-of-the-art software-managed coding have certain advantages (e.g., protection against multi-kilobyte corruptions), compared to the hardware counterpart. However, it is still weak against several failure scenarios (e.g., large region, chipkill, PMM/IMC faults) that have to be tolerated in the developing field of hybrid memory database systems.

2.5.3 OS coding and replication

This section is devoted to the data protection mechanisms provided on the operating system level. Traditionally, the persistent copy of the base data in DBMS was stored as a file or a group of files placed on slow block-addressable device. Since files are typically maintained through certain *file system (FS)* provided by the operating system – there is an opportunity to incorporate some reliability techniques on that level (e.g., file encoding or redundancy) [XZM+17, XL11]. The novel NVRAM-centric databases, though changing many aspects of the data processing, as discussed in the previous sections, still favor the file-based organization of persistently stored data. The key difference is that NVRAM-resident files are normally memory-mapped and could be written in byte-addressable way via direct access feature (DAX) provided by the OS kernel [MZHS17]. Moreover, according to SNIA persistent memory programming model (cf. Section 2.1.3) such DAX-enabled file system is indispensable for the efficient deployment of NVRAM. Thus, further this section overviews the NOVA-Fortis [XZM+17] as a most remarkable representative of fault-tolerant non-volatile main memory file system based on DAX feature.

From a reliability perspective, there are four key differences between conventional block-based file systems and NVRAM-centric file system. First, the memory controller signals medium errors as non-maskable interrupts rather than error codes from a block driver. Second, aforementioned DAX-style memory mapping allows the file contents to change without informing the file system. Third, increased vulnerability to *scribbles* - errant stores from misbehaving kernel code as the entire file system resides in the kernel's address space. Fourth, the persistent memory is significantly faster compared to block-based devices – making it necessary to re-evaluate the traditional tread-offs between reliability and performance. The design of NOVA-Fortis takes into account these differences and relies on the following features to ensure reliability: snapshots [SN10], replication, checksums [KDH15], and RAID-4 parity protection [YJC+16]. The snapshots represent consistent images of the file system at a moment in time. They are used to facilitate consistent backups without unmounting the file system, that enables protection against severe system failures and the unexpected modification or deletion of files. To protect its metadata NOVA-Fortis keeps two *distantly* allocated copies of each internal data structure (primary and replica) and adds 32-bit cyclic redundancy checksum code to both. The checksums in combination with RAID-4 parity are also adopted to protect file data. These primitives allow for successful toleration of cell, region and in some cases chip level media failures, however neither PMM nor IMC crashes. Another general weakness of the FS-based solutions, is that the DAX-style memory-mapped content is not protected in-between of snapshots as FS code is not allowed to snoop on such *direct access* operations. That contradicts with the strong data consistency requirements of hybrid memory database systems and makes such FS-provided resiliency not fully applicable as a stand-alone mean.

2.5.4 Hardware replication

The hardware approach to modular redundancy is known as *memory mirroring*. Particularly, some modern servers support it in the form of channel mirroring, assuming configuration of available channels of DRAM DIMMs in the mirrored mode [LGS+14]. Such configuration (that could be compared to RAID 1 for block-based storage [Byu10]) handles a redundant image of the memory. Thanks to that it can continue to operate regardless to the presence of sporadic uncorrectable failures of cell, region, chip and single DIMM levels. If a failure occurs within the DIMM of one channel, the memory controller shifts to the paired module without disruption, and the devices can re-synchronize when the reparation is finished. However, such feature requires half of the memory modules to

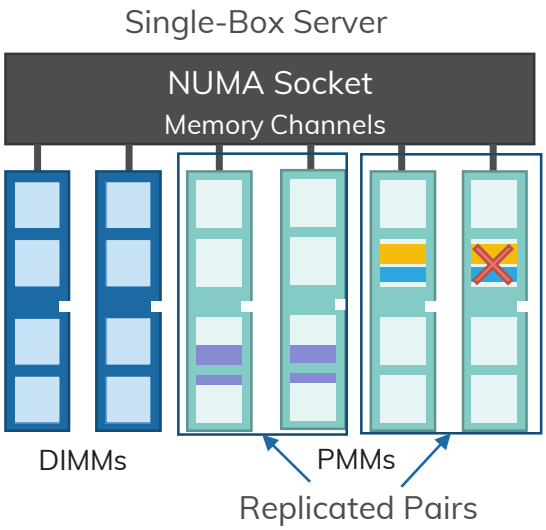


Figure 2.17: Schematic view of a single NUMA socket with mirrored PMMs. Exemplary visualization of a chip failure.

be sacrificed for reliability needs in form of replicated pairs as illustrated in Figure 2.17, as well as the integration of specifically designed memory controller. At the moment, the hardware replication techniques are available for volatile memory only and further limited to two-way modular redundancy within only single socket of a NUMA machine. Thus, while the provided protection may look reasonable in the targeted filed of hybrid memory databases, the costs of hardware implementation could be prohibitively expensive for commodity systems. Moreover, such hardware solutions are still weak against the IMC or Socket failure scenarios and lack desired for in-memory database applications flexibility.

2.5.5 Software logical replication

The software-managed approach to the data replication has two main flavors: logical [Car15] and physical [DS09], though they could be mixed together in certain scenarios. In both cases participating instances are subdivided into masters (or primaries) holding the original data and slaves (backups) accommodated with copies. The key feature of the logical approach (also known as log shipping) is that after updating the primary it ships the log of changes that has to be applied to the replicated slave data sets (this could be done asynchronously to reduce latency [Don15]). Subsequently, such log is replayed by each involved replica to actually produce a duplicated consistent copy of the data. There exists a number of log shipping protocols that vary based on the freshness and safety guarantees [WJP17]. This section considers the state-of-the-art logical replication approach for persistent memory databases exemplified with Query Fresh library [WJP17].

The library is based on ERMIA [KWJP16], an open-source main-memory database engine designed for modern hardware. The targeted application area of Query Fresh is hot standby systems. The hot standby is a common approach to high availability and is deployed by many modern database systems [LLRP14]. A typical hot standby system consists of a single primary server that executes read/write transactions, and one or more backup systems as illustrated in Figure 2.18. The primary periodically ships the log

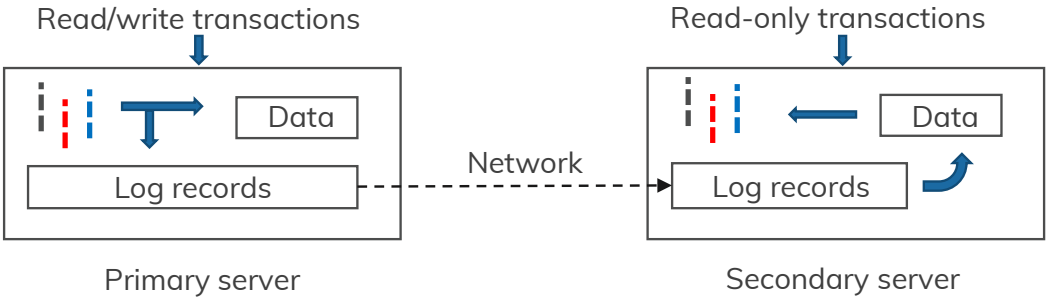


Figure 2.18: Schematic view of a hot standby system.

records to slaves, which continuously replay the respective entries. As mentioned, a log shipping protocol can be synchronous to guarantee strong safety – transactions are not committed until their log records are persisted in all nodes. Thus, the synchronous log shipping approach guaranties that if the primary fails (e.g., due to the server crash), a backup can continue as the new primary preserving committed updates. In addition to replaying log records, backups can serve read-only transactions, improving the resource utilization and performance. The asynchronous mode is typically faster as it trades consistency to performance.

Query Fresh provides both safety and freshness guaranties while maintaining high performance on the primary server. Its design relies on an append-only storage architecture used in combination with fast networks interconnects (e.g., RDMA [ZYSK19]) and byte-addressable persistent memory. This library avoids the dual-copy scheme and treats the log as the database (e.g., data could be extracted from the slave logs without reconstruction phase), that provides an opportunity for lightweight log replay that does not block the primary. The performance advantage is enabled by the NVRAM-resident append-only storage in conjunction with an additional indirection layer. Further, the log replay is accelerated via thread-level parallelism.

Generally, from the perspective of hybrid memory databases such software-coordinated logical replication seems to be more attractive compared to all previously considered techniques, as it is able to tolerate all indicated in Section 2.4.2 NVRAM failures, as well as the complete server crash. However, this protection comes at a price of holding second machine (or even several ones) next to the master. Further, the general drawback of log shipping - is the performance overhead, as the data modifications have to be logged, sent over the network and then reconstructed. Moreover, the logical replication typically does not address the correction of medium failures within the single server without potentially slow network-based transfers (as no local replicas are available). Therefore, these reasons limit the applicability of this approach for highly-performant hybrid memory databases on commodity servers – configuration setup primarily addressed in this work.

2.5.6 Software physical replication

The very last technique considered in this section is software-managed physical replication. In contrast to the previous (logical) approach physical counterpart operates on the memory regions. Those regions are being treated as binary objects that allows to directly mirror respective content. Thus, no tedious log recording, transferring and reconstruction are required. Once the data copying is finished – the addressed replica is straight ready to serve for recovery and potentially other purposes. The state-of-the-art

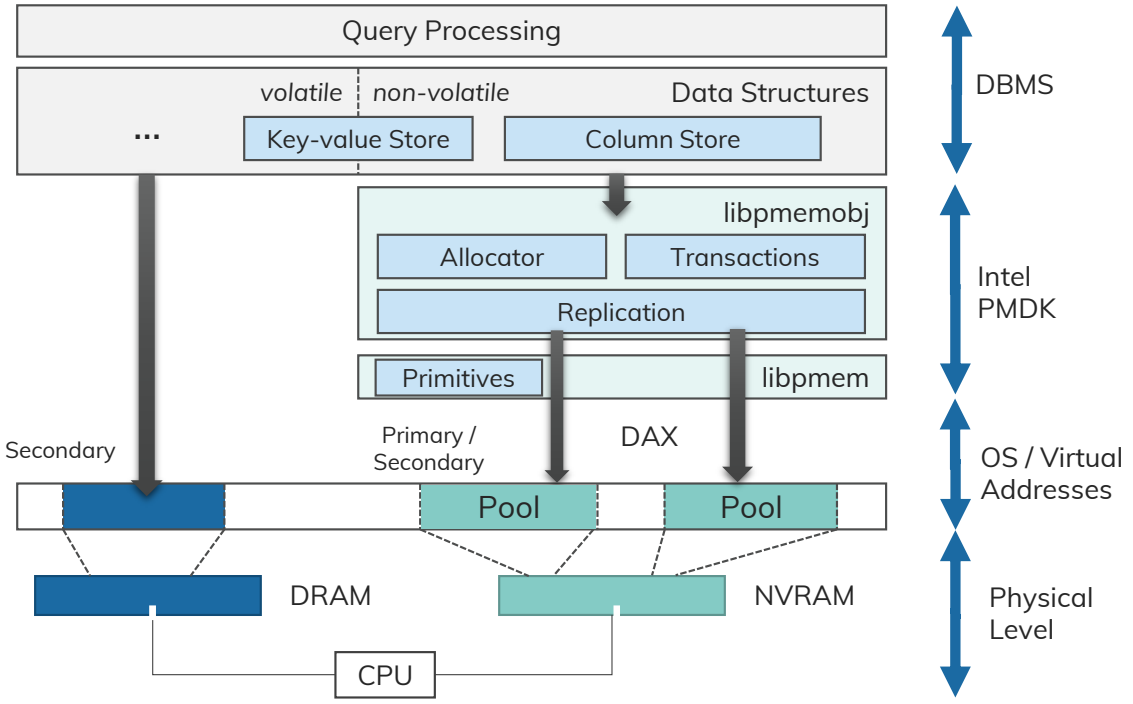


Figure 2.19: Schematic view of a PMDK-integrated hybrid memory database (full-stack).

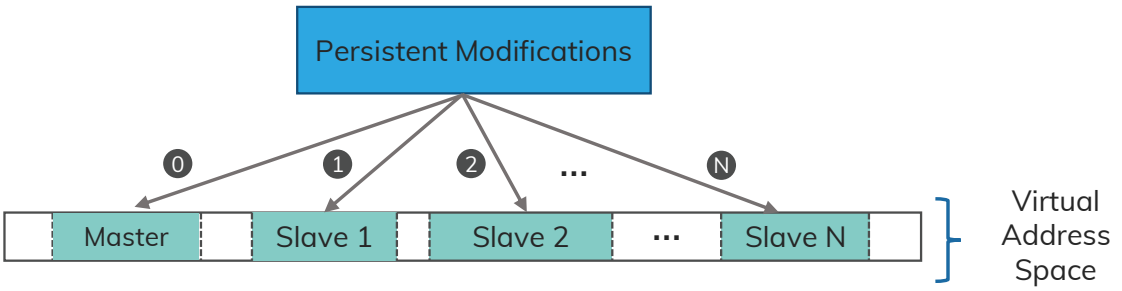


Figure 2.20: Synchronous master-slave replication model deployed by PMDK.

NVRAM-centric implementation of this replication flavor is provided by the Intel Persistent Memory Development Kit (PDMK) [Kap15].

The PMDK [Rud15] is a set of open source libraries that support the development of NVRAM applications. The architectural illustration of the PMDK-integrated database system is provided by Figure 2.19. In particular, the tool-kit provides `libpmem` that is an abstraction layer for platform-specific details such as NVRAM access methods and required instructions for NVRAM programming. `Libpmem` maps physical NVRAM regions into the virtual address space of the database system that are called *pools*. Essentially, such regions are being accessed as memory-mapped files. The next level of abstraction is provided by `libpmemobj`, which primarily adds an allocator to a pool that is able to persistently allocate dynamically sized objects with position-independent virtual addresses. Moreover, `libpmemobj` supports *pool sets* that handle the physical level replication and additionally supports persistent transactions to atomically update multiple memory locations within a pool or pool set. Thus, a DBMS can leverage the PMDK to implement its internal NVRAM-centric data structures that are used for the actual query processing. Whereby the resulting end-to-end workflow could be ordered as following: on physical

level NVRAM and DRAM mediums are directly accessed by the CPU instructions, then on OS level both memory types are distinguished by the virtual addresses (while persistent addresses are specified within NVRAM-resident memory mapped files – pools). Further, if NVRAM pointers are to be processed by the queries – the PMDK library handles the necessary operations (e.g., allocation, transactions). At this level the physical replication is ensured by the *libpmemobj* component. Thus, while performing queries on its NVRAM-centric or hybrid data structures (which place their data in both volatile and persistent memory regions simultaneously [OLN+16]) the DBMS is completely abstracted from the replication operations by the PMDK tool-kit.

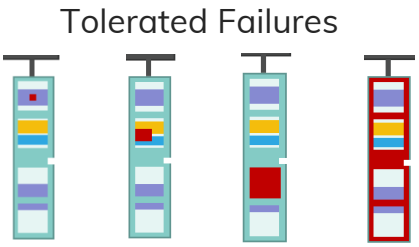
The actual replication algorithm of the PMDK library follows the synchronous master-slave model. Within this approach the replication scheme allocates a single master pool to be primarily accessed for all data processing operations and an arbitrary number of slaves (or followers) to store binary copies of the master. The replication mechanism ensures that all modifying operations are safely done first within master pool, afterwards they are *sequentially* propagated to all involved slaves. Therefore, the algorithm deploys the strongly serialized synchronization and successfully commits only once all replicas are updated as depicted in Figure 2.20. The PMDK scheme supports both individual and grouped or transactional (involving several memory locations within a single update) replication. Additionally, the replica pools could be allocated arbitrary over the NUMA sockets of the server during the pool set configuration step.

From the hybrid memory database system point of view considered above software-managed physical replication has the following advantages: (i) high performance as only coping of binary data is required; (ii) strong consistency guaranties between replicas as propagation follows synchronized algorithm; (iii) ability to tolerate all indicated in Section 2.4.2 NVRAM failure scenarios in case when replica allocation on distinct NUMA sockets is adopted.

2.5.7 Summary

To conclude the survey on the generally applicable reliability techniques to be deployed in hybrid memory database systems, this section presents an illustrative summary table. This table lists strengths and weaknesses of the discussed above six approaches aiming the storage-level fault tolerance (depicted in Figure 2.21). In this table the green tick marks indicate the full applicability of the technique (rows) to tackle the respective failure scenario (columns), yellow O sign assumes either limited usefulness or prohibitively high costs, while the red X means the weakness against the failure.

As seen from the table – all the flavors of coding are weak against the large-scale (e.g., region, chip or PMM/IMC) failures and, thus, are not able to protect the primary data as stand-alone techniques, while the data replication potentially delivers desired level of protection. The most applicable approach here is the software-managed physical replication as it features high performance and full flexibility and does not induce additional economical costs. Thus, this thesis selects the physical option of data replication for the detailed evaluation and investigation in the next chapters.



Technique	Cell	Region	Chip	PMM
Hardware Coding	✓	x	x	x
Software Coding	✓	✓	x	x
OS Coding and Replication	✓	✓	O	x
Hardware Replication	✓	✓	✓	O
Software Logical Replication	✓	O	O	O
Software Physical Replication	✓	✓	✓	✓

Figure 2.21: Strengths and weaknesses of storage reliability techniques when deployed in hybrid memory database systems domain.

2.6 OBSERVATIONS AND CHALLENGES

This section is devoted to the initial evaluation and indication of the drawbacks of the state-of-the-art NVRAM-tuned physical replication mechanism. That helps, subsequently, to draw and motivate a number of important research challenges (further abbreviated as *RC* following respective number) tackled in the thesis. As discussed in the previous section – the basic replication functionality for NVRAM data is provided by the Intel PMDK library. To understand whether the existing approach exhibits an acceptable performance straight out-of-the box – the write intensive experiment on persistent column store with enabled PMDK replication was performed. The column store was selected here as this data structure is frequently deployed to store large amounts of primary data in in-memory designs (cf. Section 2.1.4). The experiment is discussed below.

The uniformly distributed updates were propagated to NVRAM-resident column store (CS) constituted of 4 Byte integers. Each 4 Byte update was explicitly persisted applying `CLFLUSHOPT` instruction for the affected cache line (64 Byte) to ensure strong immediate durability of the modification. While the PMDK replication algorithm was configured to allocate from 1 to 3 replicas on the local NUMA socket. The same hardware platform as in Section 2.2 was used. The respective relative runtime overheads of replication are shown by Figure 2.22. One can see that the cost of replication is unexpectedly high – starting from 334% for one replica and rising up to 1260% of relative runtime overhead for 3 replicas. Such numbers are prohibitively expensive and can vanish the advantage of using NVRAM as high performance storage in the field of hybrid memory databases. Thus, there is a clear motivation for the improvement of the state-of-the-art data replication approach, that is not only concerned with the performance but with several other dimensions as listed bellow.

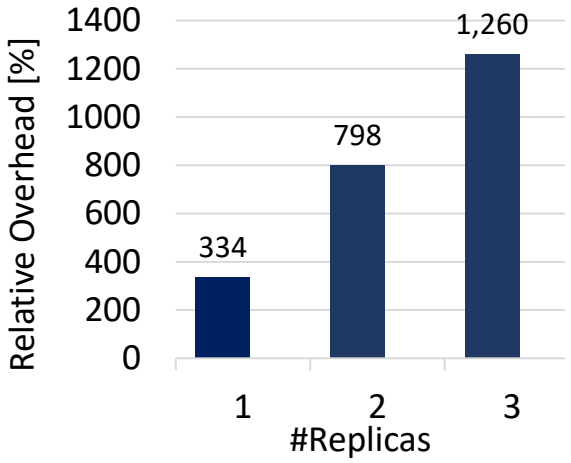


Figure 2.22: Relative *runtime* overhead of basic PMDK replication algorithm. The case of uniformly distributed updates to column store.

RC1: Runtime Overhead Reduction. The first challenge to be addressed in the thesis is the analysis of the reasons of such poor performance (the detailed experiments are discussed in Chapter 3) and the reduction of the respective replication runtime overhead on mainstream servers.

RC2: Space Overhead Reduction. Since the general drawback of the most data replication approaches is the space overhead (every replica normally requires at least 100% of extra storage volume) – the second challenge is the minimization of the additional memory footprint via deployment of data compression techniques. That is particularly reasonable as current NVRAM devices are much less capacitive compared to SSDs as well as writing compressed data implicitly could help to reduce the wear-out of persistent memory.

RC3: NUMA-aware Replication. As already mentioned, to mitigate all possible NVRAM failure scenarios (e.g., PMM or IMC crash) it is important to support the socket-remote replica allocation. Such feature is present in the PMDK tool-kit, however, the replication policy could only be specified offline and at per pool level. Such approach lacks the desired level of flexibility for the DBMS to decide at runtime on the appropriate strategy which data to protect (that could be unnecessary for secondary data) and specify individual replication scheme at data structure level. These issues are identified as third challenge.

RC4: Unified DRAM/NVRAM Replication. The hybrid memory systems achieve their great advantages (cf. Section 2.1.4) via the deployment of both DRAM and NVRAM memory types. However, the current PMDK approach is only capable of NVRAM data protection. In some cases it may be necessary to protect also DRAM-resident information or place some replicas to volatile memory (e.g., to leverage them for query processing needs). Thus, finally, it is desired to derive a concept for more flexible *unified* replication scheme applicable for both DRAM and NVRAM.

RC5: Leverage Replication for Query Processing. Typically, the data replication is used solely for the recovery purposes, unless deployed in the field of distributed database systems [EKA19], where each node holding a replica is also able to process queries. The scale-up systems behave in principle as distributed systems (however, with faster and cache coherent interconnects between sockets) that enables a challenging opportunity to deploy compute node-local replicas also for query processing and efficiently utilize available hardware of distinct NUMA sockets.

REPLICATION - MINIMIZATION OF OVERHEADS

As discussed in the previous chapter – the hybrid memory database systems are likely to keep the *primary data solely persistent in the NVRAM*, as it can be seen in the example of NVRAM adoption in the industrial SAP HANA in-memory DBMS [ALR+17]. Therefore, efficient replication mechanisms (as justified in Section 2.5) need to be considered to prevent data losses and to guarantee high availability in case of NVRAM failures (cf. Section 2.4.2).

Thus, this chapter[1] focuses on a compute node-local synchronous physical NVRAM-backed approach, since other alternatives can not replace it for the underlined persistent memory failure model without identified in Section 2.5 drawbacks. The term "compute node-local" here means replication within a single machine on the local socket and/or across multiple sockets, which also improves data locality on scale-up or non-uniform memory access systems as they are common today (cf. Section 2.1.2). NVRAM replication on a single compute node is vital to avoid data losses in case of PMM failures and is possibly augmented by synchronous or asynchronous disk-based or network-based replication mechanisms [DPVJ14, GZC+16, ZYMS15] to further improve high availability in case of a complete machine failure. Nevertheless, the compute node-local replication is the primary focus in the thesis, because of its relatively low overhead compared to the other replication mechanisms that wipe out the performance advantage of using NVRAM. Moreover, the flexible *software-based* replication approach (cf. Section 2.5) is preferred for further investigations of this chapter which presents compute node-local mechanisms to provide (i) the foundation for an efficient NVRAM replication with a low latency and throughput penalty, (ii) adoption of the lightweight integer compression to ensure the space footprint-minimized replication, and (iii) flexible NUMA-aware allocation to fully protect persistent primary data against all possible NVRAM failures. The evaluation of the proposed in this chapter concepts is embedded in respective sections for better integrity and consistency.

3.1 STATE-OF-THE-ART: EVALUATION AND ANALYSIS OF INTEL PMDK REPLICATION

The considerations of this section are based on the Intel *persistent memory development kit (PMDK)*[2] [Rud15] that already provides NVRAM-centric (essentially optimized for persistent memory) data structures and a basic replication mechanism.

3.1.1 NVRAM-centric data structures

For the detailed experiments of this section, the two NVRAM-centric data structures that are fundamental for in-memory database systems are deployed:

[1] Parts of the material in this chapter have been developed jointly with Thomas Kissinger, Patrick Damme, Dirk Habich, Thomas Willhalm, and Wolfgang Lehner. The chapter is based on [ZKHL18, ZKH+19, ZDK+19], whereby [ZKHL18, ZKH+19] mainly contributed to Section 3.1 and Section 3.2, and [ZDK+19] mainly contributed to Section 3.3. The copyright of [ZKH+19] is held by Springer-Verlag GmbH Germany, part of Springer Nature; the electronic version of the article is available at https://link.springer.com/article/10.1007%2Fs00778-019-00549-w. The copyrights of [ZKHL18] and [ZDK+19] are held by the Association for Computing Machinery (ACM); the original publications are available at https://doi.org/10.1145/3211922.3211931 and https://doi.org/10.1145/3329785.3329923, respectively.

[2] https://github.com/pmem/pmdk

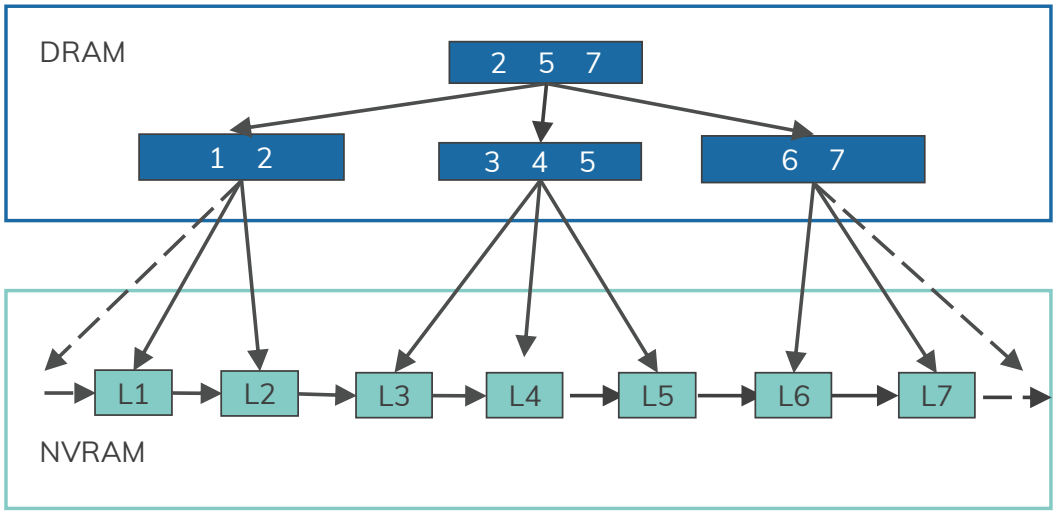


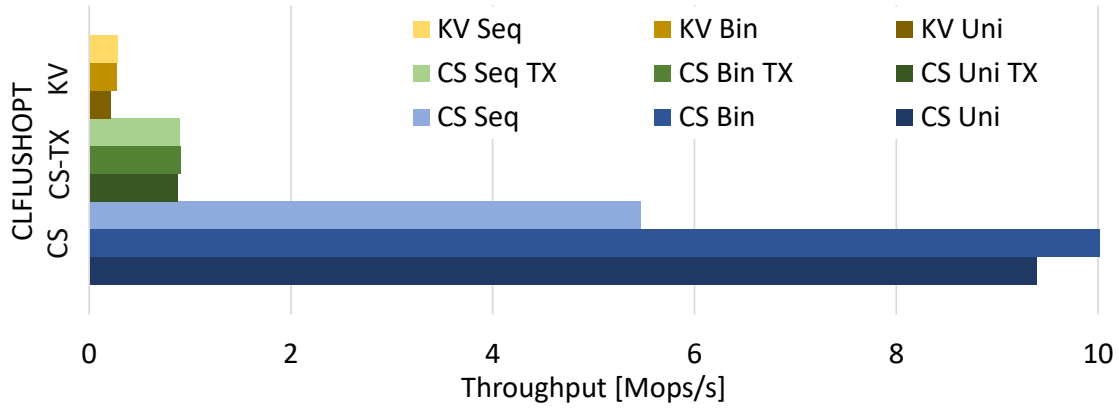
Figure 3.1: Hybrid DRAM-NVRAM key-value store (pmemkv). Leaf nodes are persistently stored as a linked list in the NVRAM and recoverable inner nodes are organized as a B+-Tree in the volatile DRAM.

Column Store (CS) The column store is an array of 4 Byte integers that is allocated via the `libpmemobj` API. Both non-transactional (CS) and transactional workloads (CS TX) are considered for the experiments. The transactional workload also updates a single integer, but uses the transaction feature of the `libpmemobj`, which causes additional writes for the undo log. Both modes are synchronized after each integer update to stress strong durability and consistency requirements.

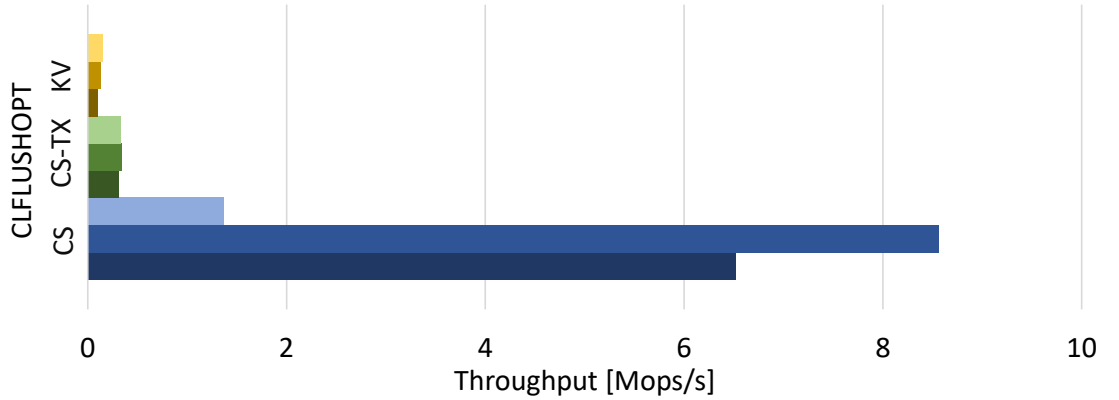
Key-Value Store (KV) Key-value stores are usually employed as indexes within a DBMS. The experiments use a pmemkv implementation[3], which is a hybrid DRAM-NVRAM data structure similar to the FPTree [OLN+16]. As shown in the Figure 3.1, the leaf nodes of the tree are stored in NVRAM in the form of linked list (holding key-value pairs), while the recoverable inner nodes (holding keys only) are stored in DRAM in the form of B+Tree. Pmemkv uses the transactional facilities of `libpmemobj` to ensure atomic updates to the persistent leaf nodes of the tree structure. Therefore, all KV workloads are implicitly transactional.

Evaluation. All experiments in this section are based on the same platform as in Section 2.2 and executed single-threaded on the same socket as the NVRAM regions are physically allocated on. Figure 3.2 shows the throughput for updates on the data structures (w/o replication) for a uniform (Uni), binomial (Bin), and sequential (Seq) key distribution using the `CLFLUSHOPT` instruction. The non-transactional CS workload gives the best performance, because only a single cache line needs to be flushed. In contrast, the CS TX workload uses transactions, which cause additional cache line flushes as it is reflected by the throughput numbers. The worst performance is observed for the KV workload that faces the transactional overhead as well as additional costs for updating to the inner tree nodes in DRAM and may issue modifications to multiple NVRAM cache lines, when modifying the leaf nodes. One can also observe that transactional workloads (CS TX and KV) are relatively insensitive to the key distribution, which is a result of the amortization of cache line eviction and fetching by the other transactional operations. In case of

[3] https://github.com/pmem/pmemkv



(a) DRAM Emulation.



(b) NVRAM.

Figure 3.2: Update throughput w/o replication for different data structures and key distributions using `CLFLUSHOPT`. Measurements are given for DRAM-emulated NVRAM and Intel Optane DC Persistent Memory.

non-transactional workload, the write backs issued by the `CLFLUSHOPT` instruction hurt the performance of the sequential CS workload due to the constant eviction of the cache line required for the next update. The eviction affects as well the binomial CS workload, but in much smaller degree. Because of the *adjacent cache line prefetch* feature, the binomial and uniform key distributions have a chance that the cache line to be updated was already fetched during a previous operation. Nevertheless, this behavior is likely to change with the full implementation of the `CLWB` instruction. The comparison of DRAM (Figure 3.2(a)) and NVRAM (Figure 3.2(b)) shows that the absolute throughput numbers for NVRAM are generally lower than for DRAM and that differences for the key distributions are amplified when working on NVRAM. The reasons for the latter observation are (as could be assumed) the differences in how the memory controller accesses the volatile and persistent memory devices [IYZ+19].

3.1.2 Pool replication

This section is devoted to the basic replication mechanism of the PMDK incorporating a common master-slave replication model. It presents the overhead measurements for

Algorithm 3.1: Basic PMDK pool replication algorithm

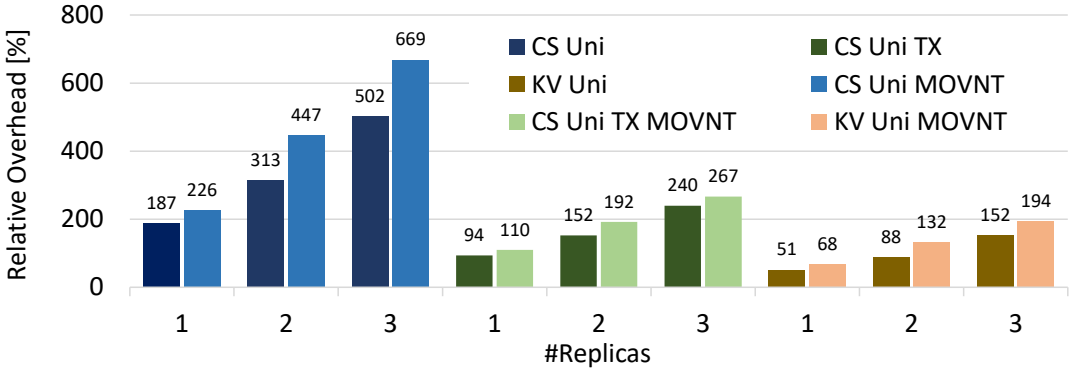
```
Input: Master Pool, Replica Pools, Size, MasterAddress, ReplicaAddress
1: Flush(MasterAddress, Size);
2: SFence();                                          // not for CLFLUSH
3: for each Replica do                                // actual replication
4:     Compute(ReplicaAddress);
5:     MemCpy(ReplicaAddress, MasterAddress, Size);
6:     Flush(ReplicaAddress, Size);                   // not for MOVNT
7:     SFence();                                      // not for CLFLUSH
8: return
```

the three workloads introduced in Section 3.1.1. The runtime overhead numbers are calculated here as *relative* values. Thus, they illustrate additional execution time induced by the replication compared to the non-replicated case as a baseline.

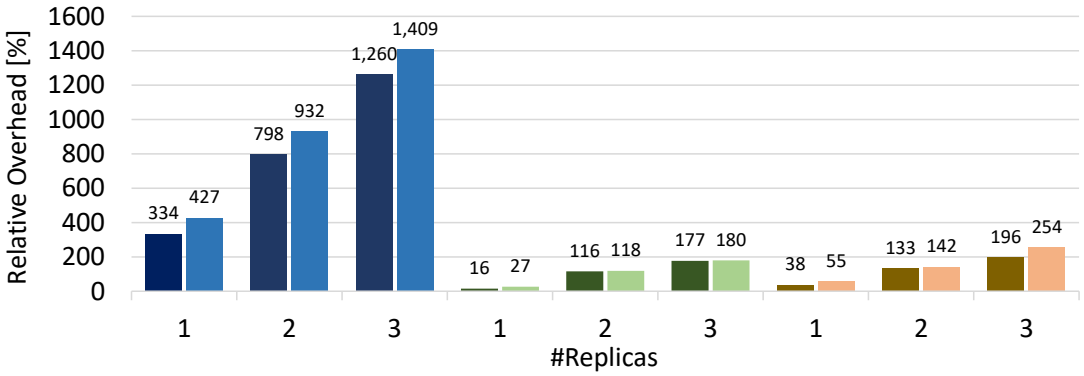
As discussed in Section 2.5, the `libpmemobj` already provides a replication feature that replicates writes to a pool across a set of pools (cf. Figure 2.20). The pool replication also works in combination with the transactional feature where transaction tables and the undo log are additionally replicated across the pool set.

Algorithm 3.1 lists the pseudocode for the compute node-local parts of the replication mechanism. The basic PMDK approach is sequentially-ordered strongly-consistent meaning that all steps need to be processed synchronously one by one for successful execution. The algorithm starts with flushing the modified cache lines of the master replica (line 1) followed by an `SFENCE` to ensure the completion of the flush operations (line 2). Subsequently, the actual replication is performed by looping over the individual replicas (lines 3–7). For each replica, the algorithm computes the memory address of the modification within the replicated pool (line 4); copies the data from the master replica to the calculated replica address either using a `memcpy` or a `MOVNT` (line 5); flushes the modified cache lines of the replica in case of a `memcpy` (line 6) and finally issues an `SFENCE` to ensure the completion of the flushes respectively the `MOVNT` (line 7).

Evaluation. Figure 3.3 shows the overhead measurements relative to the non-replicated throughput for the three workloads (cf. Section 3.1.1) using a uniform key distribution. The overheads are given for the temporal copy (`memcpy`) where replica modifications are flushed using the `CLFLUSHOPT` and the non-temporal copy (`MOVNT`) that requires no additional cache line flush. In both cases, master replica modifications are flushed using the `CLFLUSHOPT`. Each pool of the set has a size of 1 GiB. The replicas are located on the same socket as the master pool. The first general observation is that `MOVNT` copy operations induce a higher overhead compared to the combination of `memcpy` and `CLFLUSHOPT` for both memory types. Furthermore, the measurements reveal that the non-transactional random column store workload (CS) faces about 190 % additional overhead per replica for DRAM-based emulation and about 330 % for Intel Optane DC NVRAM. Note that the updates are executed non-dependent on each other by the benchmark, which is a more realistic setting for writes. Hence, the large pipelines, out-of-order execution, and speculation efforts of modern processors hide the latency for fetching the cache line of the master replica, which speeds up the baseline and thus amplifies the replication overhead. The transactional column store workload (CS TX) faces a replication penalty of about 240 % (DRAM-based emulation) and 180 % (NVRAM) for three replicas, indicating a contrary behavior compared to the non-transactional version. In contrast, the key-value store workload (KV) shows a slowdown of about 150 % (DRAM-based emulation) and



(a) DRAM Emulation



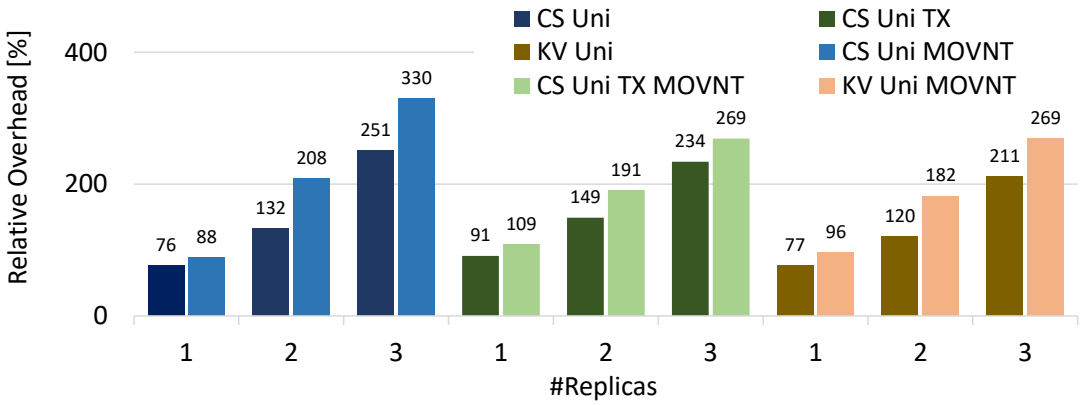
(b) NVRAM

Figure 3.3: Relative update overhead w/ replication compared to the non-replicated workload for the three workloads and 1–3 replicas. All keys are *uniformly* distributed. Additional comparison of `memcpy` + `CLFLUSHOPT` with `MOVNT` only.

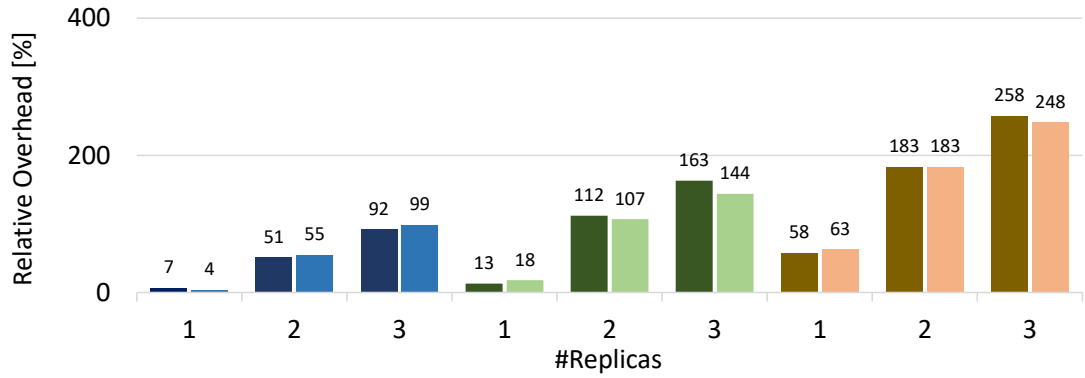
200 % (NVRAM) for the same configurations. Such an increase in relative overhead for KV on real hardware is explained by higher latencies of NVRAM (cf. Section 2.2) since they likely dominate over the volatile operations delays, imposed by the transient part of such data structure.

In case of sequential updates as shown in Figure 3.4, the overhead is within 100 % per replica range for DRAM-emulated NVRAM as it would be expected. However, for the real NVRAM hardware the situation changes considerably. Since there are some differences between DRAM and NVRAM (e.g., bus protocol, buffering, direct access, wear-leveling, prefetching mechanism etc.), it is observed that the costs of a single replica for persistent memory is much lower (just few percent) than for volatile DRAM. Nevertheless, this advantage tends to disappear with an increased replica count and almost vanishes for the KV workload. It is also important to notice that the performance of MOVNT instructions for the sequential workloads does not differ much from that of the `memcpy` and `CLFLUSHOPT` combination.

Conclusions. The overhead measurements revealed that the NVRAM replication costs (first drawn in Section 2.6) of the basic pool replication mechanism are prohibitively high,



(a) DRAM Emulation



(b) NVRAM.

Figure 3.4: Relative update overhead w/ replication compared to the non-replicated workload for the three workloads and 1–3 replicas. All keys are *sequentially* distributed. Additional comparison of `memcpy` + `CLFLUSHOPT` with `MOVNT` only.

especially in case of the CS workload, and will annihilate the advantage of using NVRAM as a storage for primary data. Hence, next section will reason about the origin of this replication penalty and propose optimizations for the basic replication mechanism.

3.2 RUNTIME OVERHEAD REDUCTION THROUGH ADAPTIVE EFFICIENT REPLICATION MECHANISMS

This part addresses the issues of the basic PMDK pool replication mechanism that cause the high overheads during the NVRAM replica maintenance. Afterwards, it presents a set of advanced pool replication mechanisms that are generated using a template-based approach. That approach automatically composes the individual building blocks to a full replication mechanism. Thus, this section is designated to the overall challenge RC1 of the thesis targeting the reduction of the replication-induced runtime overhead (cf. Section 2.6).

3.2.1 Optimization of the basic pool replication algorithm.

The basic pool replication mechanism (cf. Algorithm 3.1) faces the following issues:

1. The algorithm is over-synchronized, because cache line flushes are synchronized via an `SFENCE` instruction after each master and replica update. Such a protection is normally needed to prevent write reorderings. However, in the replication scenario, master and replica updates can be written in parallel and require only a single synchronization point.

2. The master replica is updated first, which evicts the modified data from the cache when using the `CLFLUSH`, `CLFLUSHOPT` or `MOVNT` instruction. Hence, the subsequent copy operation that updates the first replica needs to refetch the data from the memory first. When writing back through `CLWB` instruction the modified data stays in cache and, therefore, no refetch is needed for that case.

3. The `memcpy` operation that copies the modified data from the primary replica to the other replicas effectively performs a read-modify-write operation. Thus, the cache lines of the replicas to be updated need to be present in cache, which is not considered by the algorithm. `MOVNT` copies do not require the data to be in the cache, but are executed faster if the virtual-to-physical address mapping of the destination is already cached by the memory management unit of the CPU. In particular, the translation lookaside buffer (TLB) is responsible for storing such address translations [Kli19].

4. Independent copy operations from the modified primary replica to the other replicas are not parallelized and execute sequentially.

To overcome those issues, the template-based approach for generating advanced replication methods is presented in Figure 3.5. The template describes how the individual building blocks are arrangeable to guarantee persistent updates to all replicas of the pool. These individual blocks are *generally available* on most hardware systems including hybrid memory architectures. The advanced replication method generation starts from the top of the scheme and proceeds downwards by configuring the desired option for a particular building block (depicted in gray color). The respective dependencies between options and building blocks are expressed by arrows. For instance, asynchronous flushes (Flush Async) require a final `SFENCE` instruction and the master replica is either flushed before or after updating the replicas (indicated by the double-headed black arrow on the left hand side). The *Prefetch* building block is optional.

In accordance to this approach, the basic replication algorithm can be generated following a path indicated as "1-2-3" in Figure 3.6. In contrast, a thread-level parallel replication mechanism would be generated by taking the right most path of the template. In the following, this section describes the building blocks in detail and highlight how they cope with the aforementioned issues of the basic replication mechanism.

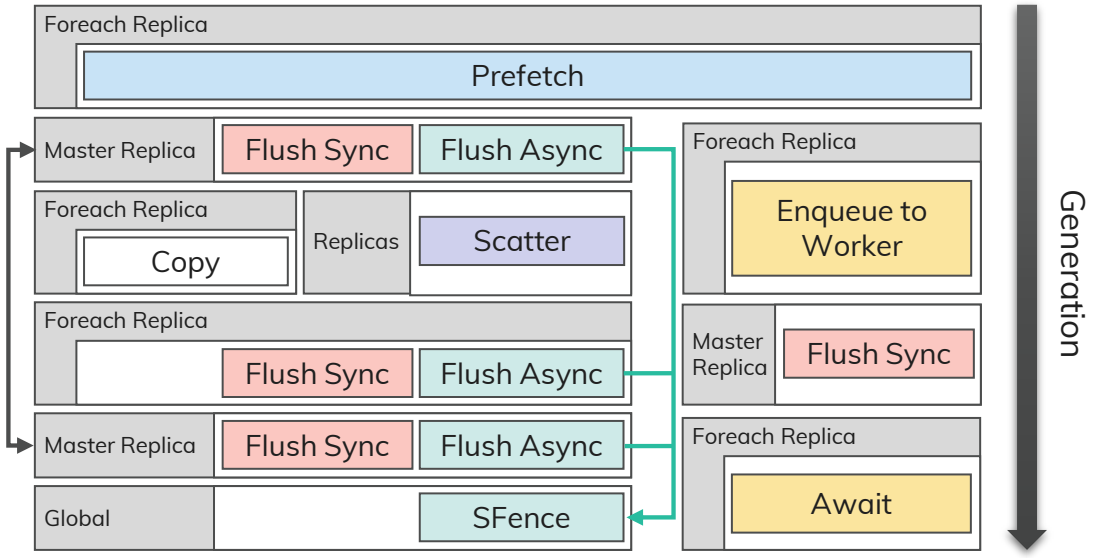


Figure 3.5: Building blocks and composition options for the template-based advanced pool replication. Replication mechanisms are generated by taking a compatible path from the top to the bottom.

Instruction-level Parallelism. To resolve issue (1), the template adds the *Flush Async (SF)* building block based on instruction-level parallelism principle. That type of parallelism attempts to improve processor performance by having multiple processor components or functional units simultaneously executing instructions [Pac11]. This block omits the `SFENCE` as an alternative to *Flush Sync* (Flush followed by an `SFENCE`). Nevertheless, to ensure completion of the NVRAM writes, a single `SFENCE` (SF) is required at the end as indicated by the green arrow in Figure 3.5. Doing so, *Flush Async* is able to execute `CLFLUSHOPT`, `CLWB` and `MOVNT` instructions among replicas in parallel.

Cache Optimization. To overcome issue (2) by preserving the necessary data in the L1–L3 caches, the template for advanced pool replication mechanisms allows to execute the master replica flush before (PRE) or after (POST) the other replicas are written and flushed. This is indicated by the leftmost double-sided arrow in Figure 3.5.

Software Pipelining. To further improve cache utilization and cope with issue (3), it is reasonable to add the *Prefetch (PF)* building block that preloads the first cache line of the memory location for a replica. Consecutive cache lines that may need to be updated are usually automatically loaded by the hardware-based prefetchers, i.e., the adjacent cache line prefetch or sequential access prefetchers. The standard single cache line prefetch call is applied here for the preloading mechanism. The alternative AVX512PF *Scatter-Prefetch* instruction[4] has not shown any possible performance advantage for the purpose of the replication-centric experiments.

Data-level Parallelism. To address issue (4) in case of small updates, we exploit the well-known SIMD approach (allowing simultaneous computations, but only a single processor command at a given moment, cf. Section 2.3). In particular, the algorithm uses AVX512 *Scatter-Store* instruction[5] (denoted in the template as *SCA*) that

[4] `_mm512_mask_prefetch_i64scatter_pd intrinsic`

[5] `_mm512_mask_i64scatter_epi32` or `_mm512_mask_i64scatter_epi64 intrinsic`

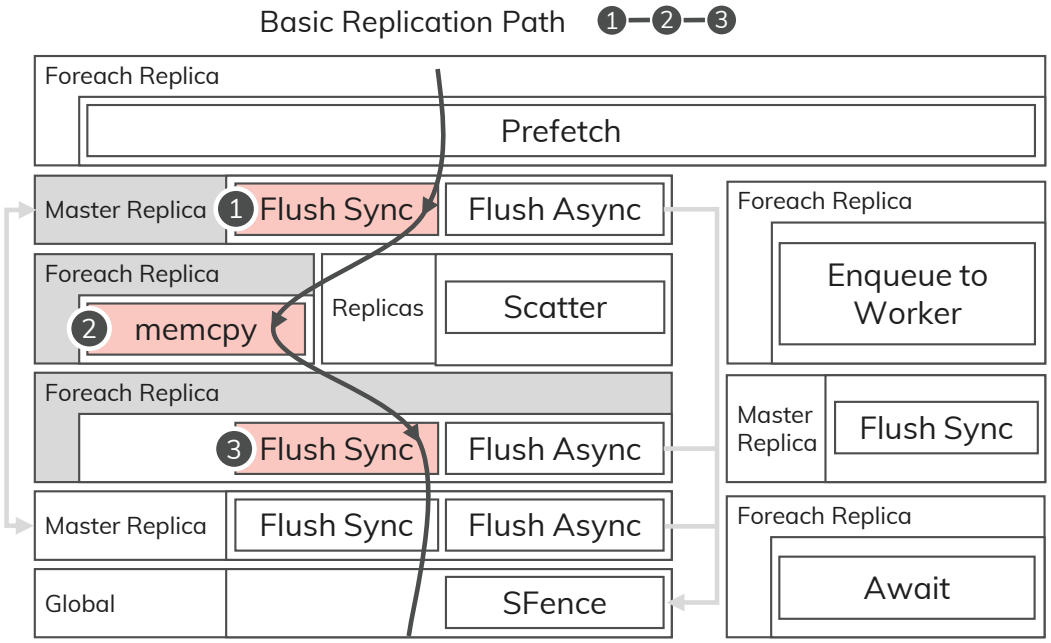


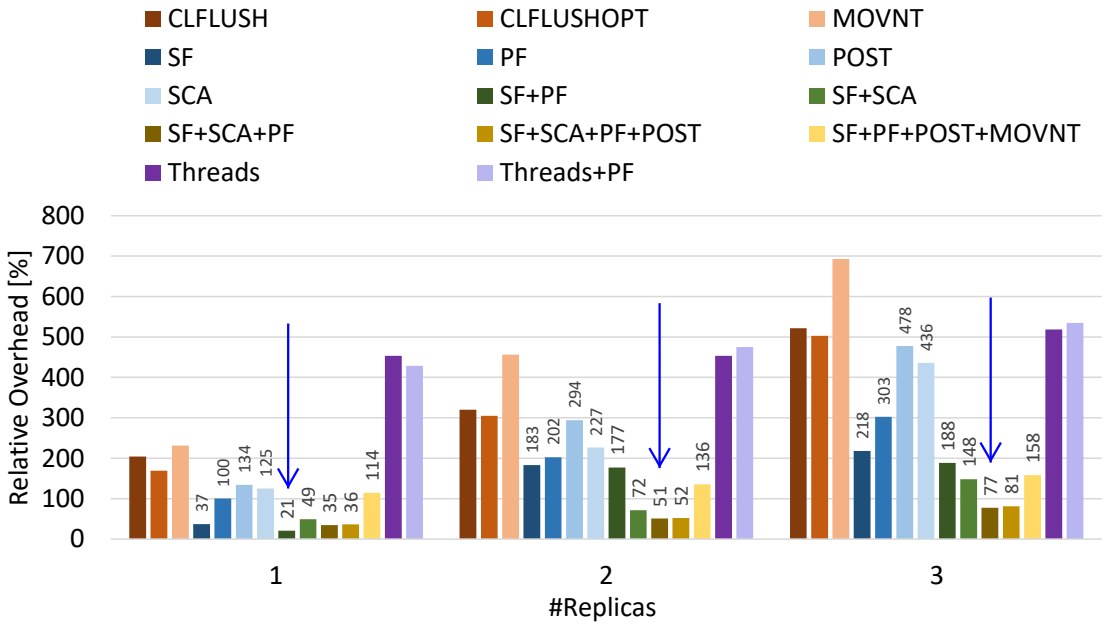
Figure 3.6: Composition of the building blocks of the advanced replication mechanisms template for the basic replication mechanism of the PMDK (cf. Section 3.1.2).

leverages data-level parallelism to update multiple replicas with a single instruction. This instruction is limited to write 8 Bytes updates to eight replicas at a time. In case of larger updates, the instruction needs to be executed multiple times.

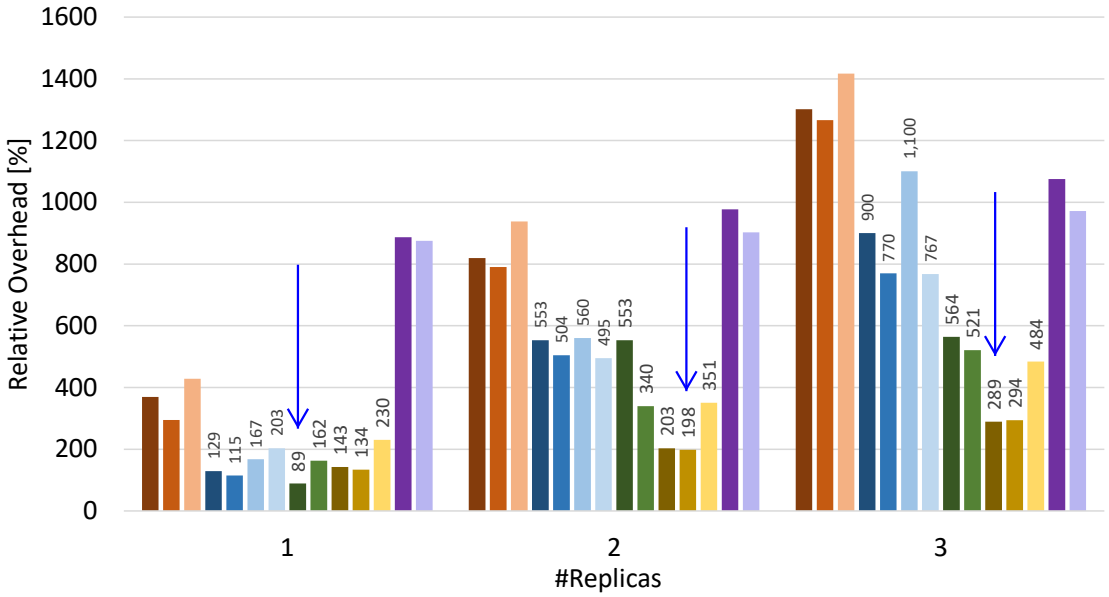
Thread-level Parallelism. To address issue (4) in case of large updates, the algorithm employs thread-level parallelism (Threads) also known as multiple instruction multiple data (MIMD) paradigm to additionally leverage the load-store units (LSU) of other cores. Hence, for each slave replica a worker thread is instantiated during the opening procedure of a pool set that executes the *Copy* and *Flush Sync* building blocks for its replica, if instructed so by the master replica thread. The master replica thread needs to wait for the successful execution of the workers that update the replicas to ensure the completion of the operation.

Note that the *Copy* and *Flush Sync/Async* building blocks can additionally be switched to a different implementation, i.e., `CLFLUSH`, `CLFLUSHOPT` or `CLWB` instructions for flushing and various platform-tuned `MOVNT` instructions for copying.

With regard to *crash-safety* property, it could be argued that since the PMDK undertakes strongly-consistent approach (to commit the modifications *all* replicas have to be successfully updated) suggested advanced algorithms do not reduce the consistency guarantees, compared to the basic scheme, as all of them are synchronized upon completion. With the help of the template-based approach, it is possible to evaluate all combinations of allowed and meaningful optimizations for several workloads and workload parameters. Moreover, the template can be leveraged to automatically switch to the optimal replication mechanism depending on the current conditions as will be discussed next in the chapter.



(a) Non-transactional column store workload (DRAM).



(b) Non-transactional column store workload (NVRAM).

Figure 3.7: Relative overheads for different optimizations and replica counts. The CS workload with *uniform* key distribution. Measurements are given for DRAM-based emulation and NVRAM (Intel Optane DC Persistent Memory). Minimal relative overheads are indicated by the blue arrows.

Evaluation. The following paragraphs provide an evaluation for previously considered workloads of the proposed advanced replication mechanisms. The experiments are executed on both Intel Optane DC and DRAM-emulated persistent memory along with the comparison to the basic replication schemes of the PMDK. The results of non-

transactional column store workload with uniformly distributed keys are primarily analyzed. Moreover, the investigation also addresses the influence of the chunk size (or number of consecutively updated column store elements).

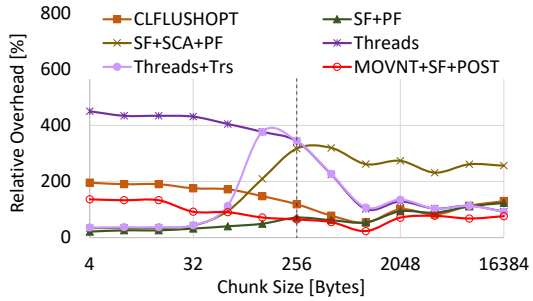
Impact of Optimizations. Figure 3.7 shows the overhead numbers of a selected set of activated optimizations including the PMDK baseline (`CLFLUSHOPT`) for the targeted workload using uniform key distribution for up to three replicas. Each pool is 1 GiB in size.

The measurements show that enabling a single optimization already reduces the overhead by at least one third on both real and emulated NVRAM except for the `POST` optimization that postpones the master replica flush. Activating combinations of optimizations further decreases the replication overhead. For a single replica, the `SF+PF` combination gives the best results. While for more replicas, the `SF+SCA+PF` combination tend to dominate.

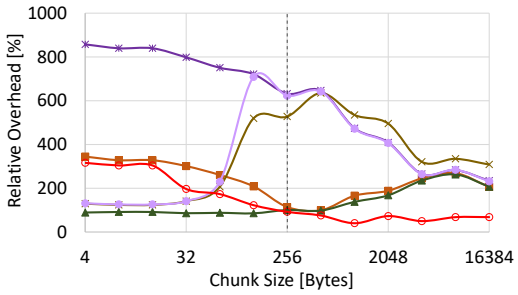
The `Thread` optimization has a high initial overhead, because of the thread maintenance and synchronization costs that start to amortize with an increasing number of replicas. Generally the behavior of this mechanism is similar between both memory types. The performance of the `Threads` and `Threads+PF` replication mechanisms for the replica count of two is comparable to that of the non-optimized mechanism and start outperform it with a further increase of replicas. The prefetch (`PF`) component contributes to 5 %–15 % to the relative overhead reduction. Overall, the `Threads` block is able to save up to 78 % of the relative overhead for the non-transactional CS workload on NVRAM, while on the DRAM-based emulation may benefit up to 85 %, respectively.

The initial comparison of replication overhead measurements shows that suggested optimized replication mechanisms give savings of up to an order of magnitude for both memory types with a tendency for NVRAM, which is actually targeted memory type. Moreover, the optimizations reduce the extra overhead costs for adding an additional replica to the pool set.

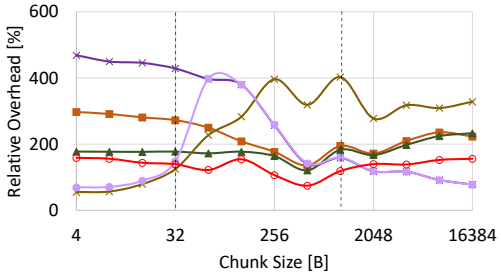
Updated Chunk Size. This series of experiments vary the size of the memory chunk that is updated by writing a varying number of consecutively located column store elements per update operation. The Figure 3.8 visualizes the measured relative overheads for the CS using different optimization combinations depending on the updated chunk size. In general, the results show that the best replication mechanism depends on the updated chunk size as well as on the workload, replica count, and memory technology. For instance Figures 3.8(e), 3.8(f) show (i) that the `SCA` building block generates more overhead for larger chunk sizes, (ii) that the `MOVNT` implementation of the `Copy` building block starts to become feasible at a chunk size of 32 Bytes for both DRAM-based emulation and NVRAM, and (iii) that the `Threads` building block gives the lowest overhead in case of DRAM-based emulation starting at a chunk size of 2 KiB and becomes feasible with a replica count starting from 3, however being outperformed in the same interval by `MOVNT`-based mechanisms on real NVRAM. Since the `Threads` building block is designed for large chunks, it is reasonable to add a `Threshold` building block (`Threads+Trs`) that falls back on the alternative `SF+SCA+PF` path for small chunk sizes ($\leq$ 32 Bytes as visible in Figure 3.8(a)) as they may occur in the transactional part of the respective workloads for logging. Such a combined approach suits for workloads with varied updated chunk size and is a first step towards an automatic replication mechanism switching that will be introduced in the next section.



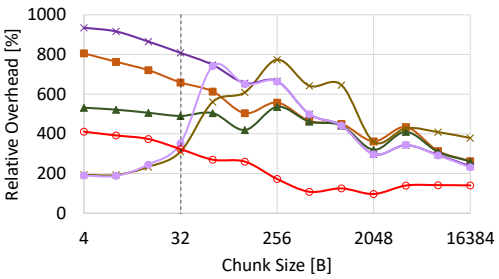
(a) 1 replica (DRAM Emulation).



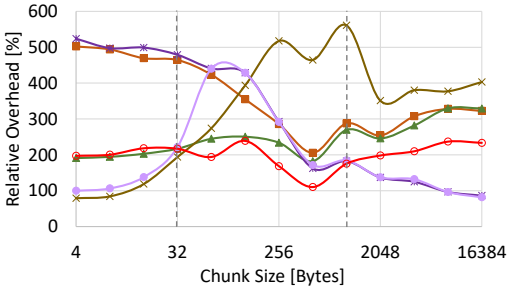
(b) 1 replica (NVRAM).



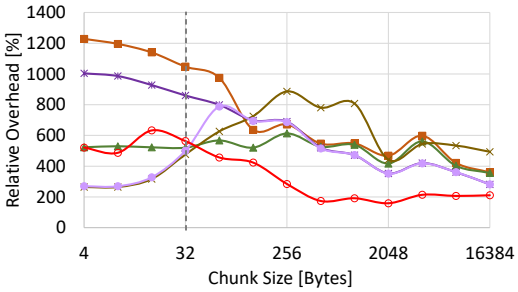
(c) 2 replicas (DRAM Emulation).



(d) 2 replicas (NVRAM).



(e) 3 replicas (DRAM Emulation).



(f) 3 replicas (NVRAM).

Figure 3.8: Relative overheads for the *column store (CS)* workload using a *uniform* key distribution. Measurements are given for different optimized replication mechanisms, replica counts, and memory types as well as the updated chunk sizes. Switching points are highlighted by the dashed lines.

3.2.2 Adaptive lightweight switching algorithm

This section presents an *automated* lightweight algorithm for the online switching between the advanced replication mechanisms discussed in Section 3.2.1. This algorithm activates the selection of the best replication mechanism for the current workload conditions.

Motivation and Requirements. As it is shown in the previous evaluation, the overhead of the different replication mechanisms depends on various factors such as the updated chunk size, replica count, workload type, and memory type. In particular, Figures 3.8 (a)-(f) show that there are certain regions where specific replication mechanisms exhibit a

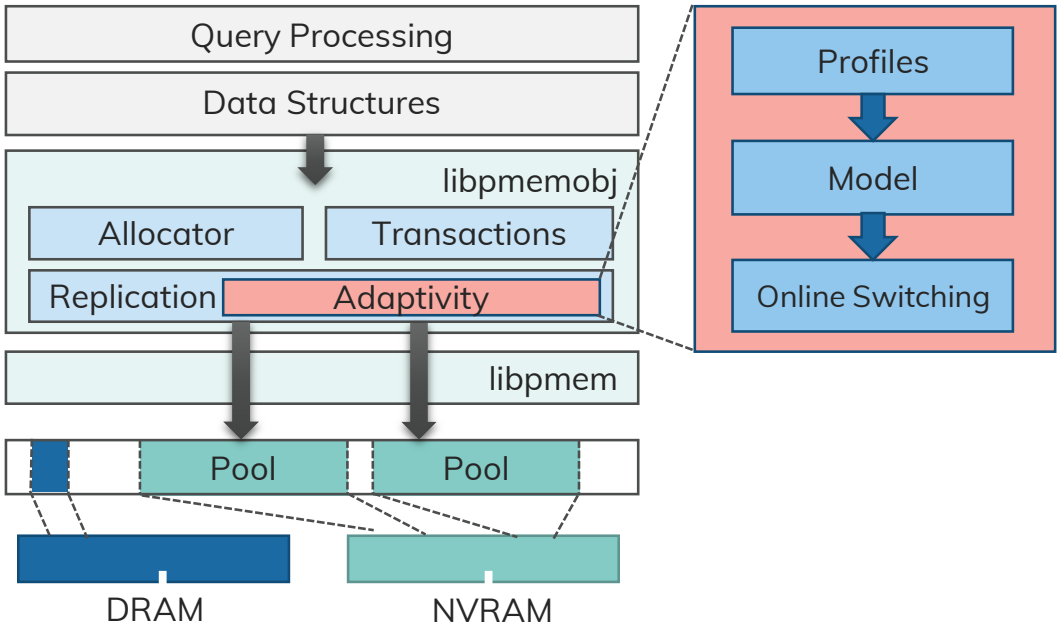


Figure 3.9: Replication mechanism adaptivity using a lightweight online switching algorithm that relies on a switching model. Adaptivity is implemented at the level of `libpmemobj`.

significantly lower overhead compared to the others. The goal of this section is to leverage these observations to design a lightweight switching algorithm that selects the best fitting replication mechanism for the current conditions at runtime. Since all the basic and optimized replication functionality is implemented at the level of `libpmemobj` library of PMDK (cf. Figure 2.20), the selection algorithm has to be integrated using the facilities available at that level. In the following, the details of the three key components of suggested adaptive replication mechanism selection approach (visualized in Figure 3.9) are provided.

Profiles. The profiles characterize the replication mechanisms that are automatically generated based on suggested template approach (cf. Section 3.2.1). Such platform-dependent profiles are generated at deployment time and provide information about the overheads of the individual replication mechanisms in the context of various workload states and pool set configurations. The profiles are the foundation for the switching model. For further considerations, this section stays with the measurements presented in paragraphs above, which provide the algorithm with profile information for the specific test system (whereby the tabular format is adopted to store actual workload characteristics as illustrated in Table 3.1).

Model. Since replication operations are executed within a very tight time scale (sub-millisecond), the switching algorithm needs to be very lightweight to keep its overhead as low as possible. Hence, the full profile information can not be taken into account and a more coarse-grained model is required to allow for fast decisions on the best replication mechanism for the current conditions. Due to this essential requirement, it is justified to to go for a small decision tree as the model for the switching algorithm. The advantages of this choice are the simplicity (as only binary branching is expected for non-terminal nodes) and determinism at any level. The top level of such a tree for the test platform is schematically shown by Figure 3.10.

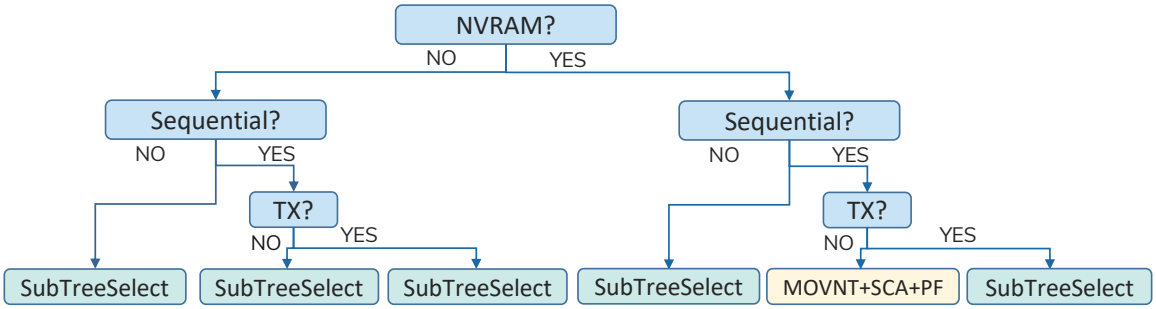


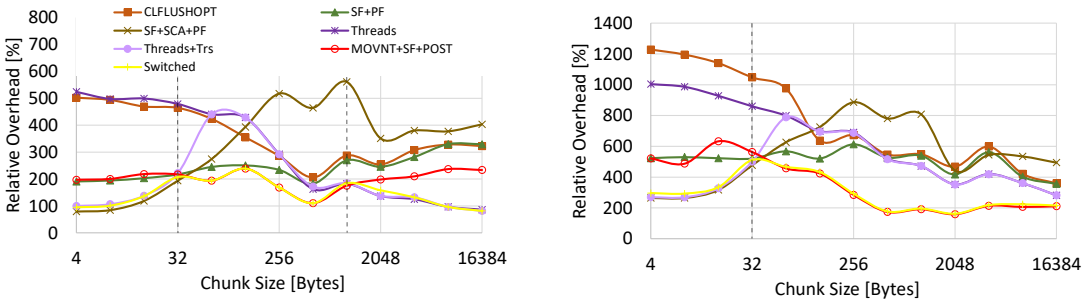
Figure 3.10: Decision tree as a model for the switching algorithm (top level). Particular instance for the testing hardware platform. Terminal nodes further query the model if necessary using the *SubTreeSelect* block.

Replica Count	Minimum Chunk Size	Maximum Chunk Size	Algorithm
...	...	...	...
3	0 B	32 B	SF+SCA+PF
3	32 B	1024 B	MOVNT+SF+POST
3	1024 B	∞	Threads
...	...	...	...

Table 3.1: An example of a profile table (only relevant cells) used by the decision tree to perform the *SubTreeSelect* request to choose an appropriate replication mechanism for the current conditions (corresponds to Figure 3.11(a)).

Online Switching. The actual online switching component leverages the information about the current workload and the pool set configuration to query the lightweight model and finally select a low-overhead replication mechanism. At the level of `libpmemobj` where the replication facilities are implemented, the available workload information is limited to the memory access pattern (sequential or non-sequential), transactional properties, and the updated chunk size. Additionally, the pool set configuration provides information about the pool size, replica count, and memory type. For the specific test system, the key discriminators are (in the order of importance) the memory type, access pattern, transactional properties, and the replica count as it is reflected by the decision tree (cf. Figure 3.10) that serves as lightweight switching model. Actual switching is executed once per replication request.

Implementation. The online switching module is integrated into the replication block of the `libpmemobj` library as depicted in Figure 3.9. All explicitly available characteristics of the workload at that level (transactional properties, updated chunk size, and the sequentiality of the access pattern) are given as parameters or can be extracted without any significant overhead, e.g., by tracking the addresses of the persistent updates for the access pattern. As the lightweight model a decision tree was generated based on the profile information that was partially presented in evaluation paragraphs above. This model is platform-dependent and needs to be calibrated for different hardware platforms. All replication mechanisms that can possibly be chosen by the algorithm are pre-generated as isolated functions that can be called by the switching algorithm instead of basic replication algorithm. Such an approach potentially allows switching also in multi-threaded workloads, however, separate detection of sequentiality is needed.



(a) DRAM Emulation.

(b) NVRAM.

Figure 3.11: Relative overheads for the non-transactional column store workload (CS) using a uniform key distribution. The pool set comprises 3 replicas that need to be maintained. The measurements include the static replication mechanisms as well as proposed lightweight switching algorithm (Switched). Updated chunk size is varied.

Evaluation and Summary. Figure 3.11 illustrates the behavior of the lightweight online switching algorithm using the non-transactional column store workload with three replicas and a uniform key distribution as selected example. The experiment shows that on both NVRAM and DRAM-emulated settings proposed lightweight online switching algorithm is able to give only slightly suboptimal performance at the price of insignificant additional overhead (few percent). That overhead almost disappears with an increasing size of the updated chunk.

Thus, having incorporated suggested above adaptive lightweight online switching component it is possible to largely (up to an order of magnitude) mitigate the drawback of replication runtime overhead within the hybrid memory DBMS and, therefore, resolve the overall challenge RC1 of the thesis (cf. Section 2.6).

3.3 SPACE OVERHEAD AND WEAR-OUT REDUCTION THROUGH DATA COMPRESSION

In contrast to the previous section that deals with the *runtime* overhead of the data replication this part aims to reduce the respective *space* overhead. As identified by the challenge RC2 of this thesis (cf. Section 2.6), the general drawback of the most data replication approaches is the space overhead as every replica normally requires at least 100% of additional storage volume. In hybrid memory DBMS it is reasonable to minimize this additional memory footprint. This is due to the fact that amounts of NVRAM while outbounding those of DRAM (currently up to $8x$) are still much smaller compared to SSDs. The traditional way to reach the objective of space footprint minimization is to compress the stored data [AMF06, GRS98]. Moreover, writing compressed data implicitly could help to reduce the wear-out problem of persistent memory as it has limited endurance. Thus, in the following, this section investigates the opportunities of data compression in NVRAM-centric data stores focusing on the lossless (as this property is obviously required for recovery purposes) lightweight integer compression techniques. Such algorithms are widely used in DRAM-backed columnar processing systems (cf. Section 2.1.4). As a result, this part suggests to store the replicas using compressed data formats.

3.3.1 Integer compression algorithms in hybrid memory databases

Data compression is a well-known optimization technique in database systems [AMF06, GRS98, HDU+19, HHDL16, RVH93]. On the one hand, data compression has been extensively used to optimize the disk access bottleneck in disk-centric database systems [GRS98, RVH93] using classical or so-called heavyweight generic data compression schemes such as Lempel-Ziv [ZL77], Huffman [Huf52] or arithmetic coding [WNC87]. On the other hand, lightweight integer compression algorithms are heavily used to optimize the in-memory processing in DRAM-located column store systems [AMF06, BZN05, HDU+19, HHDL16, LMF+16]. The difference between heavy- and lightweight compression algorithms is their computational complexity.

The goal of this section is to identify the data compression techniques that could be efficiently implemented for NVRAM-stored data in hybrid memory database systems domain. As discussed in Section 2.1.4, due to the mixture of DRAM-like characteristics (e.g., low latencies, direct load/store access semantics, higher bandwidth compared to flash memory) and non-volatility in a single device, NVRAM is primarily expected to complement or replace block-based secondary storage (e.g., HDDs or SSDs) for storing the *primary data set* [OBL+14, OL17b, APP16, Kim15, ALR+17]. While that copy can be at the same time the *working data set*. Despite the fact that persistent memory can potentially provide higher (up to $8x$ of DRAM) capacities [Int19a], data compression is still reasonable due to the following reasons: (i) the amount of data to be stored still can be large even for NVRAM-offered volumes (especially when using replicated data sets), (ii) the NVRAM bandwidth is significantly lower (as illustrated in Section 2.2) than that of DRAM and data compression is a way to increase the effective throughput.

Since in columnar in-memory data processing paradigm (targeted for hybrid memory DBMS) most of the primary data is stored in a form of integer sequences (columns) [BHF09] – main focus of this section is given to the *integer* compression techniques. Further, the intra-memory data flows are used (e.g., the same mediums are employed to store input and output data). As already mentioned, there is a large variety of data compression algorithms available ranging from lightweight to heavyweight schemes. Based on the different read and write bandwidths of DRAM and NVRAM, it is reasonable to investigate the whole spectrum of algorithms in the following experimental study. Thus, the following three different categories of compression schemes are considered: light-, middle-, and heavyweight.

Lightweight Algorithms. This category of algorithms is usually used to optimize query processing of in-memory column stores. These algorithms are not only optimized for compression rate, but also for performance and processing capabilities [DHHL17]. The recent experimental survey for this category on DRAM showed that the performance and compression rate of these algorithms vary greatly depending on the underlying data properties [DHHL17, DUH+19]. Based on that, it is justified to consider two representatives of the state-of-the-art of Null Suppression algorithms which aim at representing each integer value using a minimal number of bits.

First algorithm (*SIMD-BP128*)[LB15] subdivides the data into blocks of 128 32-bit integers each and encodes all data elements in a block using the number of bits required for the block's *largest* data element. Thus, this algorithm profits from small integers. In fact, among all algorithms considered here, SIMD-BP128 is the one investing the least effort into compressing the data. In [DHHL17, DUH+19], it is shown that SIMD-BP128 is a very good choice regarding both compression rate and performance if the data contains no or only few outliers. However, if the data contains many outliers, these dominate the block

bit widths and, thus, lead to a degradation of the compression rate [DHHL17, DUH+19]. In contrast, second algorithm (*SIMD-FastPFOR*) [LB15] is able to adapt to outliers by choosing a bit width suitable for *most* data elements in a block and storing the remaining *exceptions* separately. While this special treatment improves the compression rate in the presence of outliers, it can also yield a degradation of the performance due to the extra effort spent on the outliers [DHHL17, DUH+19].

Middleweight Algorithms. SIMD-BP128 and SIMD-FastPFOR only achieve good compression rates if the data consists of small integers. In practice, this might not always be the case, which motivates a preprocessing of the data to obtain small integers. Thus, two well known and frequently used representatives of such preprocessing techniques are considered: delta coding [RVH93] and frame-of-reference [GRS98], which replace each data element by the difference to its predecessor or to the minimum data element, respectively. The actual compression of the preprocessed data can be achieved by *cascading* the preprocessing with a Null Suppression algorithm [DHHL17, DUH+19]. Such cascades are called middleweight algorithms since the computational effort increases compared to lightweight algorithms, but it may improve the compression rate significantly on suitable data sets. In particular, test study employs *DELTA + SIMD-BP128* and *FOR + SIMD-BP128*. These cascades subdivide the data into pages of 4096 32-bit integers each (which fits into the L1 data cache of test machine) and apply for each page first the preprocessing and then SIMD-BP128 (compression) or vice versa (decompression).

Heavyweight Algorithms. While the previous two categories were mainly designed for in-memory (DRAM) systems, the different characteristics of NVRAM motivate the investigations of general-purpose heavyweight data compression algorithms. Thus, this survey also employs *LZ4* [Com] and *Snappy* [Goo19] as two state-of-the-art representatives. Both LZ4 and Snappy belong to the LZ77 family of byte-oriented compression schemes. The LZ4 source code offers a recommended standard version, which is referred to as *LZ4* in this section. Besides that, it also provides a number of variants targeting different trade-off levels between compression speed and rate. The following evaluation also employs a variant optimized for highest speed, which is denoted as *LZ4s*, and a variant for maximum compression, which is denoted as *LZ4c*. These algorithms are applied on integer data as they would be on binary blobs.

Evaluation The following paragraphs provide detailed experimental analysis of the presented above compression algorithms when deployed on NVRAM and compares their respective behavior to DRAM. Further, the single-threaded and multi-threaded settings are considered.

Implementation Remarks. The same hardware platform as in Section 2.2 is used for experiments. All algorithms are written in C/C++ and compiled using g++-7.1 with the -O3 optimization flag. The light- and middleweight algorithms are vectorized using Intel's SIMD instruction set extension SSE working on 128-bit vector registers (cf. Section 2.3). For SIMD-BP128 and SIMD-FastPFOR, the implementations from the FastPFor-library [LBK+19] are used. For the cascades, hand-tuned and vectorized versions of DELTA and FOR [DHHL17, DUH+19] are implemented. Moreover, the original source for the heavyweight algorithms is utilized. To enable the correct NVRAM usage, all algorithms are modified using two aspects: first, an access to persistent memory is provisioned via memory mapped files on a DAX-enabled file system (XFS), second, a combination of `CLFLUSHOPT` plus `SFENCE` is used to ensure persistency (cf. Section 2.1.3).

	# 32-bit ints	sorted	data distribution
D1	100 M	no	normal($\mu = 2^5$, $\sigma = 20$)
D2	100 M	no	normal($\mu = 2^{25}$, $\sigma = 20$)
D3	100 M	yes	uniform($min = 0$, $max = 10^6 - 1$)
D4	100 M	no	90% normal($\mu = 2^5$, $\sigma = 20$) 10% normal($\mu = 2^{25}$, $\sigma = 20$)

Table 3.2: The synthetic data sets used in the evaluation.

Previous works have shown that the data characteristics determine which algorithm is the most suitable one [DHHL17, DUH+19]. Thus, evaluation relies on several synthetic data sets being summarized in Table 3.2. The DRAM-centric compression benchmark framework is used as a solid execution environment [DHL15, ea19]. Figure 3.12 reports the compression rates achieved by each algorithm on each data set (short names such as BP128 for SIMD-BP128 or DELTA for DELTA+SIMD-BP128 are used in all further figures) and clearly shows that there are significant differences even within each of the categories light-, middle-, and heavyweight algorithms. Obviously these compression rates are independent of the type of memory as well as of the number of threads used, so they serve as a reference throughout the experimental evaluation.

The performance of an algorithm is measured by running it on a data set for few seconds and counting how many times it could process the entire data set. This procedure is especially important to guarantee that all threads are executed simultaneously in the multi-threaded scenarios. Thus, loading/storing data and computations are included in the measurement. We report (i) performances in *million integers per second* (mis) referring to the underlying number of logical data elements, not to the physical data size and (ii) throughput/bandwidth in GiB/s. The Snappy results are omitted in the evaluation, since they were mostly similar to those of standard LZ4. However, further discussion additionally includes LZ4s and LZ4c, which are preconfigured for higher execution speed and for better compression rate, respectively, by the developers.

Single-threaded Experiments. Figure 3.13 presents the performance overview of the single-threaded execution of all algorithms on all data sets. First, the performance on NVRAM is compared to that on DRAM. Figure 3.13(a-b) shows the speeds achieved on NVRAM relative to those on DRAM. As a general observation, the performance on NVRAM is never better than that on DRAM. As a consequence of the lower bandwidth of NVRAM compared to DRAM, the algorithms reach only between 25 % and 100 % in case of compression, or 17 % and 39 % of the DRAM runtime performance in case of decompression.

In the following, the focus is given to the impact of the memory type on the choice of the most suitable compression algorithm. This topic has already been investigated in detail for DRAM [DUH+19]. However, the lower bandwidth of NVRAM suggests to prefer algorithms investing more cycles for computation if this can improve the compression rate appropriately, since that way, the effective bandwidth may be improved. This hypothesis is checked by comparing the DRAM and NVRAM performances of the low-effort SIMD-BP128 to those of the other more compute-intensive algorithms. The compression is under the scope in Figure 3.13(c), since it is generally more compute-intensive than decompression as shown in Figure 3.13(d).

On data set D1 (small integers), no algorithm achieves a significantly better compression rate than SIMD-BP128 and in fact, this algorithm achieves the best compression speed on both DRAM and NVRAM. However, on data set D2 (large integers in a narrow range), the

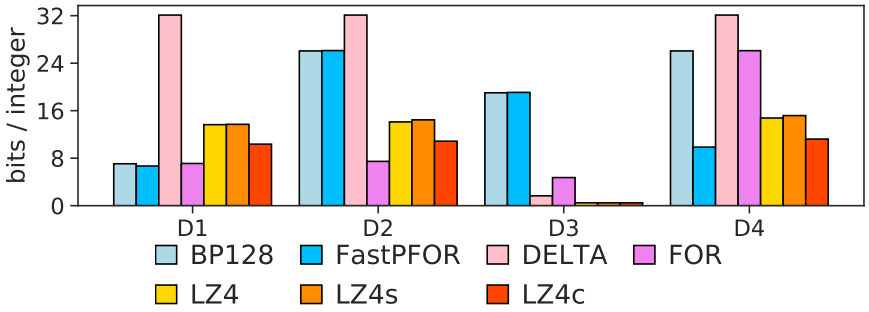


Figure 3.12: Compression rates.

situation changes. Here, FOR + SIMD-BP128 achieves by far the best compression rate. While on DRAM this cascade achieves only 87 % of the speed of SIMD-BP128, it is 72 % faster than SIMD-BP128 on NVRAM. The heavyweight variants of LZ4 also achieve much better compression rates than SIMD-BP128. Nevertheless, they fail to outperform it with respect to compression speed on both mediums, since they are too compute-intensive. On the sorted data set D3, the three variants of the heavyweight algorithm LZ4 achieve the best compression rates. However, regarding compression speed, only the standard LZ4 and LZ4s perform superb: they are fastest compressors on both DRAM and NVRAM, while LZ4c is the slowest of all algorithms on both mediums, due to its high computational effort. Regarding the lightweight algorithms, SIMD-BP128 again is the fastest on DRAM, but again it is outpaced by up to 83% by the more compute-intensive cascades on NVRAM. One tenth of data set D4 are large outliers. Here, the best compression rate is achieved by SIMD-FastPFor, but LZ4c as almost equally good. Regarding SIMD-FastPFor, the price for its special treatment of outliers is a speed of only 35 % of that of SIMD-BP128 on DRAM. However, it achieves a small speed-up of 4 % on NVRAM. On the contrary, LZ4c has a compression speed close to zero on both mediums.

Multi-threaded Experiments. In the following, the observations about the DRAM and NVRAM experiments in the multi-threaded scenario are presented. The compression speeds are shown relative to a single thread in Figure 3.14(a-b, e-f) and the respective absolute memory bandwidth consumed per second in Figure 3.14(c-d, g-h). However, the latter does not distinguish between write and read components, as it was obtained through the Intel *PQoS* monitoring tool [Int19b]. The analysis is based on two selected algorithms and data sets (SIMD-BP128, standard LZ4 on D1, D3) as the most interesting cases.

As mentioned above, the compression rate, of course, is not affected by the number of threads. The performance on NVRAM is never better than that on DRAM for the same number of threads. A further observation is that both compression algorithms exhibit different scalability behaviors on DRAM and NVRAM. The LZ4 DRAM performance grows nearly linearly until 16 and 12 threads on D1 and D3, respectively. For SIMD-BP128 this result is lower, which is explained by the fact that the corresponding consumed bandwidth increases faster: 8 threads on both data sets. However, both approaches demonstrate a stable speed growth up to 20 threads independently of the data characteristics. At this point the DRAM bandwidth limitations are actually reached (cf. Section 2.2). One exception here is the LZ4 (de)compression on D1, which demonstrates a constant growth up to 24 threads (and even further until 48), and is, in contrast to the others, not memory-bound, but compute-bound. The NVRAM scalability diverges from its ideal linear case at 8 (D1) and 4 (D3) threads for heavyweight LZ4 compression and already at 2 for the lightweight SIMD-BP128, while a certain performance increase is still observed until 16 and 8 threads, respectively.

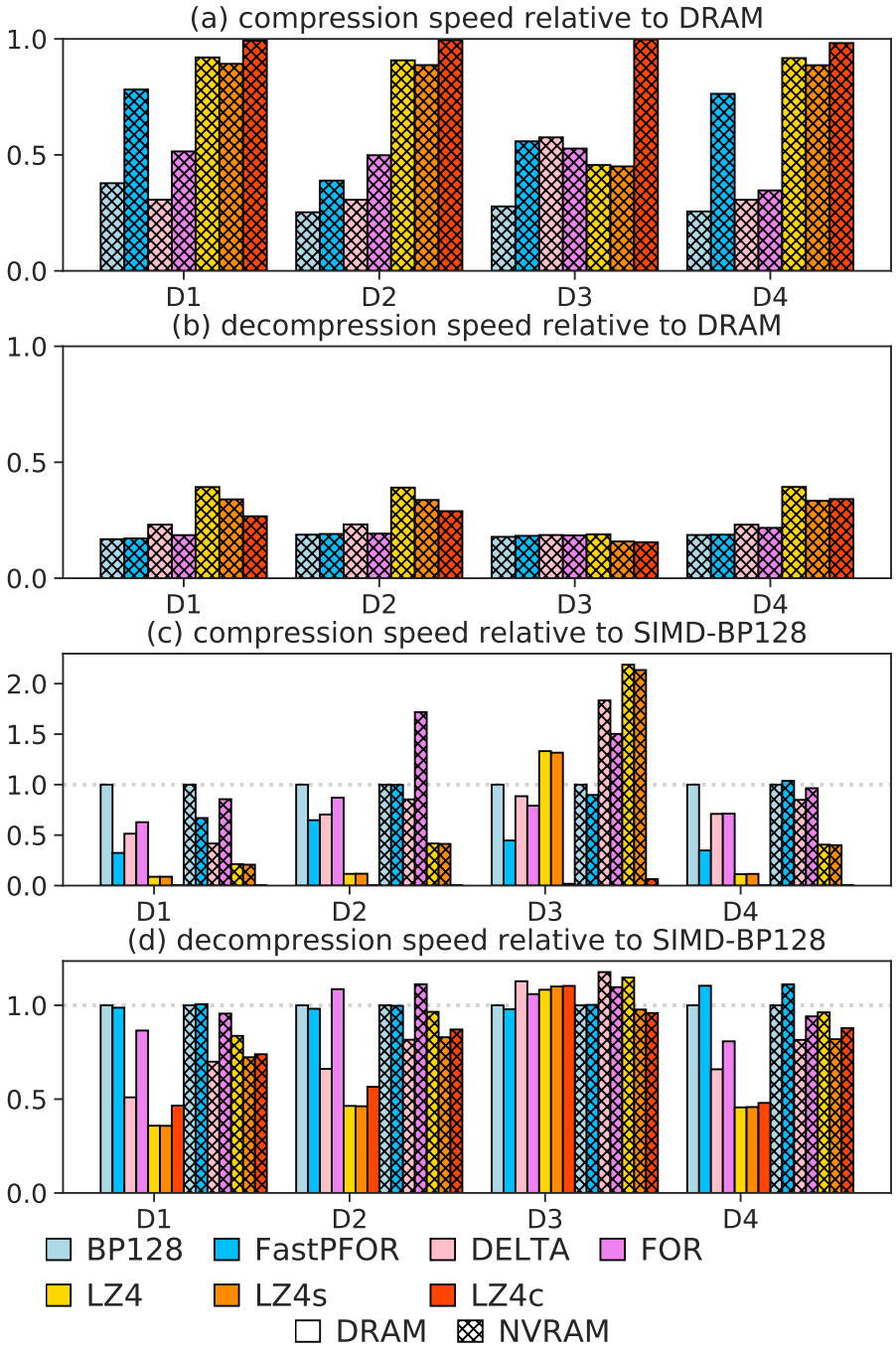


Figure 3.13: Single-threaded speeds: NVRAM relative to DRAM and algorithms relative to SIMD-BP128 *on the same medium*.

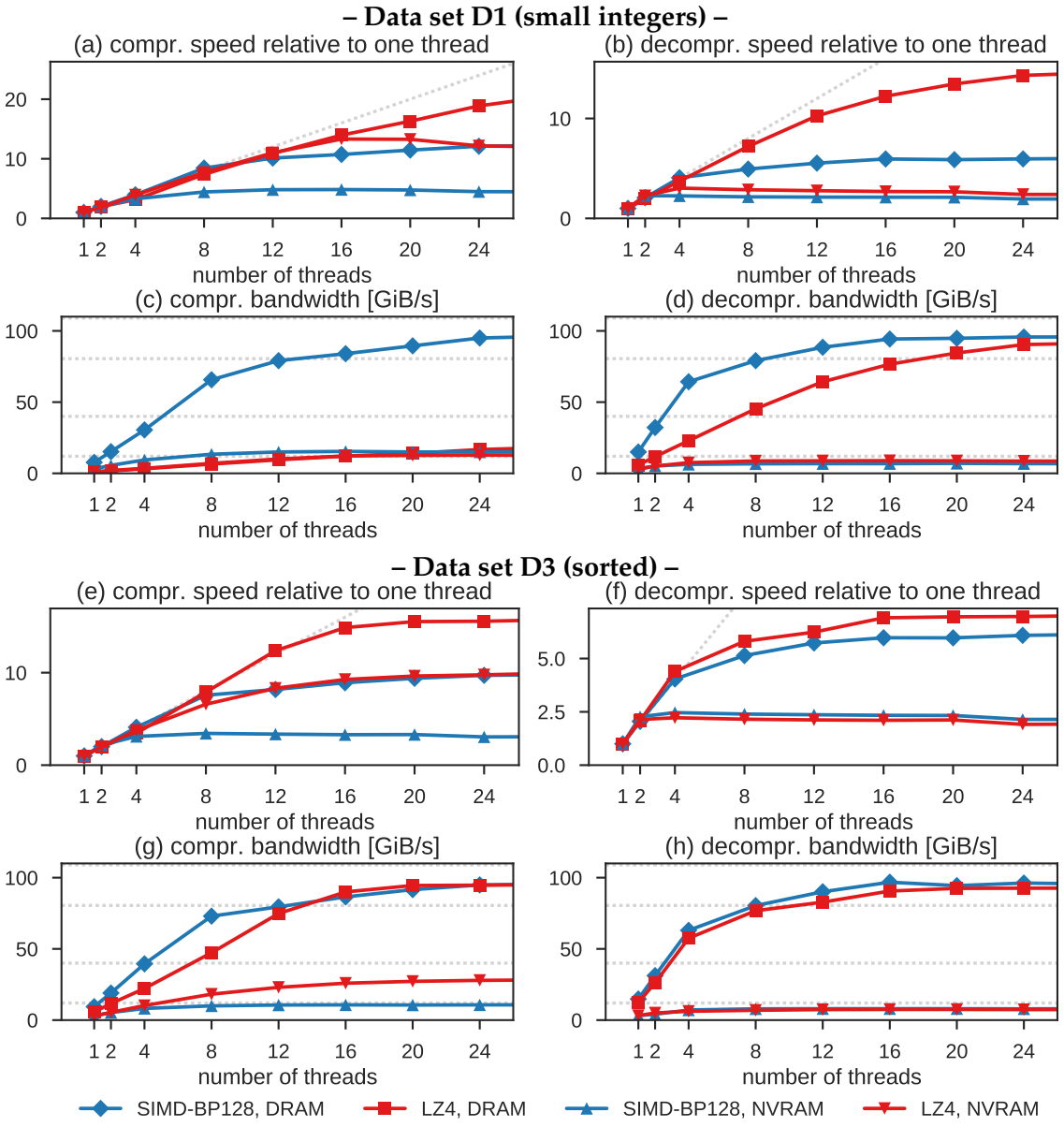


Figure 3.14: Multi-threaded performances on D1 and D3: speed relative to single-threaded (dotted line is linear scaling), absolute bandwidths (dotted lines are read and write bounds).

Table 3.3: Multi-threaded speedups of (de)compression algorithms.

	DRAM				NVRAM			
	Compression		Decompression		Compression		Decompression	
Distribution	D1	D3	D1	D3	D1	D3	D1	D3
SIMD-BP128	$11.5x$	$7x$	$5.3x$	$6x$	$3.7x$	$2.1x$	$1.2x$	$1.4x$
LZ4	$18.5x$	$11.9x$	$14x$	$6.7x$	$11.2x$	$9.5x$	$1.8x$	$1.2x$

Regarding the decompression, one can observe a mostly similar situation with a certain degradation of the thread count: on DRAM, the linear scalability is bounded by 4 threads (except for LZ4 as mentioned above) and stable growth hits the limit at around 16 threads, while on NVRAM, even 2 threads cannot achieve a linear scaling anymore after 8 threads. There is no further performance increase because decompression is more write-intensive than compression and, therefore, is more dependent on write bandwidth boundaries, which are lower than that of the read counterpart on both mediums.

An overview of the accelerations achieved via the deployment of multiple threads (best performance in a range from 1 to 24 threads) for DRAM and NVRAM is provided in Table 3.3.

Conclusions. The evaluation and analysis of this section have shown the applicability and feasibility of DRAM-backed compression mechanisms also for NVRAM domain. Having the presented above results – it could be concluded that the special characteristics of persistent memory necessitate a rethinking of the trade-offs involved in the selection of the fastest compression algorithm. On NVRAM, it is recommendable to invest more computations for a better compression rate. However, while the size reduction achieved by general-purpose heavyweight algorithms does not always balance for their computational cost, the middleweight cascades of lightweight algorithms are a good choice. Further, the multi-threaded scalability of (de)compression algorithms is much better on DRAM, due to higher throughput limits, though there is an exception – the compute-intensive standard LZ4 compression, which scales well also on NVRAM. Moreover, the scalability property depends not only on the medium but also on the algorithm and input data.

3.3.2 Compressed replication concept

This section aims to derive a concept that will allow to store the data replicas allocated in hybrid memory systems in compressed formats. The general drawback of all replication schemes is the additional memory footprint and write overhead caused by the duplication of primary data. This footprint grows linearly with the replica count, however, the increase could be more severe in case of logical replication, as apart of the data itself also the log needs to be mirrored. For hybrid memory architectures, the space complexity is of critical importance due to the following factors: (i) the capacities of both DRAM and NVRAM are much lower than those of disks, and (ii) NVRAM endurance suggests to minimize the number of persistent writes to reduce the wear-out.

Compressed Replication. To mitigate identified above issues, this section proposes to use lightweight integer compression algorithms to represent both the master and its replicas. As previously shown, the light- and middleweight integer compression algorithms are a good choice for both DRAM- and NVRAM-resident columnar data as they exhibit good compression rate, while preserving high speed of execution.

In case of analytical systems [CDN11] – it is generally sufficient to compress the replicated base columns once at startup time. This procedure would impose the compression runtime overheads (or in some cases speedups as much less physical data will be copied compared to pure replication) presented in the evaluation paragraphs above. From the recovery perspective this approach stays as strong as non-compressing state-of-the-art [Kap15] as the same amount of full data copies are safely stored. The difference, however, is that to recover original data – it could be needed to perform a decompression (or re-compression) of the respective memory region (as shown in Figure 3.13-(a) the respective overheads are reasonable and on average are only twice as large as those of DRAM).

However, in many hybrid memory database systems it is important to preserve an opportunity to execute not only analytical (OLAP) requests but also online transactional processing (OLTP) [AAP+17]. Such operations assume certain modifications of primary data. Here, to simplify the (de)compression procedures while propagating updates to the replicas of base data, it is justified to focus exclusively on appending new data elements at the end of a column (since this is the typical scenario in big data and data warehousing systems [Kim15]). Such systems collect data for complex analyses, while new data is periodically added [Kim15] and, thus, replication mechanism can handle this additions using compressed formats. Further, such approach is able to preserve the physical master-slave replication model and provide as strong protection as state-of-the-art technique.

Polymorphic Replication. As discussed above, it can be advantageous to use a certain compressed data format for storing master and replicas. However, there is no single-best compression scheme, neither regarding compression rate nor regarding (de)compression speed, since those metrics heavily depend on the data characteristics [DUH+19]. Furthermore, different query operators on compressed data may favor different formats [AMF06]. This means that query execution speed directly depends on the actual storage format. Thus, this section proposes novel *polymorphic replication* concept by using *various distinct* compressed formats for different replicas *of the same column*. In the field of hybrid memory systems, where most operations are done on byte-addressable columnar data, there is no principal obstacle of doing so. While propogating changes to the replicas the usage of different compression algorithms is expected to cause no or only negligible additional overhead. This is due to the fact that the master data needs to be compressed (or copied) for each replica anyway.

Summary. As data compression leads to a reduced space consumption it allows a better utilization of existing memory capacities and bandwidths during the propagation of changes to replicas, thereby also decreasing NVRAM wear-out. The expected benefit of such a polymorphic replication concept is that diversity in formats can amortize the possible inefficiencies of only a single format and ensure an averaged reduction of the number of NVRAM writes and, thus, replication delays. Therefore, polymorphic compressed replication is a promising approach to tackle the overall challenge RC2 of this thesis.

As shown further in Chapter 4, such replication concept can contribute to analytical query execution as well – when not only masters but also replicas are enabled for read

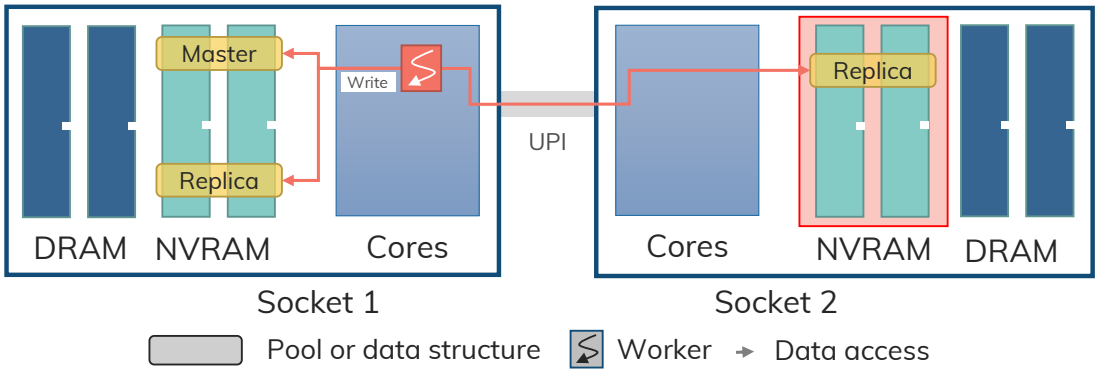


Figure 3.15: NUMA-aware replication in a scale-up hybrid memory system.

requests. The possible benefits here are the following: (i) speedup of processing due to direct operations [HDU+19] on compressed data; (ii) further performance increase when the compressed replica to use for analytical processing is selected depending on the operator's access pattern. The detailed proof-of-concept and respective evaluations are further provided in Section 4.2 of this thesis.

3.4 NUMA-AWARE REPLICA PLACEMENT AS A WAY TO INCREASE RESILIENCE

As already mentioned in Section 2.6, to mitigate all possible NVRAM failure scenarios, and PMM or socket IMC crashes in particular, it is important to support the socket-remote replica allocation on scale-up architectures. Having the replicas distributed over the *distinct* NUMA sockets will increase the resilience guarantees. That comes, however, on the price of socket-remote replica accesses [KZZL17]. If the entire local NVRAM or even the full CPU/IMC fails, such approach will still allow to use data placed on a remote socket on the same machine (possibly migrating query execution threads to healthy remote CPU cores as well). Then system can still work resiliently, while one of the sockets fails. Thus, this section proposes to incorporate such NUMA-aware replica placement as a *compulsory* extension of the basic physical replication mechanism to ensure strongest data protection for hybrid memory database systems on targeted scale-up architectures.

The socket-remote replica placement requirement could be satisfied using the PMDK tool-kit [Kap15]. For that the replicated pool sets should be configured during the opening procedure to leverage NVRAM volumes of distinct sockets accordingly. An example of such allocation is shown by Figure 3.15. Here, a single master and two replica pools are opened in the persistent domain of a two-socket NUMA machine. One of the slaves is moved to remote socket to provide protection against full PMM/IMC failure scenario (indicated as red rectangle on socket 2).

While providing desired resiliency guarantees, the state-of-the art implementation is inflexible and faces several drawbacks. First, the replication policy (or configuration) could only be specified offline. Meaning that it is not possible to change the replica allocation scheme without a system restart and respective pool set reconfiguration. Second, the replication policy could only be specified at per pool level, enforcing all the resident data to be replicated. Obviously that may be unnecessary (e.g., for secondary data) and leads to the wasting of compute and memory resources. Ideally, the DBMS should be able to

decide at runtime on the appropriate strategy which data to protect and specify individual replication scheme at data structure level. These issues were already identified as challenge RC3 of this thesis.

To mitigate these issues and resolve indicated challenge, it is justified to extend the replication component to support individual mirroring policies at *per data structure* level (essentially per column as targeted for hybrid memory DBMS data format). This is also expected to allow for *online* replication policy configuration as could be ensured during the respective memory allocation procedure. The detailed description, evaluation and proof-of-concept are further provided in Section 4.2 of this thesis.

3.5 SUMMARY

The investigations performed in this chapter allowed to reach the ultimate goal of the thesis – minimize the *runtime* overhead of the data replication process (overall challenge RC1). That was done via thorough experimental analysis of the state-of-the-art implementation provided by the Intel PMDK. That implementation follows the compute node-local synchronous physical replication model – most reasonable resilience approach according to the discussion of Section 2.5. The experiments discovered unacceptable performance of basic replication mechanism for the targeted domain of highly-performant hybrid memory databases (cf. Section 3.1). Thus, the detailed examination of the reasons of such overhead was performed. As a one of the major contributions, this chapter suggested a rich set of optimization techniques to minimize the performance costs of replication for typical database scenarios. Those primitives were subsequently united using a template-based approach to enable online adaptivity for current workload conditions (cf. Section 3.2).

The mitigation of the replication runtime overhead problem is of critical importance in the field of highly-responsive hybrid memory databases. However, the performance penalty is not the only drawback of the respective reliability approach. The second major cost of the data replication is the storage overhead. Thus, the next key contribution of this chapter is the investigation of the *space* overhead reduction opportunities targeting the data replication (overall challenge RC2). As normally additional memory foot-print grows linearly with the number of replicas (stressing the capacity limits of byte-addressable volumes), this chapter suggested to leverage the data compression in NVRAM-centric data stores. Whereby particular emphasis was given to *lightweight integer compression* techniques. Such algorithms are widely used in traditional in-memory data processing systems. As an outcome of the respective experimental analysis, it was demonstrated that, on the one hand, the data compression leads to a significantly reduced space consumption at reasonable or, in some cases, negligible computational costs. While on the other hand, it allows a better utilization of memory bandwidths during the propagation of changes to replicas. Finally, a *polymorphic compressed replication* concept was proposed to store the data copies using a variety of compressed data formats and, therefore, to average possible space reductions and minimize NVRAM wear-out (cf. Section 3.3).

As a last contribution, this chapter suggested to enhance the flexibility of the state-of-the-art replication process with regard to NUMA-awareness (overall challenge RC3). Namely, it proposed to incorporate a NUMA-aware replica placement as a *compulsory* extension of the basic physical replication mechanism at per data structure level to ensure strongest data protection on targeted scale-up architectures (cf. Section 3.4).

REPLICATION - QUERY PROCESSING PERSPECTIVE

This chapter [1] derives a conceptual vision on how the compute node-local physical replication, while efficiently facilitating the primary data protection, could be leveraged for query processing needs as well. This opportunity is now enabled by the specific properties of scale-up hybrid memory database systems that are able to persistently store and to efficiently process data exclusively in main memory (cf. Section 2.1.4). Thus, in the following the overall challenges RC3, RC4 and RC5 of this thesis are addressed (cf. Section 2.6). Furthermore, the suggested in this part ideas and optimizations are illustrated based on the proof-of-concept columnar data processing system that incorporates the ideas of compressed replication and NUMA-aware replica placement of Section 3.3.2 and Section 3.4, respectively, as well. The chapter starts with the description of the underlying system model, and then proposes and evaluates two directions for the replication-related query processing optimizations. First direction (Section 4.2) relies on the *storage* component (via diversity in data formats), while second (Section 4.3) suggests to leverage the *compute* counterpart (via diversity in available instruction set extensions).

4.1 UNDERLYING SYSTEM MODEL

With increasingly large amounts of data being collected in numerous application areas ranging from science to industry, the importance of efficient and scalable analytical data processing increases constantly, whereby traditional properties of database systems, such as consistency, longevity and resiliency must be taken into account [CDN11]. From a database perspective, analytical queries usually access a small number of columns, but a high number of rows and are, thus, most efficiently processed using a columnar data storage organization [BMK99, BZN05, HDU+19, SAB+05] which was discussed in Section 2.1.4. The key storage characteristic is that each column of a table is stored separately as a contiguous sequence of values, typically mapped to integers. Here, 64-bit integers are assumed as this is the native word size in most microprocessors today.

To keep the growing amount of data in hybrid main memory (DRAM or NVRAM) for efficiency, data compression using lightweight integer compression algorithms plays an important role in these column-oriented database systems [AMF06, DKB+19, LMF+16, RAB+13, ZHNB06]. As already discussed, such lightweight compression algorithms are well-investigated and they are able to reduce the memory footprint as well as to speed up the data processing with marginal computational effort [AMF06, DUH+19, LB15]. Their applicability for persistent memory and NVRAM-specific considerations were already presented in Section 3.3.1.

Unfortunately, there is a number of possible NVRAM failure scenarios and, as it was argued in Chapter 2, it is essential to protect the *primary* data placed in persistent (and in some cases volatile) main memory. As shown in Section 2.5 and Section 3.2.1 the *software-based data replication* is a most suitable technique for data protection in hybrid memory systems featuring low latency and low throughput penalty. However, this mechanism increases the NVRAM writes leading to the escalation of the endurance problem.

In the following this section details on the data processing workloads, concurrency mechanisms and efficient replication component of the assumed compressed column-store system.

[1] Parts of the material in this chapter have been developed jointly with Patrick Damme, Alexander Krause, Dirk Habich, and Wolfgang Lehner. The chapter is based on [ZDHL20, ZDK+21], whereby [ZDHL20] mainly contributed to Section 4.2 and [ZDK+21] mainly contributed to Section 4.3. The copyrights of [ZDHL20] and [ZDK+21] are held by the Association for Computing Machinery (ACM); the original publications are available at https://doi.org/10.1145/3383669.3398283 and https://doi.org/10.1145/3456727.3463782, respectively.

Workloads. With respect to the accesses to primary data, assumed system supports three different workloads. *Read-only workloads* represent analytical queries [GS19], which are implemented as sequences of *operators*, whereby each operator takes one or more columns as input and returns one or more columns as output. The output columns are intermediate data and are fully materialized in DRAM. The operator-at-a-time processing model [BMK99, HDU+19] is assumed here, i.e., only one operator is allowed to be executed at a time within a single query. As already mentioned in Section 2.1.4, typical columnar query operators include `select`, `project`, `aggregate`, `join`, `group-by`, and set-operations.

For instance, the `select`-operator performs a sequential read of its single input column, compares each data element with a constant, and sequentially writes matching positions to its single output column. The `project`-operator reads an input column of positions sequentially, extracts the values at the given positions from its second input column using a random read access, and sequentially writes these values to its output column. As a last example, the `aggregate`-operator calculates a cumulative statistical value, e.g., the sum, of all elements of its single input column. It performs a sequential read, while only a single value is written. *Write-only workloads* allow to change the base data. Here, the emphasis is given exclusively to *appending* new data elements at the end of a column, since this is the typical scenario in big data and data warehousing systems [CD97]. Such systems collect data for complex analyses, while new data is periodically added [CD97]. Hence, when modifying the base data, assumed system produces a sequential write memory access pattern. Finally, *mixed workloads* combine analytical queries and append-operations in an arbitrary ratio [LMF+16].

Scalability and Concurrency Control. The architecture of a data processing system determines its *scalability* behavior [AAP+17, KKS+14b, PSM+16]. The hardware resources, e.g., the number of NUMA sockets and cores per socket (CPU), define the physical limits of parallelism, while the software defines a trade-off between the speed up achieved via parallelism on the one hand and consistency and freshness guarantees on the other hand. This is especially crucial in case of write-only and mixed workloads. Such a software-level *concurrency control* [KM17] is used to solve the synchronization problems between parallel readers and writers. The *multi-threaded inter-query* parallelism is assumed, meaning that one query is mapped to one thread at each point in time. To allow a high degree of parallelism without long blocking operations, the concurrency is controlled via *snapshot isolation* [CG16] on the query level. This means that every query atomically takes a snapshot of all data columns it needs to access as its prologue phase and does not see any changes made after this point by other threads. Therefore, analytical queries are executed concurrently without exemptions, while only one appending thread is allowed per column at a time.

Replication Mechanism. In agreement with introduced above architectural principles, replication is applied only for primary data, because intermediate data can be regenerated by re-executing after a failure. As already noticed in Section 2.6, the replication mechanism of Intel PMDK faces a number of issues or inflexibilities when applied in scale-up hybrid memory environment. Those are the following: (i) only masters are readable while replicas serve solely for recovery purposes; (ii) all involved replicas should follow same (uncompressed) data format; (iii) the replication policy could only be specified offline and at per pool level; (iv) the replication affects only NVRAM-resident data within PMDK-integrated hybrid memory database; (v) existing approach has limited NUMA-awareness. Identified issues argue that the state-of-the-art approach lacks the desired level of flexibility for the DBMS to decide at runtime on the appropriate strategy

which data to protect, specify individual replication scheme at data structure level and use replicas for query processing purposes.

Because of that, assumed in this section columnar processing system proposes to incorporate a *modified* physical replication mechanism with the goal to alleviate the inconveniences of the state-of-the-art approach, while presuming its advantages of *synchronous master-slave replication* model. That means, the primary data is anyways modified first in the master, before the changes are sequentially propagated to the replicas using memory move semantics. An append-operation is only committed to an application or user when all involved copies have been processed successfully. Hence, the persistence is required only from that point on. The four core ideas of the advanced modified replication scheme are presented in the next section.

4.2 POLYMORPHIC COMPRESSED REPLICATION MECHANISM

This section describes the details of novel *polymorphic compressed replication (PCR)* mechanism which was partially introduced in Section 3.3.2. This mechanism consists of four main concepts that allow not only to reduce the replication space overhead but also to exploit the replicas for analytical processing: (i) concurrent read access to replicas, (ii) compressed replication, (iii) polymorphic replication, and (iv) hybrid memory placement.

4.2.1 Optimization concepts

Concurrent read access to replicas. In scale-up systems, the multi-threaded execution (cf. Section 2.1.2) of queries for scaling the performance using inter-query parallelism is state-of-the-art [LBKN14]. However, as already mentioned, current NVRAM replication approaches allow only the master to be accessed by the application directly [Kap15, ZYMS15]. Since the master resides on one particular socket, threads on other sockets are forced to perform remote memory accesses, which suffer from a comparably low bandwidth and high latency (cf. Section 2.2). Thus, the scalability in NUMA systems is severely limited with current replication approaches. To alleviate aforementioned issue, this paragraph proposes the employment of replicas on different sockets for reading purposes. The strict need of socket-remote replication for reliability purposes (e.g., to tolerate PMM/IMC or complete socket failure scenarios) was already stressed in Section 3.4. Using replicas to speed up readers is a well-known approach in the field of distributed data processing systems, where a cluster of compute nodes is able to serve requests at an arbitrary node as each of them holds a copy of the primary data [EKA19]. Applied to NUMA architectures, which actually behave similar to distributed systems, this scheme allows to leverage CPUs and memory bandwidths of all sockets holding at least one replica to the full extent as illustrated by Figure 4.1. It is expected that the processing throughput of read-intensive queries can be largely increased compared to a single-socket execution that way. Thus, this suggestion allows to tackle the NUMA-centric challenge RC3 of the thesis.

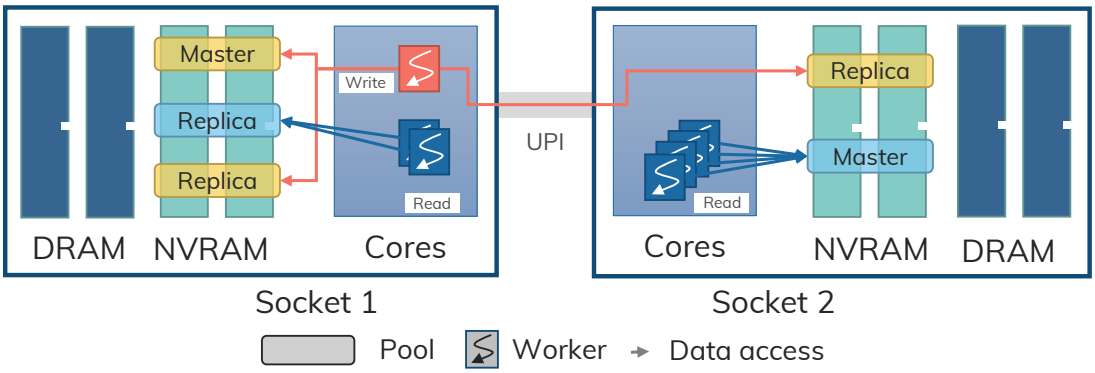


Figure 4.1: Concurrent read accesses to replicas in a scale-up hybrid memory system (illustrated for blue-colored pool).

Compressed replication. As already noted in Section 3.3.2, the general drawback of all replication schemes is the additional memory footprint and write-overhead caused by the duplication of primary data. For hybrid memory systems, the space requirement is of critical importance due to the following factors: (i) the capacities of both DRAM and NVRAM are much lower than those of disks, and (ii) NVRAM endurance suggests reducing the number of persistent writes to reduce wear-out. To mitigate these issues, Section 3.3.2 already proposed to use lightweight integer compression algorithms (schematically shown in Figure 4.2) to represent both the master and its replicas. As data compression leads to a reduced space consumption, it allows a better utilization of available memory capacities and bandwidths during the propagation of changes to replicas, thereby also decreasing NVRAM wear-out. This paragraph, in contrast, proposes to leverage compressed replica formats for query processing. Doing so analytical queries can benefit from the facts that they (i) can process more logical data within the same physical bandwidth limitation, and (ii) can speed up processing due to direct operations on compressed data [HDU+19].

Polymorphic replication. Storing master and replicas in compressed form is generally advantageous. However, as shown in Section 3.3.1, there is no single-best compression scheme, neither regarding compression rate nor regarding (de)compression speed, since

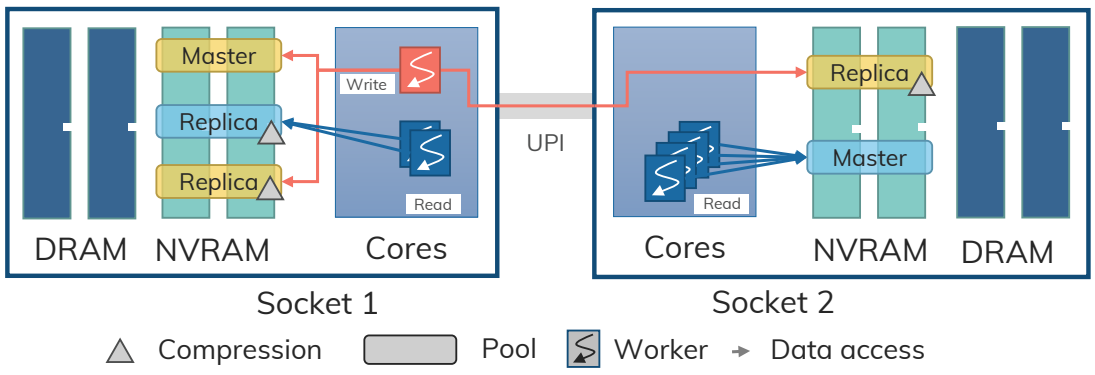


Figure 4.2: Compressed replica formats to reduce space overhead and speed up query processing.

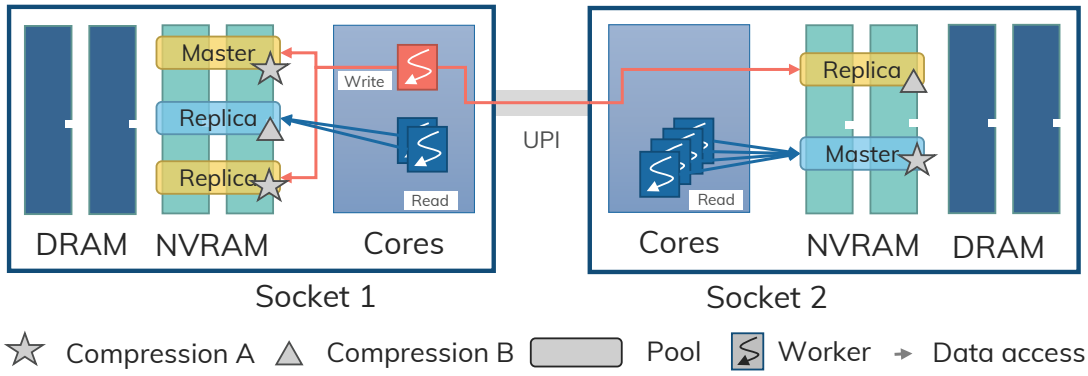


Figure 4.3: Polymorphic compressed replica formats to average space overhead reduction and speed up query processing.

those metrics heavily depend on the data characteristics [DUH+19]. Furthermore, different query operators on compressed data may favor different formats [AMF06]. Thus, this paragraph suggest to take the advantage of employing *polymorphic replication* (illustrated by Figure 4.3) by using *various distinct* compressed formats for different replicas *of the same column* as already proposed in 3.3.2. However, here the goal is not to average the space overhead reduction of replication but, in contrast, to select the best available format to speed up query execution.

During the replication, having a variety of formats implies a certain computational overhead, since the compression algorithms of all employed formats must be executed. However, as further shown in Section 4.2.3 the corresponding bandwidth consumption is not increased, since the master is read just once and each replica is written once.

For the illustrative purposes of the proof-of-concept columnar system the two representatives of *binary packing (BP) null suppression* algorithm are adopted [LB15, SV10]. The idea of this algorithm is to represent all integers in a block of 64 64-bit integers with the number of bits required for the largest of them. This variant is called *dynamic BP*. Determining the largest value requires an additional pass over each block. While this allows the algorithm to adapt to local variations in the data distribution, it also adds a computational overhead. Therefore, targeted system considers *static BP* as an alternative which views the entire column as a single block and represents each value with the bit width of the column's possible maximum value, which is assumed to be known from the application. An advantage of such simple compression algorithms is that the compressed data can be processed directly without decompression. In particular, both dynamic BP and static BP allow sequential access, while static BP also allows efficient random access.

The benefits of polymorphic compression are: (1) It can facilitate efficient processing and increase the performance, e.g., when the replica to use for analytical processing is selected depending on the operator's access pattern. For instance, on the one hand, static BP and dynamic BP both support sequential access, but dynamic BP can adapt to the local data distribution and, thus, result in a lower physical replica size, such that a sequential scan has to read less data. On the other hand, only static BP supports efficient random access. (2) A diversity in formats can amortize the possible inefficiencies of only a single format and ensure an averaged reduction of the number of NVRAM writes and, therefore, replication delays. Thus, the ideas of this and previous paragraphs allow to tackle efficiently not only the overall challenge RC2 of the thesis, but also challenge RC5 focused on query execution needs (cf. Section 2.6).

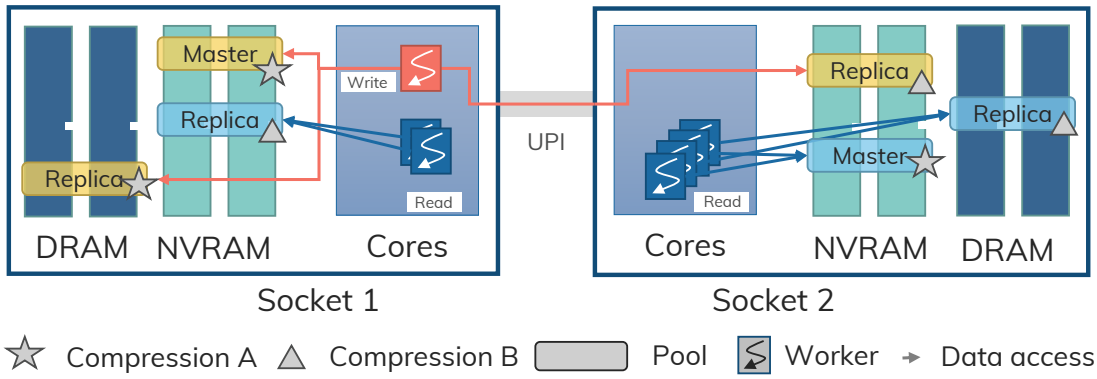


Figure 4.4: Unified hybrid memory replica placement to speed up query processing and reduce NVRAM wear-out.

Hybrid memory placement. While the resilience property strictly requires a certain number of replicas to be resident in persistent memory, it could contradict the performance requirements. This is especially true for read-intensive workloads due to the higher latencies and lower bandwidth limitations of NVRAM compared to DRAM (cf. Section 2.2). Thus, using both DRAM and NVRAM for data processing is a natural idea for single-level hybrid memory systems. To overcome persistent memory performance bottlenecks, some replicas could be placed in DRAM and leveraged for read workloads (as illustrated by Figure 4.4). Since both memories are distinguished only on the virtual addressing level, only insignificant changes are required to the application. However, that requires a certain amount of DRAM to be reserved. Volatile replicas can possibly contribute to the following aspects of the underlying data processing system: (i) performance, as accesses to DRAM are considerably faster than those to NVRAM, especially in case of multi-threaded execution, (ii) a reduced wear-out of NVRAM devices, since less writes will be forwarded to persistent memory during the replication, and (iii) persistent capacity increase, as the placement of replicas in DRAM would free the respective NVRAM memory region. Thus, this paragraph suggests to use a *unified* replication mechanisms independently of the actual byte-addressable medium and compressed format. Further, the replication policy is proposed to be specified at per data structure (essentially per column) level to allow for more flexibility. These ideas target the challenge RC4 of the thesis.

4.2.2 Implementation

This section presents C++ implementation of the previously suggested PCR mechanism for a *resiliency-aware analytical processing*. Since PCR makes replication itself and the usage of replicas more complex, it is justified to separate replication functionality from the actual data processing by factoring it out as a separate user-space *abstract library*[2] to achieve a transparent deployment of PCR in an application-agnostic way.

While the PMDK library (cf. Section 2.1.3) is considered to be a standard for industrial NVRAM-centric development, it suffers from a lack of flexibility in certain aspects (cf. Section 4.1). First, its persistent allocator is pool-based, meaning that a persistent memory region of arbitrary size is reserved during the opening procedure and is used until a reallocation is needed. However, this approach is not NUMA-aware, as such a pool

[2] https://github.com/MorphStore/Engine/tree/pcr

(a) API functions

```
replicated_t* allocate(size_t size, hint_t* hint)
void* get_buffer(replicated_t* rbuf, format_t& format, hint_t* hint)
void append(replicated_t* rbuf, uint64_t data)
delete rbuf
```

(b) Example usage of PCR abstract library

```
1  void insert_some_data(replicated_t* primary_data) {
2    for(uint64_t i = 0; i < 100; i++)
3      append(primary_data, i);
4  }
5  uint64_t aggregate(replicated_t* replicated_data) {
6    format_t f;
7    void* replica = get_buffer(primary_data, f, SEQ_READ);
8    if(f == A) return aggregate_on_format_A(replica);
9    if(f == B) return aggregate_on_format_B(replica);
10 }
11 void analytical_query(replicated_t* primary_data) {
12   replicated_t* snapshot = take_snapshot(primary_data);
13   printf(aggregate(snapshot));
14 }
15 void main() {
16   replicated_t* primary_data = allocate(100);
17   insert_some_data(primary_data);
18   analytical_query(primary_data);
19   delete primary_data;
20 }
```

Figure 4.5: API and usage of PCR abstract library.

can deploy memory of only a single socket. To support NUMA-specific allocation with PMDK, one pool per socket would need to be opened, even if it is not actually going to be used. Second, the basic replication scheme of PMDK does neither allow read-access to replicas nor to use an individual format for each replica. Finally, the PMDK covers only NVRAM-resident data, but cannot place replicas in DRAM. These limitations justify the implementation of a more flexible replication subsystem supporting PCR approach.

In the following, the abstract library is presented from a system developer's point of view, and subsequently the decisive internal implementation details are revealed.

Transparent Integration of PCR. This part introduces the basic functions provided by the abstract library. These are summarized in Figure 4.5 (a) and reflect the typical life cycle of dynamically managed memory.

Memory allocation. To reserve a byte-addressable memory region, an application typically calls a certain memory allocation function that returns a pointer. Similarly, PCR library provides an `allocate`-function, which performs the allocation of all buffers for the master and replicas. The difference compared to `malloc` is that `allocate` does not return a pointer, but an abstract handler of type `replicated_t` and optionally considers hints regarding the preferred usage of the PCR concepts.

Read-access. To make use of the replicas for reading purposes, a certain replica can be addressed as a normal pointer. This pointer is obtained via the `get_buffer`-function, which internally selects a suitable replica, optionally based on hints regarding the intended access pattern and also returns the compressed format of the chosen replica so that operators know how to process the data.

Write-access. Write operations have to take care of replication and data compression. Therefore, writing to a normal pointer does not suffice. Instead, all the data have to be appended elementwise through the `append`-function, which ensures the propagation to the replicas including an on-the-fly compression of the appended values.

Freeing. Deallocation works in the traditional way by using the `delete`-operator on a `replicated_t` handler.

Example. Figure 4.5 (b) shows a simplified example of how to use the abstract library. At system startup, a replicated column is allocated (line 16). Then, some data elements are appended to it (line 17), whereby the `append`-function is used (line 3). After that, an analytical query is processed (line 18), which first takes a snapshot (cf. Section 4.1) (line 12), which is a shallow copy of a replicated column, and passes this snapshot to an operator (line 13). The operator obtains an ordinary pointer using `get_buffer` (line 7) and forwards it to a format-specific operator implementation (lines 8–9). In practice, more complex variants of `insert_some_data` and `analytical_query` would be executed concurrently.

Implementation Details. Following part provides some important details of the internal implementation of PCR abstract library.

Main data structure. As an abstract handler for replicated data chunks, library implements a specific container `replicated_t` that holds all necessary information about a replicated buffer. In particular, it stores metadata valid for all replicas of a column, such as the logical number of elements, as well as replica-specific information, such as the socket, memory type, format, and compressed data size.

Memory allocation. With regard to the volatile memory allocation the library relies on the `libnuma` allocator [lib20] to ensure NUMA-awareness, while for NVRAM allocation, it provides a custom lightweight implementation, as justified above. Similarly to PMDK, integrated allocator provides an access to persistent memory via memory mapped files on a DAX-enabled file system(cf. Section 2.1.3). The difference is that PCR-integrated version reserves a separate file for every data buffer and, thus, is able to designate it to any of the sockets following a specified NUMA policy.

Read-access. Since there could be several replicas available, the `get_buffer`-function needs to decide which one to return to the caller, based on the optional hints. The default approach is to return a socket-local replica to follow the NMP paradigm (cf. Section 2.1.2). Obviously, this is not the most efficient way when a variety of replica formats is available. To guide the selection process, a small decision tree was designed with the goal to leverage information about the available replicas as well as user-provided hints, such as the intended access pattern. Generally, the in-depth development of a strategy for selecting the optimal replica is beyond the scope of this thesis. However, a promising direction could be the deployment of performance counters monitors to take into account the current memory workload on the socket as well.

Write-access. To avoid function call overhead, it was ensured that the `append`-function is inlined by the compiler. Internally, it is processed in two main steps: (1) the values to

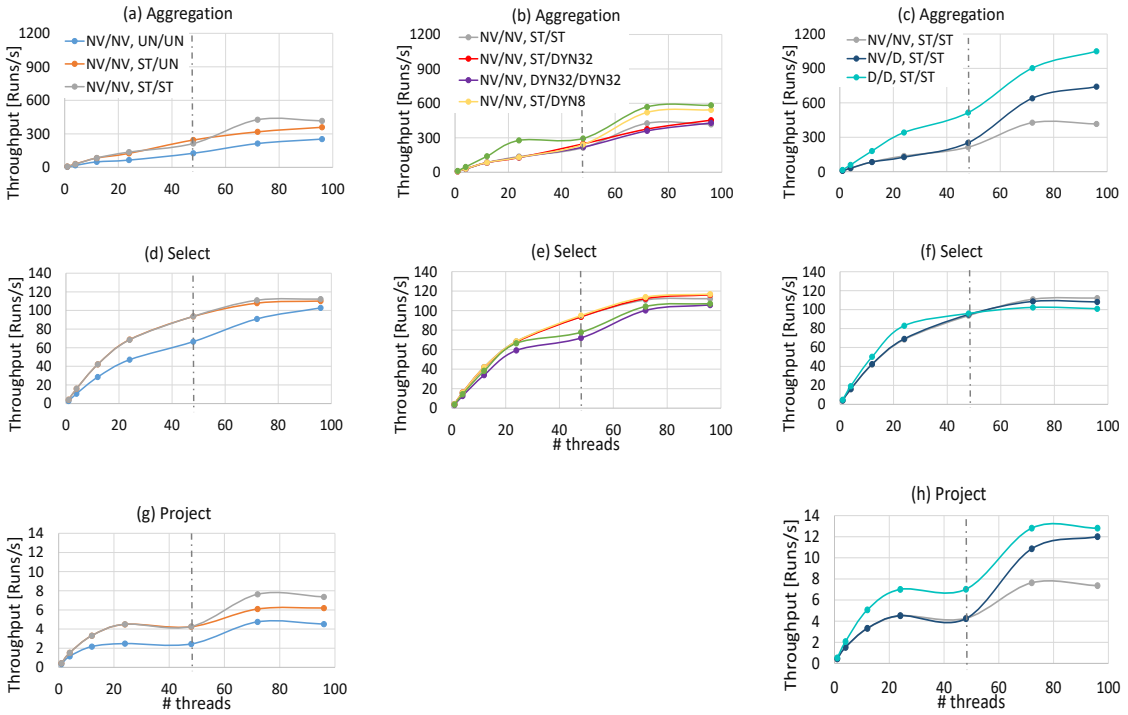


Figure 4.6: Absolute throughput of selected columnar operators for different 2-replica allocation scenarios. The gray dashed lines indicate the inter-socket boundary (thread count of 48).

append are staged in an internal cache-resident (volatile) buffer, and (2) once this buffer is full or all planned data has been inserted, its contents are sequentially propagated to all replicas by applying the respective lightweight compression algorithm on-the-fly or by pure copying. To flush the caches and to ensure persistence for NVRAM memory chunks, the library uses a combination of `CLFLUSHOPT` plus `SFENCE` (cf. Section 2.2). Since PCR follows the *synchronous* approach, the changes are only committed (i.e., guaranteed to be made persistent) if all replicas are updated successfully.

4.2.3 Evaluation

This section evaluates suggested *PCR* mechanism, first by using microbenchmarks, then by switching to an established benchmark for complex analytical queries. All experiments use *MorphStore* [HDU+19] – prototype of an in-memory query processing system for columnar data, as a solid environment for the execution and integration of *PCR*.

Hardware Setup. The evaluation platform in this section is a two-socket NUMA system equipped with Intel Scalable Cascade Lake processors, 384 GiB DDR4 DRAM, and 1.5 TiB Intel Optane DC Persistent Memory. Each processor has 24 physical cores (48 with HyperThreading). All CPUs are pinned to a frequency of 2.6 GHz. The server runs Fedora 27 with kernel version 4.15 and `g++ 8.1.0` was used as the compiler.

Read-Access Microbenchmarks. First, the impact of the proposed concepts based on concurrent analytical workloads is shown. More precisely, the experimental setting is the following: two replicas (e.g., a master and a slave) are placed in memory on two distinct sockets (as argued in Section 3.4). The logical number of data elements is 100 M per copy, corresponding to approximately 763 MiB in the uncompressed case. The synthetic data is generated following two different distributions. The first one yields random 32-bit values, thus, it is expected to be represented with 32-bit per data element by both variants of BP, thereby halving the data size. The second one yields 99.9% 8-bit values, but 0.1% 32-bit values, thus, static BP still requires 32 bit per element, while dynamic BP can use 8 bits per element in most blocks and only a few blocks require 32 bits per element, thereby almost quartering the data size. Since these distributions only have an impact on dynamic BP, they are only distinguished in the experiments involving this compression algorithm. To evaluate the idea of concurrent read accesses to replicas, the thread count is varied from 1 to 96, by first deploying the native cores of socket 1 (1 to 24), then adding its hyper cores (25 to 48), further the same scheme is applied to socket 2. Note that for thread counts between 1 and 48, only the single replica local to socket 1 is leveraged for querying following the NMP paradigm [KZZL17, PJHA, KKS+14b]. The measurements were taken using a 1 minute runtime interval.

Compressed replication. Figures 4.6 (a, d, g) illustrate the influence of compression, where the static BP (denoted by *ST*) is used on the `aggregate`, `select` and `project` operators, respectively. The experiments are executed in the baseline NVRAM-only setup (denoted by *NV*). Using the compressed replica format improves the throughput of the `aggregate`-operator in terms of queries executed per second on a single socket, e.g., for a thread count of 48, by 69% compared to the uncompressed case. If compression is applied to the second replica as well, this increase stays stable, e.g., 64% for 96 threads, and gives almost a doubled performance improvement compared to the single-socket throughput, which is also the case for uncompressed measurements. The mixed case, when the first replica is compressed and the second one is uncompressed, fills the performance gap between compressed-only and uncompressed-only configurations, e.g., 40% for 96 threads.

For the `select`-operator, it was ensured that 10% of the data elements match the condition. Here, similar observations hold on a single socket where an increase of 41%, gained by the compressed data format, is observed for 48 threads. However, the situation changes when deploying the second socket. Now, only the uncompressed format still shows a good scalability. However, it obviously has a worse throughput. The compressed case gains only 20% additional improvement compared to the single-socket performance. It could be assumed that the reason is the contention in the cores-to-memory mesh and memory controllers. Such contention is specific to test hardware and appears only for complex mixed workloads executed with high concurrency. For instance, the `select`-operator mixes sequential reads of primary data with sequential writes of intermediate data (cf. Section 4.1).

With respect to the `project`-operator, an unsorted input position column ten times larger than the input data column was generated to induce a random read access pattern on the latter. The outcome of the corresponding experiment is similar to that of the `aggregate`-operator, except for two differences: First, the absolute throughput is generally much lower, which can be explained by the random read access pattern limits. Second, the hyper cores do not contribute to throughput scalability on any of the NUMA sockets. The reason for this is similar to the previous case, as the respective memory bandwidth is already saturated by the native cores.

Polymorphic replication. Figures 4.6 (b, e) show the impact of polymorphic compression, meaning that the distinct compression schemes are used for different replicas. Here, the static BP (denoted by *ST*) and dynamic BP (denoted by *DYN*) are employed. The experiment also analyses both data distributions mentioned above and denote the performance

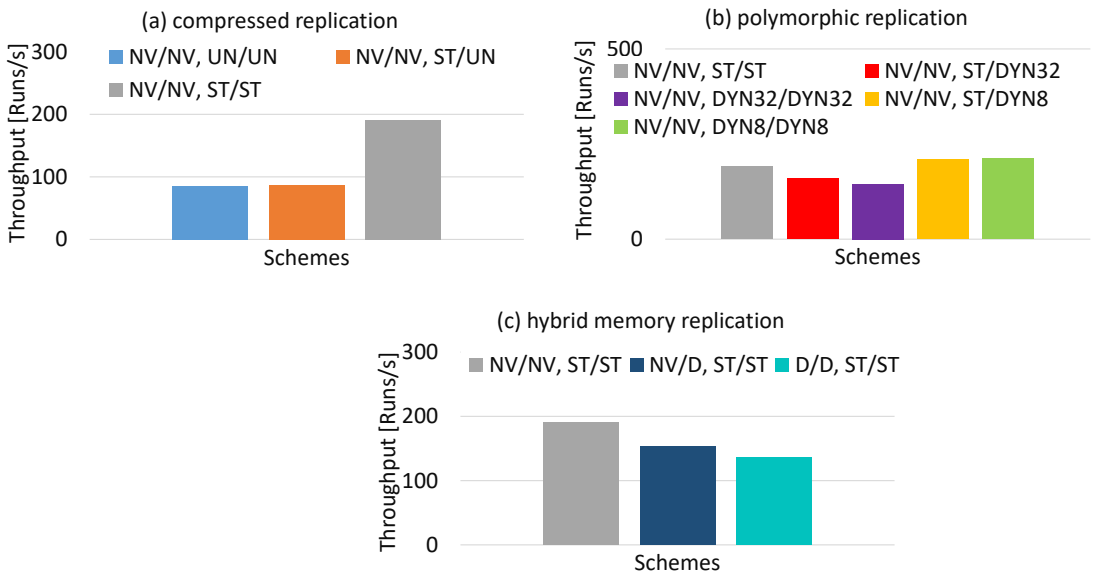


Figure 4.7: Absolute throughput of single-threaded append workloads for different 2-replica allocation scenarios.

of dynamic BP on the first (second) one as *DYN32* (*DYN8*). Again, the experiments are executed in the NVRAM-only setup. The `project`-operator involves random access. Since this is only supported for static BP, the `project`-operator is omitted here.

In the polymorphic setting, the `aggregate`-operator again demonstrates a strong scalability feature, as for all deployed schemes, the absolute throughput is nearly doubled with the socket count expansion. However, for certain scenarios, e.g., those involving static BP or heavily compressing dynamic BP (DYN8), the socket-local scalability is already bounded by native cores (24 threads). Generally, the difference in absolute performance for the same thread counts among the analyzed schemes is just a few percent, except for the well compressing dynamic BP cases. These setups can outperform the alternatives by 35%, e.g., for 96 threads. This result is explained by the fact that aggregation needs to read approximately four times less data for a dynamic BP replica in the DYN8 case compared to a static BP replica.

The behavior of the `select`-operator in the mixed case, i.e., one static and one dynamic BP replica, in terms of performance and scalability mimics the pure static BP scenario, while the pure dynamic BP schemes demonstrate slightly lower performance after 24 threads (native cores of a single socket). It is assumed that this is a consequence of hardware-specific limitations mentioned above, where the corresponding contention point is reached faster while decompressing dynamically packed buffers.

Hybrid memory replication. Figures 4.6 (c, f, h) reflect the influence of the memory class employed as the storage medium for replicas. The experiments use static BP for both involved replicas, while either one or both of them are placed in DRAM (denoted by *D*).

For the `aggregate`-operator, the choice of the memory class has a crucial impact on the behavior. Accommodating one replica in volatile memory already increases the performance by 80% for 96 threads, while the pure volatile scenario yields a 250% speedup. Furthermore, the volatile placements improve the scalability property, as the higher DRAM bandwidth limits allow more efficient usage of hyper cores, especially when both sockets are operating.

The situation changes dramatically when shifting to the more compute-intensive `select`-operator. Here, the memory choice has virtually no impact, as the maximum throughput varies at most by 10% between different mediums. While the employment of a second socket is again susceptible to hardware-specific memory controllers contention issue.

The `project`-operator experiment results are mostly similar to the outcome of the `aggregate`-operator. Volatile replicas can lead to an almost doubled throughput. There is, however, one difference compared to the `aggregate`-operator: the scalability does not improve anymore when hyper cores are used. This reflects the fact that random access patterns, which are exhibited by the projection, scale generally worse than the sequential access patterns, which dominate in the aggregation, on both mediums.

Write-Access Microbenchmarks. This part analyzes the behavior of replication in the context of workloads modifying the primary data (cf. Section 4.1). The same data distributions as above are used. Note that the possible memory reallocations were avoided in these experiments.

Replication Performance. In the following set of experiments, the impact of *PCR* on write-intensive workloads is investigated. Essentially, the behavior of the single-threaded append-only execution is analyzed, since the concurrency control scheme allows only one modifier at a time. The experiment appends data chunks of an uncompressed size of 1 MB each and varies the data distributions to support the DYN8 and DYN32 cases. The query executor was always pinned to socket 1 (reflected by the left side of / in the inscriptions).

Compressed replication. As demonstrated in Figure 4.7 (a), in the case of two compressed replicas the throughput of the `append`-operator significantly outperforms the uncompressed counterpart by approximately two times. This is explained by the fact that the amount of data to be written is basically halved as a consequence of the static BP compression. Such an increase, however, is not visible in case of a mixed scenario, where the uncompressed replica is being updated on the remote socket's memory. Therefore, the replica propagation suffers more from the delays of inter-socket communication while transmitting the uncompressed data.

Polymorphic replication. Figure 4.7 (b) shows the difference between employed compressed formats. The main observation here is that static BP should be favored if the data distribution is not well suited for dynamic BP (DYN32), while the opposite holds for highly compressable data (DYN8).

Hybrid memory replication. Finally, Figure 4.7 (c) illustrates the influence of the memory class deployed for replica placement. Surprisingly, the experiment revealed the advantage of NVRAM for the targeted workload compared to DRAM-resident schemes. Similar behavior, however, was already detected by another NVRAM replication study [ZKHL18] and actually is only observed for the single-threaded non-vectorized scenario. Hence, it could be assumed this is a consequence of the way how DRAM is accessed by the memory controller comparing to NVRAM and persistent memory buffering that internally uses a cache line size of 256 bytes [PIL+19].

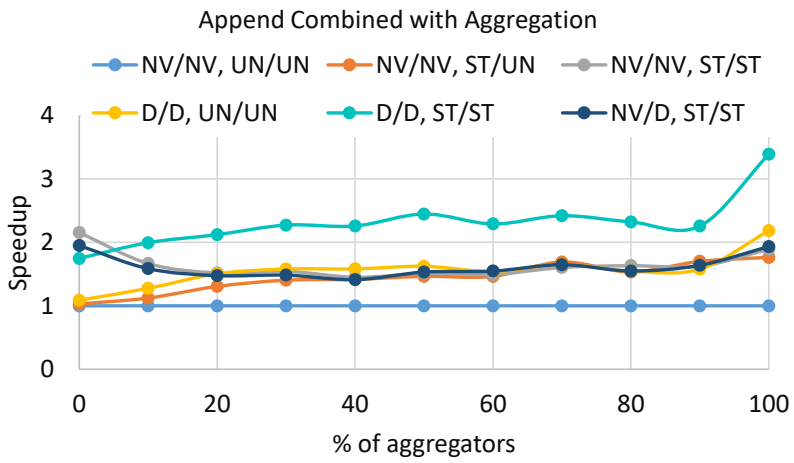


Figure 4.8: Relative speedup of a mixed workload with a specified ratio of appenders and aggregators for various 2-replica schemes and a thread count of 48.

Mixed Workloads. The next experiment is designed to show the impact of the suggested optimizations on more sophisticated workloads when writers and readers coexist. In particular, 48 threads are ran on a single socket, while the data is replicated on both. Each thread can execute either the append or the `aggregate`-operator, whereby each of them is chosen with a specified probability. Once a thread is done, it chooses a new task in the same way. The results are reported in Figure 4.8 and show the speedups relative to the purely uncompressed persistent memory-only setting at the same ratio of appenders and aggregators. The first observation is that the corner cases, i.e., 0% (append-only) and 100% (aggregate-only) reflect the results of the non-mixed workloads, i.e., either single-threaded append or 48-threaded aggregation, discussed above. Secondly, one can see that in the range 20-90% of aggregators all the alternatives behave similar and give 50% speedup on average. However, there is one outlier: when both replicas are compressed with static BP and placed in DRAM, an increase of up to 250% is demonstrated within the same interval.

Star Schema Benchmark (SSB). This paragraph focuses on the SSB [San16], a synthetic benchmark of 13 complex analytical queries from the business intelligence domain. For the *end-to-end* evaluation of the suggested *PCR* mechanism, an experiment that executes all 13 queries repeatedly in a random fashion with equal probabilities was designed, while all the primary data columns are duplicated using the 2-replica variants. To avoid any possible hardware-induced contention, only the small thread counts of 1 and 3 (for only single-socket accesses), and 6 threads (for balanced execution with 3 threads per socket) are deployed. The corresponding measurements are illustrated by Figure 4.9. Since a typical SSB query consists of dozens of operators and only few of them access the actually replicated primary data, it is assumed that the impact of *PCR* mechanisms is somewhat amortized by the other intermediate steps. Nevertheless, the experiment reveals that overall improvements of 20%, 25% and 20% are reachable for thread counts of 1, 3, and 6, respectively, compared to the uncompressed NVRAM-only setting. It is also clearly visible that any alternative allocation outperforms the basic one. The best improvement is given by the compressed DRAM-only scheme, as SSB queries leverage the higher performance of volatile memory as well as reduced amount of base data. Finally, for the considered thread counts, a sublinear scalability is reached.

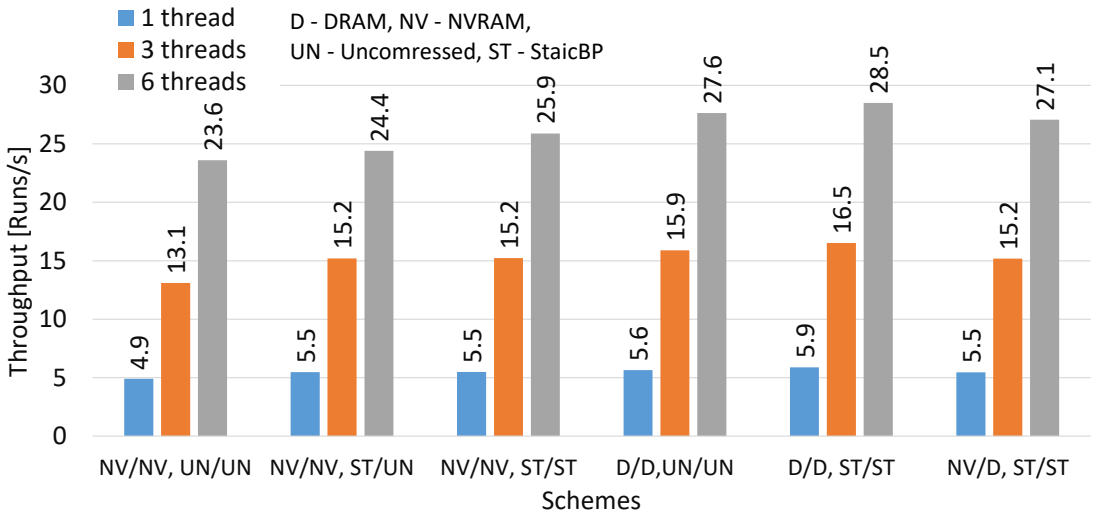


Figure 4.9: Absolute throughput for SSB queries executed for various 2-replica schemes and thread counts of 1, 3 (on one socket), and 6 (3 per socket).

4.2.4 Conclusions

Presented above evaluation verified that *PCR* mechanism is useful and can be employed in a resiliency-aware columnar data processing system on scale-up hybrid memory architectures. This holds not only for basic cases with simple access patterns, but also for highly complex analytical workloads. Employing *PCR*, speedups of up to 2.5x could be achieved compared to the uncompressed NVRAM-only case. However, the benefits depend on the operators, read/write ratio, and the data characteristics. Moreover, with proposed *PCR* mechanism, it is possible to lower the amount of NVRAM writes to address the endurance problem. Thus, PCR component is able to reduce both the runtime overhead and the endurance problem caused by the replication, while at the same time facilitating efficient analytical query processing. And, as such, is a suitable means to efficiently tackle the majority of challenges addressed in the thesis – namely enumerated as RC2-RC5 in Section 2.6.

4.3 SIMD-MIMD COCKTAIL TO SPEED UP QUERY PROCESSING

Previous section focused on the replica *allocation policies* and specific *data formats* to guarantee the resilience *and* speed up query processing, thus mostly relying on the storage component (essentially the byte-addressable memories) of hybrid memory database system. In contrast, this section investigates a compute-enabled optimization of the query operators. As underlined in Section 2.1.4, all the data (not only replicas) processed by the database operators in targeted systems are stored on byte-addressable mediums which are directly exposed to the available CPU instructions. And, as already discussed in Section 2.3, *single instruction multiple data* instruction set extensions (SIMD) is a state-of-the-art optimization technique in in-memory databases. It is most heavily deployed for columnar data organization and typically applied to isolated query operators [AMF06, PRR15, ZR02, DUP+20, RAB+13, MPM17, LMF+16]. Moreover, usefulness of SIMD instructions for physical data replication was already demonstrated in Section 3.2.1 as a way to reduce the respective runtime overhead.

It is important to stress that all considerations and optimizations suggested further could be referred to any type of columnar data (according to the underlying system model of Section 4.1) placed in DRAM or NVRAM and, thus, are not limited to replicas only. In the following particular emphasis is given to multiple instruction multiple data (MIMD) execution. This is motivated by the highly parallel nature of modern analytical processing systems, that normally serve hundreds of clients concurrently.

Since it is not well yet investigated how such a SIMD-MIMD interplay could be leveraged efficiently in hybrid memory systems – on the one hand, this section delivers an extensive experimental evaluation of typical workloads on columnar data. Moreover, it shows that the throughput of concurrent queries can be boosted (up to 2x) when combining various SIMD flavors in a multi-threaded execution. On the other hand, to enable that optimization, adaptive *SIMD-MIMD cocktail* approach incurring only a negligible runtime overhead is suggested and evaluated.

4.3.1 Motivation for SIMD-MIMD interplay

As a particular case of the hybrid memory system described in Section 4.1 this part relies on data analytical tools. Such tools, e.g., interactive dashboards, are usually deployed on top of data systems. The tasks of those data systems are to persistently manage the data and to execute simple analytical queries over the data. As recently shown [VML+19, VHF+18], simple queries like `SELECT MIN(a), MAX(b) FROM r` are issued millions of times, e.g., to populate drop down fields in the dashboard. Similarly, queries like `SELECT SUM(a) FROM r WHERE b = const` are used to calculate tailored aggregates. These analytical queries typically access a small number of columns or attributes, but a high number of rows and are, thus, most efficiently processed using a columnar data organization [BMK99, BZN05, DUP+20, SAB+05]. Nevertheless, the multitude of these queries demand two major performance requirements of columnar data systems: (i) a high query throughput, since analyses are performed by many users concurrently, and (ii) a low query latency, since analyses are expected to be interactive. Requirement (i) is commonly addressed by leveraging the well-known *multiple instruction multiple data (MIMD)* parallel paradigm, also known as thread-level parallelism (cf. Section 2.1.2). Here, each query is processed by an individual thread. Addressing requirement (ii) typically involves the *single instruction multiple data (SIMD)* parallel paradigm. Here, a single SIMD instruction processes multiple data elements at once, thereby increasing the single-thread performance. Both techniques have been available in x86-processors for many years, and their combination is a logical necessity to address the need of an overall high throughput with low query latencies across the board. To satisfy aforementioned demands scale-up hybrid memory systems is a natural hardware fit. Precisely, they facilitate both MIMD and SIMD parallelism, while supporting highly-performant in-memory only processing (cf. Section 2.1.2). Since SIMD usage in database domain is already discussed in Section 2.3, next paragraph motivates the deployment of MIMD paradigm.

MIMD Parallel Processing Opportunities. MIMD is a heavily used optimization technique in in-memory column-stores, whereby two approaches can be distinguished. On the one hand, MIMD is used to realize a *data-partitioned intra-operator* parallelism. Here, every column is partitioned into data chunks that are exclusively processed by one operator. More precisely, every operator is parallelized through a set of spawned sub-operators [PR20]. These sub-operators are mapped to designated threads and every thread is assigned to a specific column partition. This OpenMP-like processing style is typically used for operators with a sequential memory access pattern and does not require any sophisticated controlling mechanism, since there are mostly no data or control

flow dependencies between processed partitions. On the other hand, MIMD is utilized for a *multi-threaded inter-query* parallelism. This approach maps one query to one thread at any point in time and may also employ *snapshot isolation (SI)* [CG16] on the software level, to allow for a high degree of parallelism. As discussed in Section 4.1, in SI, queries atomically take a snapshot of all data columns they need to access during their prologue phase. From there on, the queries do not see any changes made to those columns after that point in time. To leverage potential caching of shared data in this case, it is common to batch a number of queries that are touching the same columns [MGAK16, GMAK14, PAA13]. Then, the execution of this query batch is triggered at once. Usually, all involved query operators are vectorized using the newest SIMD instruction set extension such as AVX-512 [PR20]. However, this SIMD-MIMD interplay is sub-optimal as shown in the next section which focuses exclusively on such inter-query approach to MIMD.

4.3.2 Experimental analysis

After motivating SIMD-MIMD interplay, this section is devoted to its experimental analysis for concurrent query execution in a hybrid memory system. Based on this experimental analysis, next section subsequently proposes an approach to select a suitable SIMD-MIMD cocktail for a particular query workload.

Evaluation Setup. The evaluation platform of this section is a two-socket hybrid memory NUMA system equipped with Intel Xeon Platinum 8276L (Scalable Cascade Lake family) processors, 384 GiB DDR4 DRAM, and 1.5 TiB Intel Optane DC Persistent Memory. Only CPUs of a single socket are used, i.e., 18 physical cores (36 with HyperThreading). Besides scalar processing, each core provides the following Intel SIMD instruction set extensions: SSE with 128-bit, AVX2 with 256-bit, and AVX-512 with 512-bit vector registers. The server runs Fedora 27 with kernel version 5.4.45 (CPU governor is set to "performance"), and `g++ 8.3.0` with -O3 flag was used for compilation. The NVRAM chunks are allocated using memory mapped files (PMDK-style) on XFS file system.

The columns in DRAM as well as NVRAM are filled with 100 M 64-bit integer values (763MiB) uniformly drawn from the interval [1, 1M]. The evaluation focuses on simple analytical queries which execution plans are compound of the following operators (introduced in Section 4.1): `aggregate`-operator, `select`-operator and `project`-operator.

As already mentioned, the output of each operator is an intermediate result and, thus, written to DRAM. The input columns could be from DRAM or NVRAM. Aside from these read-intensive operators, this section also investigates the *write-only* `append`-operator. As commonly done in big data and data warehouse applications, new values are added to the end of the columns by such `append`-operator [CD97].

All operators are implemented using the specific SIMD abstraction library *TVL* for column-stores [UPD+20][3]. Based on that, it is possible to automatically derive variants for the different Intel SIMD extensions as well as a scalar variant. In the experiments, both the SIMD and scalar operator variants are investigated. Finally, the performances are reported in terms of *runs per second*, i.e., how many times the particular operator was executed by all threads within 1 second (averaged through a 1 minute execution period).

[3] https://github.com/MorphStore/TVLLib

Impact of Vectorization. To analyze the interplay of SIMD and MIMD, this part investigates how operators behave in a concurrent setting, when they are vectorized with different SIMD extensions. In detail, the scalar execution, AVX2, and AVX-512 extensions are considered. The results for SSE are omitted, as they mostly exhibit a similar behavior as scalar or AVX2 vectorization. In addition to the different SIMD variants, the number of cores (or the number of operators that are executed simultaneously) are also varied. In particular, the typical use case where all concurrent operators access a single shared column is examined.

`Aggregate`-operator: The throughput of this operator is illustrated in Figures 4.10-(a,b) for base columns stored in DRAM and NVRAM, respectively. The first observation is that the execution behavior differs significantly for the two memory types. For NVRAM, the expected behavior is observed – larger registers imply higher performance, even with increasing thread count. When all cores are active the difference between SIMD variants is no longer observed, most likely through memory bandwidth saturation. The DRAM case, while showing significantly higher throughput (reflecting the higher read bandwidth limits), yields surprising results. Primarily, one can see that the NVRAM-like large register's domination is only visible until a certain concurrency level (12 threads). Afterwards, the efficiency leadership is taken over by smaller counterparts, e.g., by scalar starting from 18 threads. Apart from this, it is observed that performance gains are also possible until full CPU occupancy of 36 threads. It is assumed that these effects are induced by the caching of large registers and the resulting contention, which could not be reached in case of NVRAM due to differences between persistent and volatile memory controllers [IYZ+19] and slower cache replacement/trashing.

`Select`-operator: The next operator under test is the `selection`. Here, the two degrees of selectivity (1 % and 10 % of qualifying elements) are distinguished as they obviously impact the resulting memory access pattern. The first case is shown in Figures 4.10-(c,d). Since a sequential access pattern is dominating here, one can mostly confirm the observations made for `aggregation`. The most notable difference is the decreased level of performance variations between SIMD variants on DRAM, while the opposite is observed for NVRAM. AVX-512 loses its domination on volatile memory later, just starting from 24 threads onwards, while the scalar implementation is not able to significantly outperform AVX2 at all. The situation changes with the increase of the selectivity percentage.

Figures 4.10-(e,f) demonstrate the case of 10%. Now, more data has to be written, which changes the pressure on the memory controller. This becomes even more severe in multi-threaded scenarios. As a result, one can see a general performance drop for both memory types, compared to their 1% counterparts. Furthermore, the impact of the employed SIMD version to vectorize the operator becomes less significant, however large registers are still preferable in most cases. Lastly, it is observed that a slight performance increase is still reachable until full CPU occupancy for both mediums.

`Project`-operator: The last read-balanced operator in the analysis is `projection`. Figures 4.10-(g,h) demonstrate the case of data/positions size ratio equaling to 10%, while positions are unsorted and uniformly distributed. In such a scenario, the output column would contain 10 times more elements than the input column. The key difference from a memory access pattern point of view, compared to the previous cases, lies in the randomness of reads when extracting the targeted elements. As already reported by previous research [PR20], such stochastic data accesses can diminish the performance advantage of using large registers. Indeed, such behavior is confirmed for both memory types and all concurrency levels. While insignificant advantages of smaller registers could still be detected on DRAM, there is virtually no difference between the SIMD versions on NVRAM. However, the general performance of persistent memory execution drops reflecting its pure random read latency and bandwidth limits. These limits also determine the thread

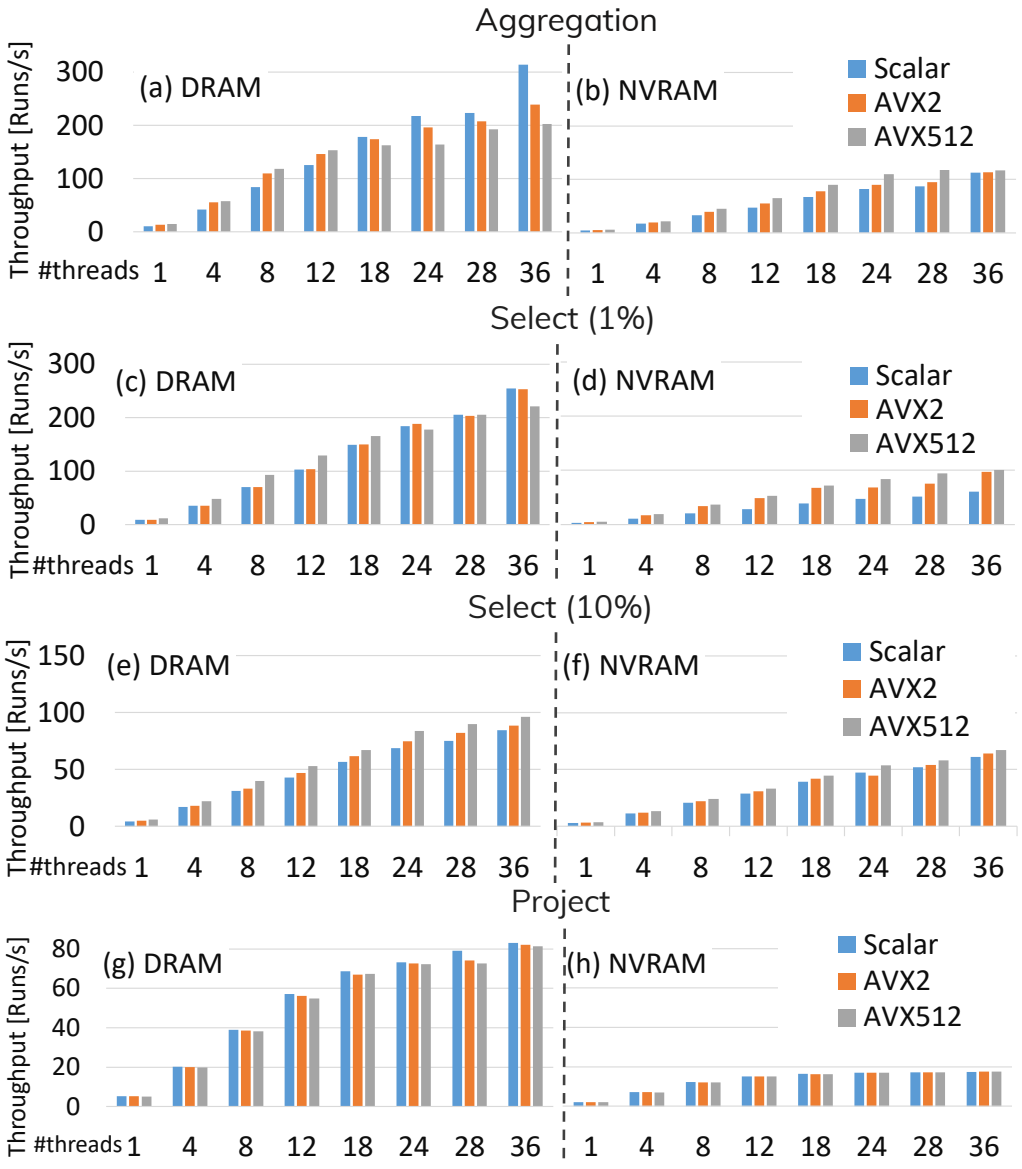


Figure 4.10: The performance of read-intensive operators for various SIMD options and two memory types.

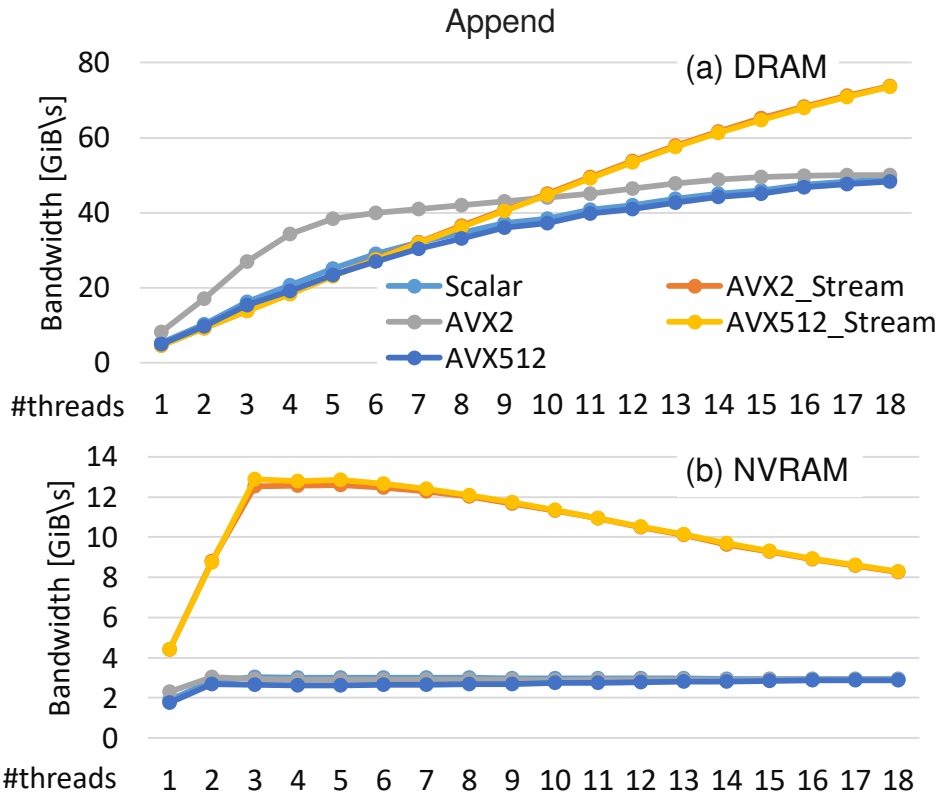


Figure 4.11: The performance of the append-operator for various SIMD options and both memory types.

counts after which performance gains are not feasible any more (e.g., from 18 threads for NVRAM).

Append-operator: As already mentioned, with regards to data modifications, this section focuses exclusively on the append-operator. The respective experiments are illustrated by Figures 4.11-(a,b). Here, the synthetically generated data from CPU registers is forwarded to be *sequentially* stored at the end of the targeted columns, whereby each thread appends to an individual column. Neither cache leveraging nor `memcpy()` operations are involved. The first observation is that there is a huge difference between the two memory types, not only in terms of performance. While DRAM prefers AVX2 as the fastest write mechanism for a moderate concurrency level, there is almost no difference between SIMD flavors on NVRAM. However, what really matters for NVRAM is the streaming style [Int18] of store operations, as they are able to deliver up to 5x better throughput (e.g., for a thread count of 3), compared to the non-streaming counterparts. Another crucial result of these experiments is that there are break-even points where one implementation hands over its leadership to another. For instance, from 10 threads onward, AVX-512 with streaming store is the fastest for DRAM. Moreover, there are most-performing points after which the performance starts to drop. For example, from 4 threads onward, the performance of streaming stores for NVRAM deteriorates.

SIMD-heterogeneous MIMD Execution. Since the underlying system model assumes the use of MIMD in form of an inter-query parallelism, it is possible to use *distinct* SIMD extensions (or scalar processing) per thread, e.g., simultaneously run scalar and AVX-512 schemes, possibly on the same data columns. This combination of heterogeneous

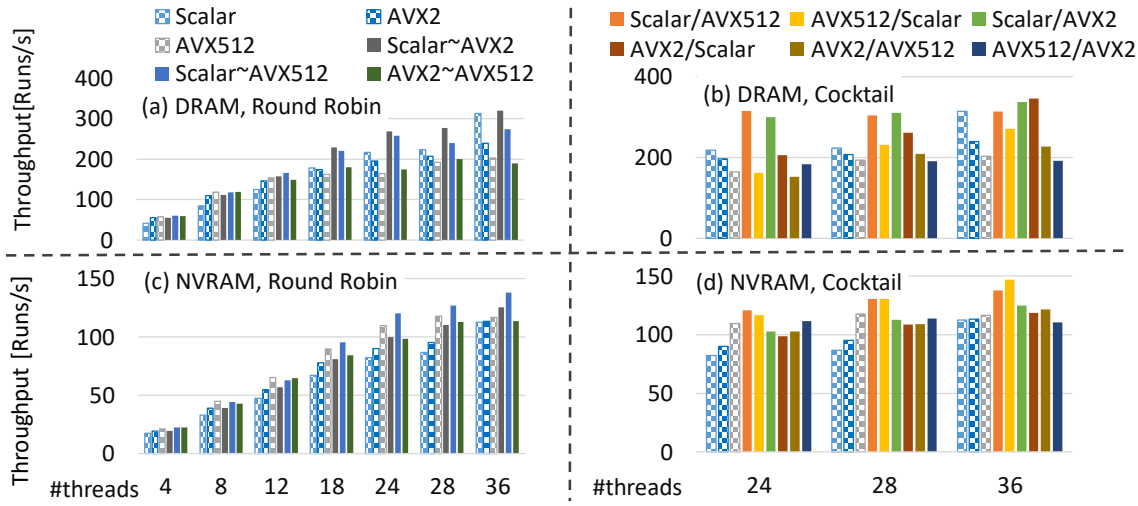


Figure 4.12: The performance of `aggregate`-operators for various homogeneous and mixed SIMD options.

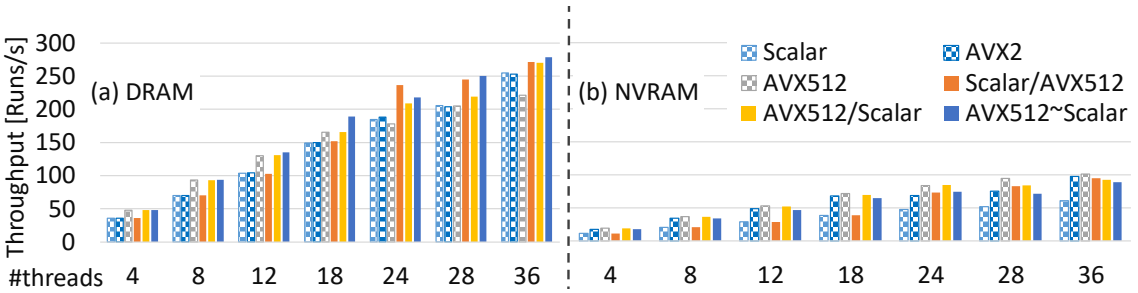


Figure 4.13: The performance of `select`-operators (1%) for various homogeneous and mixed SIMD options.

SIMD-MIMD parallelization is worth investigating, as the usage of AVX-512 registers for multiple threads leads to a reduction of the CPU's clock frequency and, thus, diminished performance gains. Hence, this part is devoted to the respective experimental examination.

The evaluation design space involves *all distinct combinations* of available SIMD extensions (or scalar processing) per thread, multiplied by the number of used threads. For illustration purposes, the analysis is limited to a selection of options, combining two SIMD flavors at a time. The following abbreviations for such *shaking* mechanisms are used:

SIMD1~SIMD2: means that the employed cores are evenly (e.g., one-by-one) distributed between two SIMD options in a *round robin* (RR) or *alternating* fashion.

SIMD1/SIMD2 is the combination that could be logically called as a sliced *cocktail*. It devotes the first half of all employed CPUs, i.e., CPU1 through CPU18 or all *physical* cores, to the SIMD1 vectorization flavor (reflecting the "bottom" slice). If more than 18 cores are used for the query execution, the SIMD2 flavor is filled on top for these additional resources. This allows to vary the *ratio* between both flavors.

`Aggregate`-operator: Figures 4.12-(a-d) show the results for both *RR* and *cocktail* style on both DRAM and NVRAM. The thread counts below 24 are omitted for the latter,

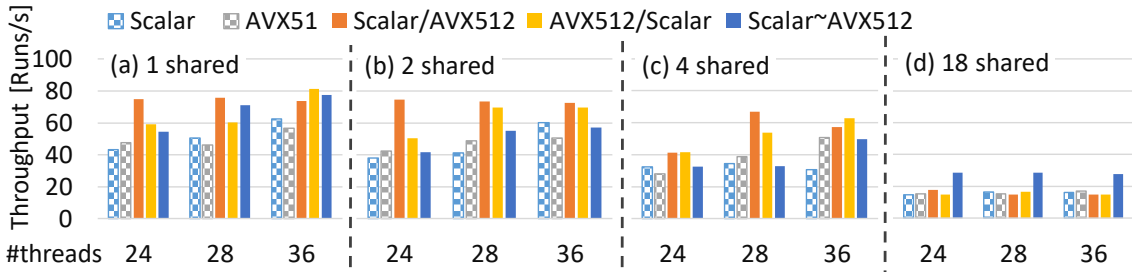


Figure 4.14: DRAM performance of `aggregate`-operators for various SIMD options and different # shared columns.

as they perform identically to the previously shown pure SIMD variants. Most importantly, it is revealed that the homogeneous SIMD alternatives (depicted as pattern filled bars) can be outperformed by the suggested combinations for all considered experimental setups. However, the particular throughput improvement varies depending on the shaking mechanism (i.e., RR or cocktail), concurrency level and memory class. For instance, on DRAM, for an intermediate level of CPU occupancy (18 threads) the round robin "Scalar~AVX2" shaking yields a 29% increase compared to the best (Scalar) homogeneous approach, while NVRAM favors the "Scalar~AVX512" scheme with 5% speedup (over "AVX512") for the same case. For higher degrees of concurrency, proposed cocktail shaking mechanism shows a surprising behavior. On DRAM, for 24 concurrent threads, "Scalar/AVX512" is able to double the performance of the pure AVX-512 implementation. Interestingly, the opposite approach "AVX512/Scalar" is by far not the best on volatile memory, while it is the superior scheme for persistent memory - delivering a 10% improvement for 36 threads.

`Select`-operator: Comparable behavior is detected for the `select`-operator, but only for small selectivities, i.e., only a few percent. While varying the selectivity on NVRAM, it is observed that no throughput increase through shaking SIMD flavors was possible for any selectivity higher than 1%. It is assumed that this is due to lower performance of writes that are proportional to the selectivity degree. Thus, Figure 4.13 depicts the measurements for 1% selectivity on both memory types. Though the mixed performance gains are lower than for the `aggregation`-operator, they still provide significant improvement, e.g., 25% for "Scalar/AVX512" compared to the best homogeneous option on DRAM for 24 threads.

Impact of number of shared columns: These facts lead to the assumption that such unexpected performance gains most likely depend on the way how various SIMD extensions interact with the caching subsystem, which can in turn favor sophisticatedly downclocked physical or hyper cores. Thus, the actual impact of caching via experiments by using different numbers of very large shared columns is investigated. The `aggregation`-operator on volatile memory is depicted by Figures 4.14-(a-d) for a single, two, four, and eighteen shared columns, respectively. The actual data size was tripled, i.e., increased to 2.3 GiB per column, to reduce spatial locality influence. The previously detected behavior of SIMD mixtures (e.g., possible superior performance) combined with *sequential access* operators is preserved for up to 18 shared columns per 36 queries, i.e., 1 column is shared by 2 threads. However, the level of throughput diversity between mixed schemes is decreasing with the number of shared columns. With 18 columns, only the round robin scheme considerably outperforms the homogeneous vectorization and, thus, yields a performance increase. The data access pattern exhibits crucial importance as well, as for random-read or heavily read-write mixed operators (e.g., project or select with a high degree of selectivity), the actual mixed performance increase tends to disappear even for the single shared data scenario.

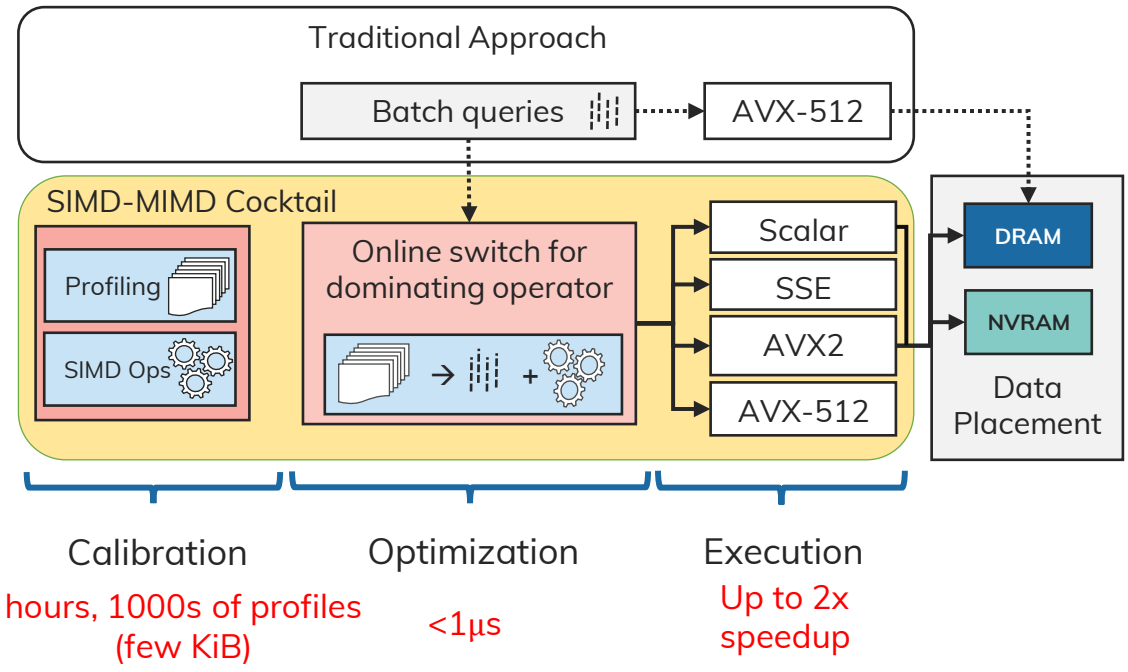


Figure 4.15: Deployment model (yellow box) of the adaptive SIMD-MIMD vectorization cocktail.

Conclusions. From the experimental analysis of typical data intensive workloads the following important conclusions are drawn: (1) Given the assumptions of the data processing system model, the employment of SIMD parallelism can significantly improve the performance compared to a scalar execution. (2) However, it is important to carefully select among the available SIMD options (including scalar processing) to reach the highest performance. A naïve strategy of always selecting the largest registers, i.e., AVX-512, may be even harmful compared to the scalar execution. (3) Furthermore, even within one instruction set, different ways to store data (e.g., streaming) can have a significant impact. (4) The best-performing SIMD option depends on the level of concurrency, the memory type, and the operator (or access pattern). (5) The newly discovered effects of SIMD-cocktails offer a great opportunity for performance optimization, and the following section presents a strategy to leverage them effectively.

4.3.3 Optimizing SIMD-MIMD interplay

As described above, batching of queries in a multi-threading environment is a common optimization technique to benefit from caching effects. This batch of queries is executed at once and all operators are typically vectorized using one specific SIMD extension such as AVX-512 in DRAM as shown in Figure 3.11. However, as clearly shown in the previous section, the choice of the optimal SIMD version applied for operator vectorization depends on various factors such as the degree of MIMD parallelism, memory type, access pattern and level of shared (and, therefore, potentially cached) data processing. In particular, Figures 4.11, 4.12, and 4.13 indicate that some vectorization schemes exhibit a significantly higher performance compared to others for certain fixed thread counts. Based on those observations, this section now proposes and evaluates an optimization design for an online decision mechanism as shown in Figure 4.15 to shake the best fitting cocktail for the current conditions at runtime.

Table 4.1: An example profile for <aggregate> on <NVRAM> @ <24> threads over <1> shared column.

SIMD Version	#Core Begin	#Core End	Round Robin
Scalar	1	18	No
AVX2	–	–	–
AVX-512	19	24	No

Optimization Design. Proposed above online decision mechanism extends the existing query optimizer of a database system. The main task of a query optimizer is to translate a descriptive SQL query into an efficient query execution plan consisting of several operators. To shake the best fitting cocktail for the operators under the current conditions, suggested approach consists of the following three components:

Profiles. Similarly to the optimization design of Section 3.2.2 the profiling data is used to abstract optimizations of specific hardware platform. The profiles characterize the behavior of particular operators (or access patterns). Such *platform-dependent* profiles are generated *once* at deployment time and provide information about the performance of the individual vectorization schemes (either homo- or heterogeneous) in the context of various concurrency levels and data set configurations. The profiles build the foundation for the adaptive optimization mechanism. Essentially, one profile per combination <operator> x <medium> x <#threads> x <#shared columns> is retrieved. For illustrative purposes, in the following discussion stays with the measurements of Section 4.3.2, which provide it with profile information for the test system. The tabular format is used as shown in Table 4.1, which reflects the case of 24 threads in Figure 4.12-(b). Thus, a single profile consists of several rows corresponding to the available SIMD extensions in the system, while each row specifies the enumeration interval of cores that has to run accordingly vectorized queries/operators to produce an optimal cocktail. The example in Table 4.1 tells database that it should use scalar processing for threads 1–18 and AVX-512 for threads 19–24. Furthermore, the profile indicates that the cocktail-style is to be favored (for round robin, the corresponding cell would contain the "Yes" indicator). This format allows for profiling of multiple SIMD versions (e.g., more than two) used in the cocktail.

Model. As high performance is among the highest priorities of hybrid memory database systems, the switching algorithm is required to be very lightweight to keep its overhead as low as possible. Due to this essential requirement, a small lookup formula/function was adopted as a model for the selection algorithm. This formula is derived based on the measurements obtained at deployment time. Essentially it returns the profile best fitting to the current conditions, and is defined as: `Profile = F(<memory>, <operator>, <#threads>, <#shared columns>)`.

Online Switching. The actual online switching component leverages the information about the current workload (or batch) including queries, memory type and the data set configuration to calculate the aforementioned formula and navigate to the recommended profile. This step imposes only a few microseconds of fixed runtime overhead and thanks to the batching, so there is no need to carry out online monitoring. The batches are executed one after the other and contain all necessary information. Subsequently, that profile is used to find out the optimal SIMD-configuration of vectorized operators to be executed by MIMD-parallel queries. Here, by workload information the optimization approach understands the dominating operator (in terms of runtime) within the currently executed query and data set, which provides information about the number and size of shared data columns and the respective memory type. The complete procedure can be executed in user-space and is possible with user-space knowledge, yet it imposes only negligible overhead due to its simplicity and is usually completely amortized by the improvements of the query execution runtime.

Implementation. A proof-of-concept of the switching approach was implemented using the columnar query engine *MorphStore* [DUP+20, HDU+19]. The schematic view of the implementation is shown in Figure 4.15. The advantage of using *MorphStore* is that its operators are implemented in a hardware-oblivious way using the vector abstraction library TVL [UPD+20]. Thus, each operator implementation can automatically be specialized to different SIMD extensions, which is required for the overall approach. In the proof-of-concept, all alternative vectorized operators that can be possibly chosen are precompiled as isolated functions that can be called by the online decision mechanism. As illustrated in Figure 3.11, the concurrent queries accessing the same base columns are batched [MGAK16, GMAK14, PAA13].

Such a batch determines the characteristics of the workload (the concurrency level, memory type, and the dominating access pattern (i.e., operator)) that are given as parameters or can be extracted during query compilation phase without any significant overhead. A decision formula as the switching model was generated based on the profile information. Obviously, this model is *platform-dependent* and needs to be calibrated for different hardware platforms. However, this calibration has to be done only once per platform.

Evaluation. To show the efficiency and applicability of suggested optimization, this part now presents selective experimental results and discusses the deployment costs. The evaluation setup is the same as in Section 4.3.2, but targets more complex scenarios, not just single operators. Thus, the behavior of adaptive SIMD-conscious vectorization is analyzed exemplified by three selected queries similar to those mentioned in Section 4.3.1:

(1) `SELECT SUM(a) FROM r`

This query resembles the basic aggregation case involving a single shared column analyzed in Section 4.3.2.

(2) `SELECT SUM(a), SUM(b), SUM(c), SUM(d) FROM r`

Here, the aggregated data set is extended to four separate equally sized columns being scanned in a sequence in accordance with the column-at-a-time processing model.

(3) `SELECT SUM(a) FROM r WHERE x < const`

This query consists of three operators: selection, projection and aggregation. Both selection and projection (of different instances of this query) access shared base data, however, the superior domination of selection was ensured by using a selectivity of 1% on a large base column.

The resulting performance of the optimization mechanism is reflected by Figure 4.16 for both mediums. Here, for query (1) the decision mechanism is always able to select the best SIMD vectorization scheme among the ones considered (Figures 4.16-(a,b)). This is, however, expected as the query execution can be exactly mapped to the aggregation profile and demonstrates the respective performance. The situation slightly changes with regard to query (2) which features its specific data set, while preserving the absolute domination of the `aggregation`-operator. According to the optimization assumption, the switching model again deploys the profile of elementary aggregation. The respective decisions result in a strong correlation with the best reachable performance on both DRAM and NVRAM (Figure 4.16-(c,d)). Although the actual best measured strategy is not always selected by the model (e.g., for thread count 4, 8 and 36 on DRAM), the respective performance losses compared to the optimum do not exceed a few percents. Finally, the

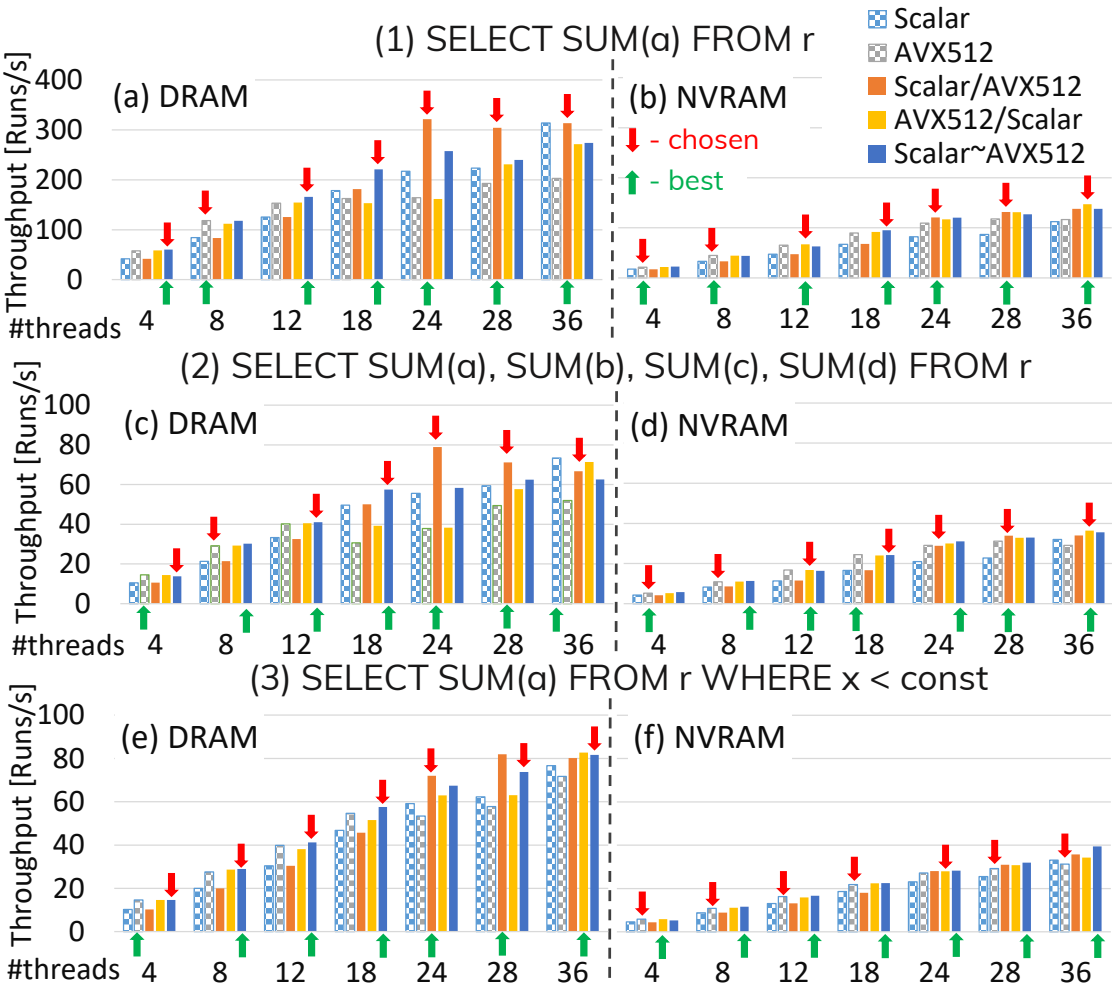


Figure 4.16: The performance of test queries indicating best measured and the selected SIMD combination.

behavior of query (3) is presented by Figures 4.16-(e,f) for volatile and persistent memory, respectively. As mentioned before, the selection part of this data processing task dominates within its runtime and, therefore, the optimization model deploys the profile of elementary selection (i.e., extracted from Figures 4.10-(c,d)), the remaining operators follow their default SIMD policies. The DRAM case demonstrates a high hit rate with the best measured option (only thread counts 1, 28 and 36 are mispredicted) with a worst-case loss of 10% compared to the best setting (e.g., for 28 threads). Nevertheless, the approach chosen in this case still outperforms the best respective basic scheme (Scalar) by 11%. The NVRAM-backed adaptive selection faces much worse correlation with the optimal measured scheme. However, this is not an issue for this scenario as all involved SIMD schemes demonstrate roughly similar throughput here, due to the slower nature of persistent memory. Thus, the under-gained performance does not exceed 7% (except for a single outlier at 36 threads).

Deployment Costs. The deployment of the proposed optimization imposes the following costs: (i) implementation of alternatively SIMD vectorized operators (though possibly automated using the TVL); (ii) one-time profile calibration overhead at system deployment phase – several hours and a few KiB of memory space to store thousands of measured profiles; (iii) a few microseconds of runtime overhead spent in the model for online switching.

4.3.4 Conclusions

Based on the presented evaluation, it could be concluded that suggested adaptive SIMD-conscious vectorization approach is applicable and useful. In nearly all examined cases, it is able to suggest either the optimal or an only slightly sub-optimal solution. While the performance delivered that way is able to considerably outpace the static homogeneous strategies (e.g., always using AVX-512), the runtime overhead is negligible. It is important to note that this approach remains useful even if only basic SIMD vectorization schemes are explicitly allowed as, in principle, it will just limit respective decision making algorithms to the corresponding profiles (with the goal to select the best among the homogeneous options).

Therefore, for both such settings the SIMD-MIMD cocktail optimization is able to speed up the query processing for targeted scenarios on both DRAM- and NVRAM-resident data (obviously applicable not only to replicas) and, thus, contribute to the solution of the overall challenge RC5 addressed in the thesis (cf. Section 2.6).

4.4 SUMMARY

This chapter contributed two innovative approaches which allow to employ the software-managed physical replication not only for the base data protection, but also for the query processing operations. According to the provided discussion, this opportunity is now enabled by the specific properties of scale-up hybrid memory systems. Essentially, such applications are able to persistently store and to efficiently process data exclusively in *byte-addressable* memory, which is in turn physically distributed over distinct NUMA sockets. Those properties activate the *immediate* on-demand usage of the replicated information within the single-box machine without any significant buffering and I/O overhead. Based on that consideration, this chapter started with the description of the underlying

data processing system model (cf. Section 4.1), and then proposed two options for the replication-related query execution optimizations.

The first contributed approach, that was called *polymorphic compressed replication (PCR)* (Section 4.2), leveraged the *storage* component (via diversity in data layouts). This mechanism united four optimization concepts that allowed not only to reduce the replication space overhead but also to exploit the replicas for analytical processing: (i) concurrent read access to replicas, (ii) compressed replication, (iii) polymorphic replication, and (iv) hybrid memory placement. Essentially, *PCR* represents the replicas using lightweight compression algorithms to reduce NVRAM writes, while different compressed formats are supported for the replicas of a single column to facilitate different database operations during query processing. Thus, the overall challenges RC2-RC5 of this thesis were addressed here. Furthermore, the suggested ideas and optimizations were illustrated based on the proof-of-concept columnar data processing system and integrated as a separate abstract user-space library. Presented evaluation verified that *PCR* mechanism is useful and can be beneficially employed in a resiliency-aware columnar data processing system on targeted architectures. That was demonstrated not only for the basic cases with simple access patterns, but also for highly complex analytical workloads. In particular, *PCR* provides speedups of up to $2.5x$ compared to the state-of-the-art execution. However, the benefits depend on the operators, read/write ratio, and the data characteristics. Moreover, PCR component was able to reduce both the runtime overhead and the endurance problem caused by the replication, while at the same time facilitating efficient analytical query processing and improving hardware utilization.

The second key contribution of this chapter is the technique called *SIMD-MIMD cocktail* (cf. Section 4.3). This approach proposed to shift optimization focus from the *storage* side to its *compute* counterpart (via diversity in available vectorized instruction set extensions). As all the data (including replicas) processed by the database operators in targeted systems is stored in byte-addressable memory – it is directly accessible by the available CPU instructions. This includes the data level parallelism or *single instruction multiple data (SIMD)* extensions, which could be now deployed for optimization goals similarly to DRAM-backed in-memory databases (to tackle the overall research challenge RC5). Thus, this chapter also investigated the usefulness of SIMD instructions for query processing on replicas (and not only). It provided the detailed experimental analysis of the vectorized behavior of typical database operators and inferred several important design decision rules to be respected when leveraging SIMD in hybrid memory database systems. Furthermore, those observations inspired a novel concurrent SIMD-heterogeneous query execution model *(SIMD-MIMD cocktail)* to overcome the traditional SIMD-homogeneous performance in MIMD-concurrent environments. This cocktail approach suggested to mix various SIMD flavors for vectorization of concurrent queries executed on shared data. As it was subsequently evaluated – speedups of up to $2x$ are achievable that way, compared to SIMD-homogeneous execution. Similarly to the first optimization, respective proof-of-concept component was evaluated within MorphStore – a prototype of an in-memory query processing system for columnar data developed at the Chair of Databases of the TU Dresden.

CONCLUSION

The in-memory database systems adopting a columnar storage model play a crucial role with respect to both analytical and transactional data processing. While information is completely kept in-memory by these systems for efficiency, data has to be stored on a non-volatile medium for persistence and fault tolerance as well. Traditionally, slow block-level devices like HDDs or SSDs are used which, however, can be replaced by fast byte-addressable NVRAM nowadays. Thus, hybrid memory systems consisting of DRAM and NVRAM offer a great opportunity for column-oriented database systems to persistently store and to efficiently process data exclusively in main-memory. However, possible DRAM and NVRAM failures still necessitate the protection of primary data. While data replication is a suitable means, it drops the performance and stresses the NVRAM capacity and endurance issues through increased write activities. Therefore, the research of how to optimize data replication for modern highly scalable machines is plentiful. Moreover, it suggests best practices not only for reduction of replication overheads, but also for leveraging replication for query processing needs.

5.1 SUMMARY

This thesis investigated the problem of data reliability in scope of novel scalable hybrid memory database architectures and its implications for the query processing by respective systems. Therefore, the required hardware and software foundations affecting hybrid memory databases are provided in Chapter 2. This also included low-level performance evaluation of NVRAM as a key enabler of single-level hybrid memory layout. Further, the ultimate need for the protection of NVRAM-resident data was motivated and a survey on potentially applicable resilience techniques was provided. As a result, the software-managed data replication was chosen as the most applicable approach. However, the evaluation of respective state-of-the-art implementation demonstrated prohibitively high runtime and space overheads motivating further research on possible optimizations (addressed in Chapter 3).

Chapter 4 elaborates on conceptual ideas that suggest to efficiently use compute node-local physical replication not only for data protection, but also for query processing needs. This is inspired by the specific properties of scale-up hybrid memory database systems that are able to persistently store and to efficiently process data exclusively in byte-addressable memory, which is in turn distributed over distinct NUMA sockets. Further, proposed in this part mechanisms and optimizations are illustrated based on the proof-of-concept columnar data processing system that incorporates the ideas of compressed replication and NUMA-aware replica placement of Chapter 3, as well. This approach was called *polymorphic compressed replication (PCR)* and integrated as an abstract user-space library into MorphStore – a prototype of an in-memory query processing system for columnar data developed at the Chair of Databases of the TU Dresden. PCR represents the replicas using lightweight compression algorithms to reduce NVRAM writes, while different compressed formats are supported for the duplicates of a single column to facilitate different database operations during the query processing. Based on this prototype, the conducted experiments shown the effectiveness (e.g., execution speedups of up to $2.5x$) of the proposed storage-centric techniques.

Up to this point, the investigations focused on the replica allocation policies and specific data formats to guarantee the resilience and speed up query processing, thus mostly relying on the storage component of hybrid memory database system. To be complete, Chapter 4 also researched the compute-enabled optimization of the query processing on replicated data. Namely, it shifted the research focus to single instruction multiple data instructions set extensions (SIMD) and studied how they could be leveraged efficiently

in MIMD-concurrent environments (natural for scale-up systems). Subsequent experiments delivered the detailed evaluation of such a SIMD-MIMD interplay. This examination revealed that the performance of concurrent query execution in certain cases could be boosted (up to $2x$) when mixing various SIMD flavors, compared to the traditional SIMD-homogeneous strategy. That observation was further used to implement a proof-of-concept optimization system *SIMD-MIMD Cocktail* within MorphStore framework.

5.2 FUTURE RESEARCH DIRECTIONS

This section aims to indicate the possible directions for the further research fitting into the expanded scope of this thesis. The main goal, already achieved by this study, is provision of the efficient and flexible data replication mechanisms for hybrid memory databases within the *compute node-local* scenario (resolved challenges RC1–RC5). Thus, they tackle the reliability and respective performance issues within the single-socket machines. To achieve strongest resiliency and support high availability (cf. Section 2.5) in case of complete server crash scenario – it is natural to consider the remote machine expansions of the proposed techniques and solutions (cf. Chapter 3 and Chapter 4).

The state-of-the-art network interface that is exploited in in-memory database systems is *remote direct memory access (RDMA)* [ZYSK19]. Such RDMA-enabled network controllers feature higher performance compared to the traditional Ethernet solutions. This is achieved via *direct* memory accesses that bypass CPU and avoid unnecessary buffering throughout the traditional data transfer stack. Moreover, RDMA technology supports the byte-addressable remote data transfers that makes it particularly advantageous in the field of hybrid memory databases (where all data is stored on byte-addressable mediums). Therefore, the RDMA-enabled connection is considered to be the primary interface for high-speed transmissions in such systems. Particularly, these interconnects could be leveraged for the remote machine replication scenario [TG19, ZYSK19, WJP17].

When applying ideas of this thesis to the compute node-remote replication case, two important research challenges could arise:

1. The first prospective challenge is the provision of low-overhead mechanism that ensures the immediate persistence of the replicated data on the remote machine and, thus, guaranties the data durability and strong consistency. As with the current RDMA API (verbs) it could be non-trivial or prohibitively expensive to ensure at fine granularity level that the transmitted data reached the remote persistent domain [DLL+21, KAK20].

2. The second future research challenge is concerned with the compressed replica formats of PCR concept. The idea here is to push the (de)compression operations down to the RDMA-enabled network controllers. Such approach potentially could free CPU for other useful workloads, however it would require advanced networking hardware capable of doing necessary for (de)compression computational operations on-the-fly while transmitting the data (e.g., smart network cards or programmable switches) [TMS20, TSJ+21].

BIBLIOGRAPHY

[AAP+17] Raja Appuswamy, Angelos Anadiotis, Danica Porobic, Mustafa Iman, and Anastasia Ailamaki. Analyzing the impact of system architecture on the scalability of OLTP engines for high-contention workloads. *PVLDB*, 11(2):121–134, 2017.

[ABH+13] Daniel Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. The design and implementation of modern column-oriented database systems. *Found. Trends Databases*, 5(3):197–280, 2013.

[ALR+17] Mihnea Andrei, Christian Lemke, Günter Radestock, Robert Schulze, Carsten Thiel, Rolando Blanco, Akanksha Meghlan, Muhammad Sharique, Sebastian Seifert, Surendra Vishnoi, Daniel Booss, Thomas Peh, Ivan Schreter, Werner Thesing, Mehul Wagle, and Thomas Willhalm. Sap hana adoption of non-volatile memory. *Proc. VLDB Endow.*, 10(12):1754–1765, 2017.

[ALRL04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[AMF06] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682, 2006.

[APP16] Joy Arulraj, Matthew Perron, and Andrew Pavlo. Write-Behind Logging. *PVLDB*, 10(4):337–348, 2016.

[AvR18] Alfons Kemper Thomas Neumann Takushi Hashida Kazuichi Oe Yoshiyasu Doi Lilian Harada Sato Mitsuru Alexander van Renen, Viktor Leis. Managing Non-Volatile Memory in Database Systems. In *SIGMOD*, pages 691–706, 2018.

[BAKS16] Shahriar Bagheri, Alireza Akhoondi Asadi, Witold Kinsner, and Nariman Sepehri. Ferroelectric random access memory (fram) fatigue test with arduino and raspberry pi. In *2016 IEEE International Conference on Electro Information Technology (EIT)*, pages 0313–0318, 2016.

[BCB16] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-Juergen Boehm. Makalu: Fast Recoverable Allocation of Non-volatile Memory. In *OOPSLA*, pages 677–694, 2016.

[BHF09] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD*, pages 283–296, 2009.

[BKM08] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12):77–85, 2008.

[BLP11] Spyros Blanas, Yinan Li, and Jignesh M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *SIGMOD*, pages 37–48, 2011.

[BMK99] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, pages 54–65, 1999.

[Bon00] André B. Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd International Workshop on Software and Performance*, WOSP '00, page 195âĂŞ203, New York, NY, USA, 2000. Association for Computing Machinery.

[BTAÖ15] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. Main-memory hash joins on modern processor architectures. *IEEE Trans. Knowl. Data Eng.*, 27(7):1754–1766, 2015.

[Byu10] S. Byun. A design of raid-1 storage using hard disk drive and flash memory drive. 2010.

[BZN05] Peter A. Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, 2005.

[Car15] P. A. Carter. Implementing log shipping. 2015.

[CB18] Hyunkyoung Choi and Hyokyung Bahn. Accelerating storage system performances with nvram cache by considering storage access characteristics. In *2018 5th International Conference on Information Science and Control Engineering (ICISCE)*, pages 107–111, 2018.

[CD97] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1):65–74, 1997.

[CDN11] Surajit Chaudhuri, Umeshwar Dayal, and Vivek R. Narasayya. An overview of business intelligence technology. *Commun. ACM*, 54(8):88–98, 2011.

[CG16] Andrea Cerone and Alexey Gotsman. Analysing snapshot isolation. 07 2016.

[CGN11] Shimin Chen, Phillip B. Gibbons, and Suman Nath. Rethinking Database Algorithms for Phase Change Memory. In *CIDR*, pages 21–31, 2011.

[CJ15] Shimin Chen and Qin Jin. Persistent B+-Trees in Non-Volatile Main Memory. *PVLDB*, 8(7):786–797, 2015.

[CK85] George P. Copeland and Setrag Khoshafian. A decomposition storage model. In *SIGMOD*, pages 268–279, 1985.

[CLX+13] Yuanyuan Cui, M. Lou, Jianqing Xiao, Xunying Zhang, Senmao Shi, and Pengwei Lu. Research and implementation of sec-ded hamming code algorithm. *2013 IEEE International Conference of IEEE Region 10 (TENCON 2013)*, pages 1–5, 2013.

[Com] LZ4 Community. Lz4. https://lz4.github.io/lz4/.

[DBY+19] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and protection in the zofs user-space nvm file system. In *SOSP*, pages 478–493, 2019.

[DHHL17] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. Lightweight data compression algorithms: An experimental survey (experiments and analyses). In *EDBT*, pages 72–83, 2017.

[DHL15] Patrick Damme, Dirk Habich, and Wolfgang Lehner. A benchmark framework for data compression techniques. In *TPCTC*, pages 77–93, 2015.

[Dim21] Ultimate memory guide. https://technick.net/guides/hardware/umg/05_001/, 2021.

[DKB+19] Markus Dreseler, Jan Kossmann, Martin Boissier, Stefan Klauck, Matthias Uflacker, and Hasso Plattner. Hyrise re-engineered: An extensible database system for research in relational in-memory data management. In *EDBT*, pages 313–324, 2019.

[DKK+14] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *EuroSys*, pages 15:1–15:15, 2014.

[DLL+21] Zhuohui Duan, Haodi Lu, Haikun Liu, Xiaofei Liao, Hai Jin, Yu Zhang, and Song Wu. Hardware-supported remote persistence for distributed persistent memory. New York, NY, USA, 2021. Association for Computing Machinery.

[Don15] V. L. V. Donselaar. Low latency asynchronous database synchronization and data transformation using the replication log. 2015.

[DPVJ14] Rohit Dhamane, Marta Patiño-Martínez, Valerio Vianello, and Ricardo Jiménez-Peris. Performance Evaluation of Database Replication Systems. In *IDEAS*, pages 288–293, 2014.

[DS09] R. Davis and K. Simmons. Database mirroring overview. 2009.

[DUH+19] Patrick Damme, Annett Ungethüm, Juliana Hildebrandt, Dirk Habich, and Wolfgang Lehner. From a comprehensive experimental survey to a cost-based selection strategy for lightweight integer compression algorithms. *ACM Trans. Database Syst.*, 44(3):9:1–9:46, 2019.

[DUP+20] Patrick Damme, Annett Ungethüm, Johannes Pietrzyk, Alexander Krause, Dirk Habich, and Wolfgang Lehner. Morphstore: Analytical query engine with a holistic compression-enabled processing model. *Proc. VLDB Endow.*, 13(11):2396–2410, 2020.

[ea19] Patrick Damme et al. Lightweight compression benchmarking and selection, 2019.

[EKA19] Katembo Ezechiel, Shri Kant, and Dr Agarwal. A systematic review on distributed databases systems and their techniques. *Journal of Theoretical and Applied Information Technology*, 96, 01 2019.

[FKV+16] Xuanyao Fong, Yusung Kim, Rangharajan Venkatesan, Sri Harsha Choday, Anand Raghunathan, and Kaushik Roy. Spin-transfer torque memories: Devices, circuits, and systems. *Proceedings of the IEEE*, 104(7):1449–1488, 2016.

[Fly72] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972.

[Fuj18] Kazuhisa Fujimoto. Replication. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems, Second Edition*. Springer, 2018.

[GBB20] Mathias Gottschlag, Peter Brantsch, and Frank Bellosa. Automatic core specialization for AVX-512 applications. In *SYSTOR*, pages 25–35, 2020.

[GH93] M.D. Godfrey and D.F. Hendry. The computer as von neumann planned it. *IEEE Annals of the History of Computing*, 15(1):11–21, 1993.

[GM09] Hector Garcia-Molina. Database systems: The complete book, 2009.

[GMAK14] Georgios Giannikis, Darko Makreshanski, Gustavo Alonso, and Donald Kossmann. Shared workload optimization. *Proc. VLDB Endow.*, 7(6):429–440, February 2014.

[Goo19] Google. Snappy - a fast compressor/decompressor, 2019. https://github.com/google/snappy.

[GRS98] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. Compressing relations and indexes. In *ICDE*, pages 370–379, 1998.

[GS19] Jana Giceva and Mohammad Sadoghi. Hybrid OLTP and OLAP. In *Encyclopedia of Big Data Technologies*. 2019.

[GS20] Manuel Le Gallo and Abu Sebastian. An overview of phase-change memory device physics. *Journal of Physics D: Applied Physics*, 53(21):213002, mar 2020.

[GZC+16] Jinwei Guo, Chendong Zhang, Peng Cai, Minqi Zhou, and Aoying Zhou. Low Overhead Log Replication for Main Memory Database System. In *WAIM*, pages 159–170, 2016.

[HDU+19] Dirk Habich, Patrick Damme, Annett Ungethüm, Johannes Pietrzyk, Alexander Krause, Juliana Hildebrandt, and Wolfgang Lehner. Morphstore - in-memory query processing based on morphing compressed intermediates LIVE. In *SIGMOD Conference*, pages 1917–1920, 2019.

[Hei14] J. Heidecker. Evaluation of magnetoresistive ram for space applications. 2014.

[HHDL16] Juliana Hildebrandt, Dirk Habich, Patrick Damme, and Wolfgang Lehner. Compression-aware in-memory query processing: Vision, system design and beyond. In *ADMS@VLDB*, pages 40–56, 2016.

[HKHL18] Dirk Habich, Till Kolditz, Juliana Hildebrandt, and Wolfgang Lehner. Reliable in-memory data management on unreliable hardware. In *DATA*, pages 365–372, 2018.

[Huf52] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.

[Int09] Intel. An introduction to the intel quickpath interconnect. 2009.

[Int18] Intel. Intel instruction reference manual (vol 2a, 3-147). 2018.

[Int19a] Intel. *Intel Optane DC Persistent Memory Module*, 2019.

[Int19b] Intel. *Intel PQoS Utility*, 2019.

[IYZ+19] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane dc persistent memory module, 2019.

[KAK20] Anuj Kalia, David Andersen, and Michael Kaminsky. Challenges and solutions for fast remote persistent memory access. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, pages 105–119, New York, NY, USA, 2020. Association for Computing Machinery.

[Kap15] Tomasz Kapela. An introduction to replication, 2015. http://pmem.io/2015/11/23/replication-intro.html.

[KD18] Amandeep Khurana and Julien Le Dem. The modern data architecture: The deconstructed database. *login Usenix Mag.*, 43(4), 2018.

[KDH15] P. Koopman, K. Driscoll, and B. Hall. Selection of cyclic redundancy code and checksum algorithms to ensure critical data integrity. 2015.

[Kim15] Hideaki Kimura. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *SIGMOD*, pages 691–706, 2015.

[KKS+14a] Tim Kiefer, Thomas Kissinger, Benjamin Schlegel, Dirk Habich, Daniel Molka, and Wolfgang Lehner. ERIS live: a numa-aware in-memory storage engine for tera-scale multiprocessor systems. In *SIGMOD*, pages 689–692, 2014.

[KKS+14b] Thomas Kissinger, Tim Kiefer, Benjamin Schlegel, Dirk Habich, Daniel Molka, and Wolfgang Lehner. ERIS: A numa-aware in-memory storage engine for analytical workload. In *ADMS@VLDB*, pages 74–85, 2014.

[Kli19] Yauhen Klimiankou. Translation lookaside buffer management. pages 20–24, 12 2019.

[KM17] Sonal Kanungo and Rustom Morena. Issues with concurrency control techniques. 06 2017.

[KMG14] Rakesh Kumar, Alejandro Martínez, and Antonio González. Efficient power gating of SIMD accelerators through dynamic selective devectorization in an HW/SW codesigned environment. *ACM Trans. Archit. Code Optim.*, 11(3):25:1–25:23, 2014.

[KSKN18] Wook-Hee Kim, Jihye Seo, Jinwoong Kim, and Beomseok Nam. clfb-tree: Cacheline friendly persistent b-tree for nvram. *ACM Trans. Storage*, 14(1):5:1–5:17, February 2018.

[KWJP16] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. Ermia: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1675âĂŞ1687, New York, NY, USA, 2016. Association for Computing Machinery.

[KZZL17] Soroosh Khoram, Yue Zha, Jialiang Zhang, and Jing Li. Challenges and opportunities: From near-memory computing to in-memory computing. In *ISDP*, pages 43–46, 2017.

[LB15] Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. *Softw., Pract. Exper.*, 45(1):1–29, 2015.

[LBK+19] Daniel Lemire, Leonid Boytsov, Owen Kaser, Maxime Caron, Louis Dionne, Michel Lemay, Erik Kruus, Andrea Bedini, Matthias Petri, Robson Braga Araujo, and Patrick Damme. The FastPFOR C++ library: Fast integer compression, 2019.

[LBKN14] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. *SIGMOD*, 06 2014.

[LGS+14] Yixin Luo, Sriram Govindan, Bikash Sharma, Mark Santaniello, Justin Meza, Aman Kansal, Jie Liu, Badriddine Khessib, Kushagra Vaid, and Onur Mutlu. Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous-reliability memory. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 467–478, 2014.

[lib20] Libnuma, 2020.

[LLRP14] J. Lindstrom, Kyosti Laiho, V. Raatikka, and J. Parkkinen. Replication for hot standby online database. 2014.

[LMF+16] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In *SIGMOD*, pages 311–326, 2016.

[Loh08] Gabriel H. Loh. 3d-stacked memory architectures for multi-core processors. *SIGARCH Comput. Archit. News*, 36(3):453–464, June 2008.

[MAK+13] Iulian Moraru, David G. Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. Consistent, Durable, and Safe Memory Management for Byte-addressable Non Volatile Main Memory. In *TRIOS@SOSP*, pages 1:1–1:17, 2013.

[MGAK16] Darko Makreshanski, Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. Mqjoin: Efficient shared execution of main-memory joins. *Proc. VLDB Endow.*, 9(6):480–s491, January 2016.

[MMNLM20] Diana Martinez-Mosquera, Rosa Navarrete, and Sergio Lujan-Mora. Modeling and management big data in databases - a systematic literature review. *Sustainability*, 12(2), 2020.

[MPM17] Prashanth Menon, Andrew Pavlo, and Todd C. Mowry. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proc. VLDB Endow.*, 11(1):1–13, 2017.

[MZHS17] Zhixiang Mao, Shengan Zheng, Linpeng Huang, and Yanyan Shen. A dax-enabled mmap mechanism for log-structured in-memory file systems. *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8, 2017.

[ndc21] Ndctl utility guide. https://docs.pmem.io/persistent-memory/getting-started-guide/what-is-ndctl, 2021.

[Nvr20] Intel optane dc persistent memory: Quick start guide. https://www.intel.com/content/dam/support/us/en/documents/memory-and-storage/data-center-persistent-mem/Intel-Optane-DC-Persistent-Memory-Quick-Start-Guide.pdf, 2020.

[OBL+14] Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, and Thomas Willhalm. SOFORT: A Hybrid SCM-DRAM Storage Engine for Fast Data Recovery. In *DaMoN*, pages 8:1–8:7, 2014.

[OL17a] Ismail Oukid and Wolfgang Lehner. Data structure engineering for byte-addressable non-volatile memory. In *SIGMOD*, pages 1759–1764, 2017.

[OL17b] Ismail Oukid and Wolfgang Lehner. Towards a Single-Level Database Architecture on Non-Volatile Memory. In *NVMW*, 2017.

[OLN+16] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In *SIGMOD*, pages 371–386, 2016.

[PAA13] Iraklis Psaroudakis, Manos Athanassoulis, and Anastasia Ailamaki. Sharing data and work across concurrent analytical queries. *Proc. VLDB Endow.*, 6(9):637–648, July 2013.

[Pac11] Peter S. Pacheco. Chapter 2 - parallel hardware and parallel software. In Peter S. Pacheco, editor, *An Introduction to Parallel Programming*, pages 15 – 81. Morgan Kaufmann, Boston, 2011.

[PALG19] Kyriakos Paraskevas, Andrew Attwood, Mikel Luján, and John Goodacre. Scaling the capacity of memory systems; evolution and key approaches. In *MEMSYS*, pages 235–249, 2019.

[PIL+19] Onkar Patil, Latchesar Ionkov, Jason Lee, Frank Mueller, and Michael Lang. Performance characterization of a dram-nvm hybrid memory architecture for hpc applications using intel optane dc persistent memory modules. In *MEMSYS*, pages 288–303, 2019.

[PJHA] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. Data-oriented transaction execution. *Proc. VLDB Endow.*, 3(1):928–939.

[PLH21] Hyeon Woo Park, Jae-Gil Lee, and Cheol Seong Hwang. Review of ferroelectric field-effect transistors for three-dimensional storage applications. *Nano Select*, 2(6):1187–1207, 2021.

[PM13] Iro Pantazi-Mytarelli. The history and use of pipelining computer architecture: Mips pipelining implementation. In *2013 IEEE Long Island Systems, Applications and Technology Conference (LISAT)*, pages 1–7, 2013.

[PR14] Orestis Polychroniou and Kenneth A. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *SIGMOD*, pages 755–766, 2014.

[PR20] Orestis Polychroniou and Kenneth A. Ross. VIP: A SIMD vectorized analytical query engine. *VLDB J.*, 29(6):1243–1261, 2020.

[PRR15] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. Rethinking SIMD vectorization for in-memory databases. In *SIGMOD*, pages 1493–1508, 2015.

[PSM+16] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. Adaptive numa-aware data placement and task scheduling for analytical workloads in main-memory column-stores. *PVLDB*, 10(2):37–48, 2016.

[QSR09] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. Association for Computing Machinery.

[RAB+13] Vijayshankar Raman, Gopi K. Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent Kulandai Samy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, René Müller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam J. Storm, and Liping Zhang. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.

[Rud15] Andy Rudoff. Persistent Memory Programming. *Login: The Usenix Magazine*, 42:34–40, 2015.

[RVH93] Mark A. Roth and Scott J. Van Horn. Database compression. *SIGMOD Rec.*, 22(3):31–39, 1993.

[SAB+05] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-store: A column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[San16] Jimi Sanchez. A review of star schema benchmark. *CoRR*, abs/1606.00295, 2016.

[SBF+15] David Schwalb, Tim Berning, Martin Faust, Markus Dreseler, and Hasso Plattner. nvm malloc: Memory allocation for nvram. In *ADMS@VLDB*, 2015.

[Sca20] Steve Scargall. *Introduction to Persistent Memory Programming*, pages 1–10. Apress, Berkeley, CA, 2020.

[SDB+15] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. Memory errors in modern systems: The good, the bad, and the ugly. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 297–310, New York, NY, USA, 2015. ACM.

[SN10] Jinsun Suk and Jaechun No. File system snapshot. *Journal of the Institute of Electronics Engineers of Korea*, 47:88–95, 2010.

[Sni17] Nvm programming model. https://www.snia.org/sites/default/files/technical_work/final/NVMProgrammingModel_v1.2.pdf, 2017.

[SV10] Fabrizio Silvestri and Rossano Venturini. Vsencoding: efficient coding and fast decoding of integer lists via dynamic programming. In *CIKM*, 2010.

[TG19] Jan Marian Michalski Tomasz Gromadzki. Persistent memory replication over traditional rdma, 2019.

[TMS20] Maroun Tork, Lina Maudlej, and Mark Silberstein. Lynx: A smartnic-driven accelerator-centric architecture for network servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 117–131, New York, NY, USA, 2020. Association for Computing Machinery.

[TSJ+21] Lasse Thostrup, Jan Skrzypczak, Matthias Jasny, Tobias Ziegler, and Carsten Binnig. Dfi: The data flow interface for high-speed networks. In *SIGMOD/PODS '21: Proceedings of the 2021 International Conference on Management of Data*, pages 1825–1837. ACM, June 2021.

[UPD+20] Annett Ungethüm, Johannes Pietrzyk, Patrick Damme, Alexander Krause, Dirk Habich, Wolfgang Lehner, and Erich Focht. Hardware-oblivious SIMD parallelism for in-memory column-stores. In *CIDR*, 2020.

[VHF+18] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. Get real: How benchmarks fail to represent the real world. In *DBTest@SIGMOD*, pages 1:1–1:6, 2018.

[Vig14] Stratis Viglas. Write-limited sorts and joins for persistent memory. *PVLDB*, 7(5):413–424, 2014.

[VML+19] Adrian Vogelsgesang, Tobias Mühlbauer, Viktor Leis, Thomas Neumann, and Alfons Kemper. Domain query optimization: Adapting the general-purpose database system hyper for tableau workloads. In *BTW*, pages 313–333, 2019.

[VTRC11] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*, pages 5–5, 2011.

[WJP17] Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. Query fresh: Log shipping on steroids. *Proc. VLDB Endow.*, 11(4):406–419, December 2017.

[WNC87] Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Commun. ACM*, 30(6):520–540, June 1987.

[WPB+09] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2(1):385–394, 2009.

[WRK+10] H. Wong, S. Raoux, Sangbum Kim, J. Liang, J. Reifenberg, B. Rajendran, M. Asheghi, and K. Goodson. Phase change memory. *Proceedings of the IEEE*, 98:2201–2227, 2010.

[XL11] Lihao Xu and Jianqiang Luo. Hyfs: design and implementation of a reliable file system. 2011.

[XZM+17] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *SOSP*, pages 478–496, 2017.

[YJC+16] Jie Yao, H. Jiang, Q. Cao, Lei Tian, and C. Xie. Elastic-raid: A new architecture for improved availability of parity-based raids by elastic mirroring. *IEEE Transactions on Parallel and Distributed Systems*, 27:1044–1056, 2016.

[YMC+11] Doe Hyun Yoon, Naveen Muralimanohar, Jichuan Chang, Parthasarathy Ranganathan, Norman P. Jouppi, and Mattan Erez. Free-p: Protecting non-volatile memory against both hard and soft errors. In *HPCA*, pages 466–477, 2011.

[YWW+16] Jun Yang, Qingsong Wei, Chundong Wang, Cheng Chen, Khai Leong Yong, and Bingsheng He. NV-Tree: A Consistent and Workload-Adaptive Tree Structure for Non-Volatile Memory. *IEEE Trans. Computers*, 65(7):2169–2183, 2016.

[YXD+15] Songping Yu, Nong Xiao, Mingzhu Deng, Yuxuan Xing, Fang Liu, Zhiping Cai, and Wei Chen. WAlloc: An efficient wear-aware allocator for non-volatile main memory. In *IPCCC*, pages 1–8, 2015.

[YXD+17] Songping Yu, Nong Xiao, Mingzhu Deng, Fang Liu, and Wei Chen. Redesign the memory allocator for non-volatile main memory. *J. Emerg. Technol. Comput. Syst.*, 13(3), 2017.

[Zar14] Mikhail Zarubin. Petri nets for modelling and calculations. Master's thesis, Norwegian University of Science and Technology, Trondheim, May 2014. https://emecs.eit.uni-kl.de.

[ZDHL20] Mikhail Zarubin, Patrick Damme, Dirk Habich, and Wolfgang Lehner. Polymorphic compressed replication of columnar data in scale-up hybrid memory systems. In *SYSTOR*, pages 98–110, 2020.

[ZDK+19] Mikhail Zarubin, Patrick Damme, Thomas Kissinger, Dirk Habich, Wolfgang Lehner, and Thomas Willhalm. Integer compression in nvram-centric data stores: Comparative experimental analysis to DRAM. In *DaMoN 2019*, pages 11:1–11:11, 2019.

[ZDK+21] Mikhail Zarubin, Patrick Damme, Alexander Krause, Dirk Habich, and Wolfgang Lehner. SIMD-MIMD cocktail in a hybrid memory glass: shaken, not stirred, 2021.

[ZHNB06] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. Super-scalar RAM-CPU cache compression. In *ICDE*, page 59, 2006.

[ZKH+19] Mikhail Zarubin, Thomas Kissinger, Dirk Habich, Thomas Willhalm, and Wolfgang Lehner. Efficient compute node-local replication mechanisms for nvram-centric data structures. *The VLDB Journal*, 29:775–795, 2019.

[ZKHL18] Mikhail Zarubin, Thomas Kissinger, Dirk Habich, and Wolfgang Lehner. Efficient compute node-local replication mechanisms for nvram-centric data structures. In *DaMoN@SIGMOD*, DAMON '18, pages 7:1–7:9, New York, NY, USA, 2018. ACM.

[ZL77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theor.*, 23(3):337–343, 1977.

[ZLL+15] Wenzhe Zhang, Kai Lu, Mikel LujÃąn, Xiaoping Wang, and Xu Zhou. Write-combined logging: An optimized logging for consistency in nvram. *Scientific Programming*, 2015:1–13, 12 2015.

[ZR02] Jingren Zhou and Kenneth A. Ross. Implementing database operations using SIMD instructions. In *SIGMOD*, pages 145–156, 2002.

[ZS19] Lu Zhang and Steven Swanson. Pangolin: A fault-tolerant persistent memory programming library. In *ATC*, pages 897–912, 2019.

[ZYMS15] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A Reliable and Highly-Available Non-Volatile Memory System. *SIGARCH Comput. Archit. News*, 43(1):3–18, March 2015.

[ZYSK19] Erfan Zamanian, Xiangyao Yu, M. Stonebraker, and Tim Kraska. Rethinking database high availability with rdma networks. *Proc. VLDB Endow.*, 12:1637–1650, 2019.

LIST OF FIGURES

LIST OF TABLES

CONFIRMATION

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, January 3, 2022