T. Grust et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2019),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2019 477

Architectural Principles for Database Systems on Storage-Class Memory[1]

Ismail Oukid[2]

Abstract: Storage-Class Memory (SCM) is a novel class of memory technologies that combine the byte addressability and low latency of DRAM with the density and non-volatility of traditional storage media. Hence, SCM can serve as *persistent main memory*, i.e., as main memory and storage at the same time. In this thesis, we dissect the challenges and pursue the opportunities brought by SCM to database systems. To solve the identified challenges, we devise necessary building blocks for enabling SCM-based database systems, namely memory management, data structures, transaction concurrency control, recovery techniques, and a testing framework against new failure scenarios stemming from SCM. Thereafter, we leverage these building blocks to build SOFORT, a novel hybrid SCM-DRAM transactional storage engine that places data, accesses it, and updates it directly in SCM, thereby doing away with traditional write-ahead logging and achieving near-instant recovery.

Keywords: Non-Volatile Memory, Storage-Class Memory, Indexing, Transaction Processing, Memory Management, Testing, Recovery.

1 Introduction

Database systems have long been optimized to hide the higher latency of storage media, yielding complex persistence mechanisms. With the advent of large DRAM capacities, it became possible to keep a full copy of the data in DRAM. Systems that leverage this possibility, such as main-memory databases, keep two copies of the data: one in main memory and the other one in storage. The two copies are kept synchronized using snapshotting and logging. This main-memory-centric architecture yields orders of magnitude faster analytical processing than traditional, disk-centric architectures. The rise of Big Data emphasized the importance of such systems with an ever-increasing need for more main memory. However, DRAM is hitting its scalability limits: It is intrinsically hard to further increase its density.

Storage-Class Memory (SCM) is a group of novel memory technologies that promise to alleviate DRAM's scalability limits. They combine the non-volatility, density, and economic characteristics of storage media with the byte-addressability and a latency close to that of DRAM. Therefore, SCM can serve as *persistent main memory*, thereby bridging the gap between volatile main memory and persistent storage. In this thesis, we explore the impact

[1] This paper is a summary of the author's PhD thesis of the same title.

[2] TU Dresden & SAP SE, Database Systems Group, Nöthnitzer Str. 46, D-01062 Dresden, ismail.oukid@sap.com

doi:10.18420/btw2019-29

of SCM as persistent main memory on the design and architecture of database systems. Assuming a hybrid SCM-DRAM hardware setting, we propose a novel software architecture for database systems that places primary data in SCM and directly operates on it, eliminating the need for traditional I/O. This architecture yields many benefits: First, it obviates the need to reload data from storage to main memory during restart, as data is discovered and accessed directly in SCM. Second, it allows replacing the traditional logging infrastructure by fine-grained, cheap micro-logging techniques at data-structure level. Third, secondary data can be stored in DRAM and efficiently reconstructed during recovery. Fourth, system runtime information can be stored in SCM to improve recovery time. Finally, the system may retain and continue in-flight transactions in case of system failures.

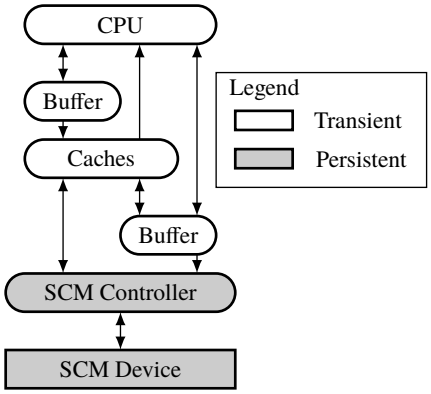

Fig. 1: Volatility chain in x86-like processors.

Unfortunately, SCM is no panacea as it raises unprecedented programming challenges. Given its byte-addressability and low latency, processors can access, read, modify, and persist data in SCM using load/store instructions at a CPU cache line granularity. The path from CPU registers to SCM is long and mostly volatile, as illustrated in Figure 1, including store buffers and CPU caches, leaving the programmer with little control over when data is persisted. Therefore, there is a need to enforce the order and durability of SCM writes using persistence primitives, such as cache line flushing instructions. This in turn creates new failure scenarios, such as missing or misplaced persistence primitives.

Within this thesis, we devise several building blocks to overcome these challenges [OL17]. First, we tackle memory management, as the first required building block to build a database system, by designing a highly scalable SCM allocator, named PAllocator [Ou17a], that fulfills the versatile needs of database systems (Section 2). Thereafter, we propose the FPTree [Ou16b], a highly scalable hybrid SCM-DRAM persistent B^+-Tree that bridges the gap between the performance of transient and persistent B^+-Trees (Section 3). Using these building blocks, we realize our envisioned database architecture in SOFORT [Ou14, Ou15], a hybrid SCM-DRAM columnar transactional engine. We propose an SCM-optimized MVCC scheme that eliminates write-ahead logging from the critical path of transactions (Section 4). Since SCM-resident data is near-instantly available upon recovery, the new recovery bottleneck is rebuilding DRAM-based data. To alleviate this bottleneck, we propose a novel recovery technique that achieves nearly instant responsiveness of the database by accepting queries right after recovering SCM-based data, while rebuilding DRAM-based data in the background [Ou17b] (Section 5). Additionally, SCM brings new failure scenarios that existing testing tools cannot detect. Hence, we propose an online testing framework that is able to automatically simulate power failures and detect missing or misplaced persistence primitives [Ou16a] (Section 6). In summary, our proposed building blocks can serve to devise more complex systems, paving the way for future database systems on SCM.

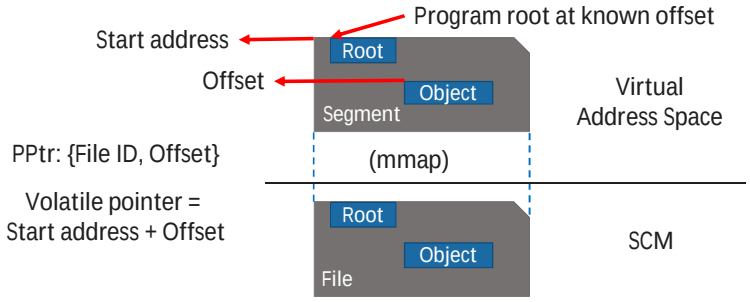

Fig. 2: Data recovery using persistent pointers consisting of a file ID and an offset within that file.

2 Persistent Memory Management

The rise of SCM might spur a major change in the architecture of database systems, as it invalidates long-standing architectural assumptions. In this incoming era of SCM-based database systems, everything is yet to be done, starting from persistent memory management as a fundamental building block. In general, SCM is handled using a file system. User space access to SCM is granted via memory mapping using *mmap()*. The mapping behaves like *mmap()* for traditional files, except that the persistent data is directly mapped to the virtual address space, instead of to a DRAM-cached copy. When a program crashes, its pointers become invalid since the program gets a new address space when it restarts. This implies that these pointers cannot be used to recover persistent data structures.

To solve this problem, we propose a new pointer type, denoted Persistent Pointer (PPtr), that remains valid across restarts. It consists of a base, which is a file ID, and an offset within that file that indicates the start of the block pointed to. Persistent pointers can easily be translated into regular pointers by adding the offset to the start address of the memory-mapped file, as illustrated in Figure 2. To perform recovery, we need to keep track of an entry point. One entry point is sufficient for the whole storage engine since every structure is encapsulated into a larger structure up to the full engine.

To manage SCM, we propose PAllocator [Ou17a], a highly scalable, fail-safe, and persistent allocator for SCM, specifically designed for databases that require very large main memory capacities. PAllocator uses internally two different allocators: SmallPAllocator, a small block persistent allocator that implements a segregated-fit strategy; and BigPAllocator, a big block persistent allocator that implements a best-fit strategy and uses hybrid SCM-DRAM trees to persist and index its metadata. The use of hybrid trees enables PAllocator to also offer a fast recovery mechanism. PAllocator uses big SCM files that are cut into smaller blocks for allocation. It maps these files to virtual memory to enable the conversion of PPtrs to regular pointers. At restart time, a new mapping to virtual memory is created, allowing to re-convert PPtrs to new valid regular pointers.

Moreover, PAllocator addresses fragmentation in persistent memory, which we argue is an important challenge, and implements an efficient defragmentation algorithm that is able to reclaim the memory of fragmented blocks by leveraging the hole punching feature of sparse

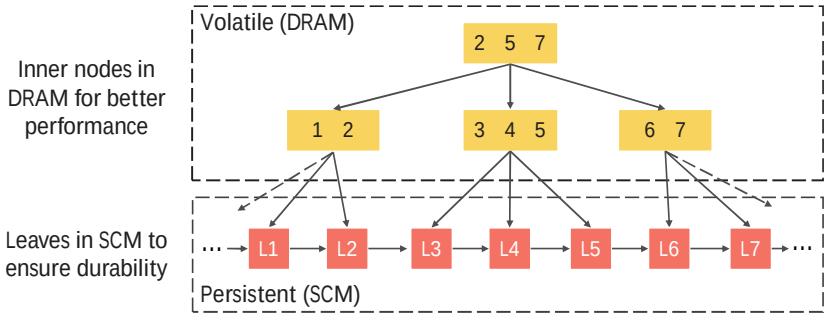

Fig. 3: Selective persistence for a B$^+$-Tree: Inner nodes are in DRAM while leaf nodes are in SCM.

files. To the best of our knowledge, PAllocator is the first SCM allocator that proposes a transparent defragmentation algorithm as a core component for SCM-based database systems. Our evaluation shows that PAllocator improves on state-of-the-art persistent allocators by up to two orders of magnitude in operation throughput, and by up to three orders of magnitude in recovery time. Furthermore, we integrate PAllocator and a state-of-the-art persistent allocator in a persistent B$^+$-Tree, and show that PAllocator enables up to 2.39x better operation throughput than its counterpart.

3 Persistent Data Structures

In this section we investigate the design of persistent index structures as one of the core database structures, motivated by the observation that traditional main memory B$^+$-Tree implementations do not fulfill the consistency requirements needed for such a use case. Furthermore, while expected in the range of DRAM, SCM latencies are higher and asymmetric with writes noticeably slower than reads. We argue that these performance differences between SCM and DRAM imply that the design assumptions made for previous well-established main memory B$^+$-Tree implementations might not hold anymore. Therefore, we see the need to design a novel, persistent B$^+$-Tree that leverages the capabilities of SCM while exhibiting performance similar to a traditional transient B$^+$-Tree.

To lift this shortcoming, we propose the *Fingerprinting Persistent Tree* (**FPTree**) [Ou16b] that is based on the following design principles to achieve near-DRAM performance:

1. **Fingerprinting**. Fingerprints are one-byte hashes of in-leaf keys, placed contiguously in the first cache-line-sized piece of the leaf. The FPTree uses unsorted leaves with in-leaf bitmaps – originally proposed in [CGN11] – such that a search iterates linearly over all valid keys in a leaf. By scanning the fingerprints first, we are able to limit the number of in-leaf probed keys to **one** in the average case, which leads to a significant performance improvement.
2. **Selective Persistence**. The idea is based on the well-known distinction between primary data, whose loss infers an irreversible loss of information, and non-primary data that can be rebuilt from the former. Selective persistence consists in storing

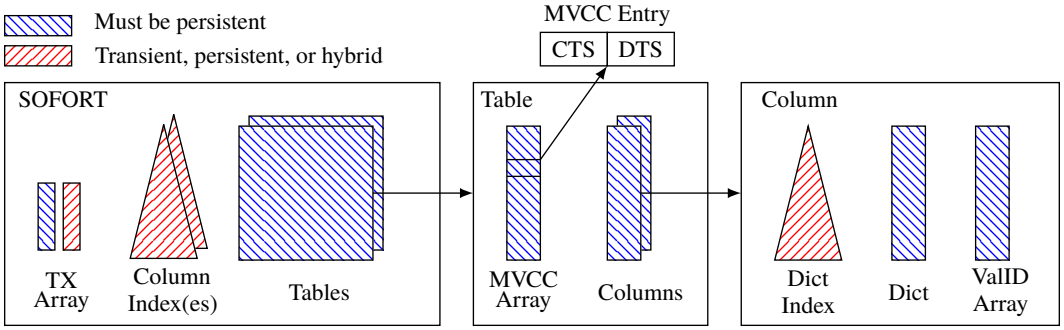

Fig. 4: Data layout overview in SOFORT.

primary data in SCM and non-primary data in DRAM. Applied to the FPTree (illustrated in Figure 3), this corresponds to storing the leaf nodes in SCM and the inner nodes in DRAM. Hence, only leaf accesses are more expensive during a tree traversal compared to a fully transient counterpart.

3. **Selective Concurrency**. This concept consists in using different concurrency schemes for the transient and persistent parts: The FPTree uses Hardware Transactional Memory (HTM) to handle the concurrency of inner nodes, and fine-grained locks to handle that of leaf nodes. Selective Concurrency elegantly solves the apparent incompatibility of HTM and persistence primitives required by SCM such as cache line flushing instructions which cause HTM transactions to abort.

We implemented the FPTree and two state-of-the-art persistent trees, namely the NV-Tree [Ya16] and the wBTree [CJ15]. Using microbenchmarks, we show that the FPTree outperforms the two competitors by up to 4.8× for an SCM latency of 90 ns (DRAM's latency), and by up to 8.2× for an SCM latency of 650 ns. The FPTree achieves these results while keeping less than 3% of its data in DRAM. Additionally, we demonstrate how the FPTree scales on a machine with 88 logical cores. Moreover, we show that the FPTree recovery time is 76.96× and 29.62× faster than a full rebuild for SCM latencies of 90 ns and 650 ns, respectively.

4 SOFORT: A Hybrid SCM-DRAM Storage Engine

After having outlined the core building blocks, namely memory management and data structures for the design and implementation of SCM-based data-management systems, we demonstrate such a system and present SOFORT, a hybrid SCM-DRAM dictionary-encoded columnar transactional engine tailored for hybrid transactional and analytical workloads and fast data recovery [Ou14, Ou15]. SOFORT is a single-level store, i.e., the working copy of the data is the same as the durable copy of the data. To achieve this, SOFORT leverages the byte-addressability of SCM to persist data in small increments at cache-line granularity. Since the database state is always up-to-date, SOFORT does not need a redo log. SOFORT implements serializable multi-version concurrency control (MVCC) coupled with cooperative garbage collection.

SOFORT is architected as a twin-store columnar main-memory database, with a larger static immutable read-optimized store and a smaller dynamic read/write store. The dynamic store is periodically merged into the static store to do compaction. This keeps the size of the dynamic store small and therefore results in good performance for reads and writes. Andrei et al. [An17] showed that the static store can be entirely kept in SCM at a negligible cost for query execution, and instantly recovered after failure. Therefore, we focus only on the dynamic part in this thesis.

Figure 4 gives an overview of the data organization in SOFORT. Tables are stored as a collection of append-only columns. Each column consists of an unsorted dictionary stored as an array of the column's unique data values, and an array of value IDs, where a value ID corresponds to a dictionary index (position). These two arrays are sufficient to provide data durability and constitute the *primary* data. SOFORT stores primary data, accesses it, and updates it directly in SCM. Other data structures are required to achieve reasonable performance including, for each column, a dictionary index that maps values to value IDs, and for each table, a set of multi-column inverted indexes that map sets of value IDs to the set of corresponding row IDs. We refer to these structures as *secondary* data since they can be reconstructed from the primary data. SOFORT can keep secondary data in DRAM or in SCM. Putting all indexes to DRAM gives the best performance but might stress DRAM resources, while placing indexes in SCM exposes them to its higher latency which compromises performance.

To track conflicts between transactions, MVCC keeps for every transaction, among other metadata, a write set that contains the row IDs of the tuples that the transaction inserted or deleted. This information is enough to undo a transaction in case it is aborted. We make the observation that the same information can be used to undo the effects of in-flight transactions during recovery. Therefore, to provide durability, SOFORT places the MVCC write set in SCM, which enables it to remove the traditional write-ahead log from the critical path of transactions. Furthermore, by persisting more MVCC metadata, SOFORT can allow the user to continue executing open transactions after a system failure.

SOFORT stores its columns contiguously in memory, which complicates memory reclamation of deleted tuples. Indeed, the latter would require a writer-blocking process which would replace the current columns with new, garbage-free ones. To remedy this issue, we propose to keep track of deleted rows and re-use them when inserting new tuples whenever possible instead of appending them to the table. Through an extensive experimental evaluation, we show that SOFORT exhibits competitive OLTP performance despite being a column-store.

5 SOFORT Recovery Techniques

SCM-enabled database systems such as SOFORT keep a single copy of the data that is stored, accessed, and modified directly in SCM. This eliminates the need to reload a consistent state from durable media to memory upon recovery, as primary data is accessed

directly in SCM. Hence, the new recovery bottleneck is rebuilding DRAM-based data structures. We address this bottleneck following two orthogonal dimensions: The first one pertains to secondary data placement in SCM, in DRAM, or in a hybrid SCM-DRAM format. We show that near-instant recovery is achievable if all secondary data is persisted in SCM. However, this comes at the cost of a decreased query performance by up to 51.1%. Nevertheless, near-instant recovery offers guarantees that are appealing to business applications where availability is critical. Hybrid SCM-DRAM data structures offer a good compromise by reducing recovery time by up to 5.9× while limiting the impact on query performance between 16.6% and 32.7%. The second dimension is optimizing the recovery of DRAM-based data independent of data placement.

After recovering its SCM-based data structures, SOFORT rebuilds DRAM-based secondary data structures, then starts accepting requests. If DRAM-based secondary data structures are large, restart times can still be unacceptably long. To address this, we propose an *Instant Recovery* strategy. It allows queries to be answered concurrently with the rebuilding of DRAM-based data structures, i.e., while recovery is still in progress. However, while the secondary data structures are being rebuilt, request throughput is reduced, partially because of the overhead of rebuilding, but more importantly because of the unavailability of the DRAM-based secondary data structures, resulting in sub-optimal access plans.

To remedy the shortcomings of instant recovery, we propose a novel recovery strategy, named *Adaptive Recovery* [Ou17b]. It is inspired by the observation that not all secondary data structures are equally important to a given workload. It improves on instant recovery in two ways. First, it prioritizes the rebuilding of DRAM-based secondary data structures based on their benefit to a workload (instant recovery simply uses an arbitrary order). Second, it releases most of the CPU resources dedicated to recovery once all of the important secondary data structures have been rebuilt (instant recovery statically splits CPU resources between recovery and query processing for the entire recovery period).

Through our experimental evaluation, we showed that SOFORT regains its peak performance up to 4.3× faster with adaptive recovery than with synchronous recovery, while allowing the system to be responsive near-instantly. We demonstrated that our benefit ranking adapts well to workload changes during recovery and allows to regain pre-failure throughput up to 2.1× faster than rankings that do not take into consideration the recovery workload.

6 Testing of SCM-Based Software

Consistency failure scenarios and recovery strategies of software that persists data depend on the underlying storage technology. In the traditional case of block-based devices, software has full control over when data is made persistent. Basically, software schedules I/O to persist modified data at a page granularity. The application has no direct access to the primary copy of the data and can only access copies of the data that are buffered in main memory. Hence, software errors can corrupt data only in main memory which can be reverted

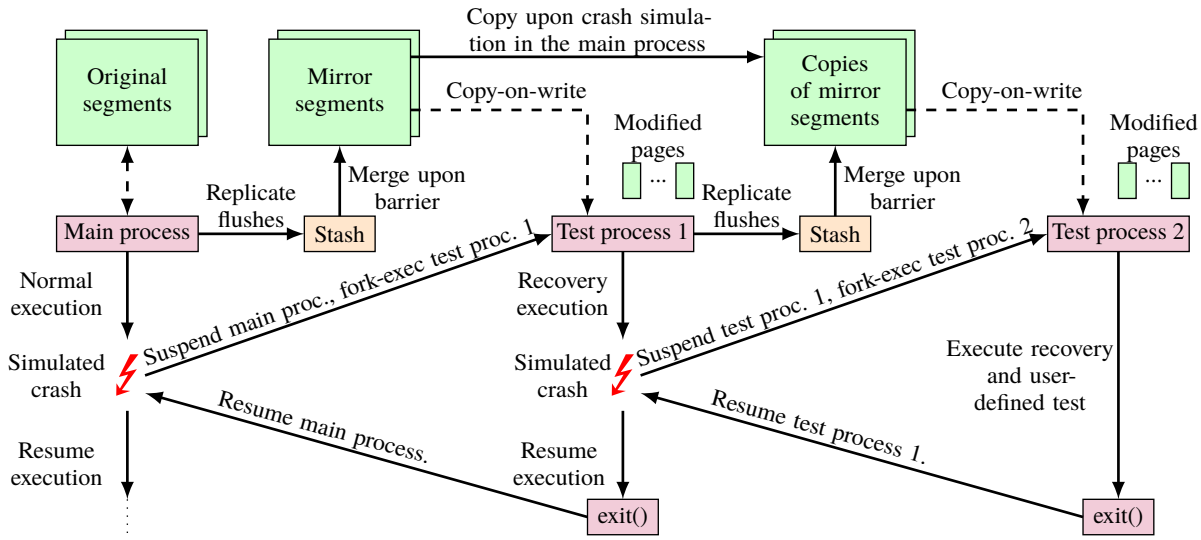

Fig. 5: Illustration of crash simulation in the testing framework.

as long as the corruption was not explicitly propagated to storage. In fact, crash-safety for block-based software highly depends on the correctness of the underlying file system. In contrast, SCM is byte-addressable and is accessed via volatile store buffers and CPU caches, over which software has little control. As a side effect, changes can be speculatively propagated from the CPU cache to SCM at any time, and compilers and out-of-order CPU execution can jeopardize consistency by reordering memory operations. Moreover, changes are made persistent at a cache line granularity which necessitates the use of CPU persistence primitives. This adds another level of complexity as enforcing the order in which changes are made persistent cannot be delayed like with block-based storage devices, and must be synchronous. Hence, storing the primary copy of the data in SCM and directly updating it in-place significantly aggravates the risk of data corruption.

Several proposals tackled these challenges following two main approaches. The first one focuses on providing global software-based solutions, mainly transactional-memory-like libraries, to make it easier for developers to write SCM-based software. The second and more mainstream approach is to rely solely on existing hardware persistence primitives, such as cache line flushing instructions and memory barriers to achieve consistency. Nevertheless, all approaches have in common that SCM-related errors may result in data corruption. In contrast to volatile RAM where data corruption can be cured with a restart of the program, data corruption in SCM might be irreversible as it is persistent. Therefore, we argue for the need of testing the correctness of SCM-based software against software crashes and power failures–which result in the loss of the content of the CPU cache.

We tackle this challenge by proposing a lightweight automated on-line testing framework, illustrated in Figure 5, that helps detecting and debugging a wide range of SCM-related bugs that can arise upon software or power failures [Ou16a]. We particularly focus on detecting missing cache line flushing instructions. Our testing framework employs a suspend-test-

resume approach and simulates power failures using data replication, similar to shadow memory testing approaches [NS07]. The testing framework creates a mirror segment for each segment that the program creates; the mirror segment contains only data that is explicitly flushed by the program. The testing framework triggers randomly simulated crashes in the path of persistence primitives, upon which a test process is forked and the main process is suspended. Then, the test process executes a user-defined program that recovers using the mirror segment with copy-on-write access semantics. Upon completion of the user-defined program, the test process is terminated and the changes to the mirror segments are discarded.

An important feature of our testing framework is its ability to avoid excessive duplicate testing by tracking the call stack information of already tested code paths, which leads to fast code coverage. Additionally, our testing framework is able to partially detect errors that might arise due to the compiler or the CPU speculatively reordering memory operations. It can further simulate crashes in the recovery procedure of the tested program, which we argue is important since hidden SCM-related errors in the recovery procedure may compromise the integrity of the data upon every restart. We show with an experimental evaluation on the FPTree and PAllocator that our testing framework achieves fast testing convergence, even in the case of nested crash simulations.

7 Conclusion

SCM is emerging as a disruptive hybrid memory and storage technology, requiring us to fundamentally rethink current database system architectures. In this thesis, we endeavored to explore this potential by building a hybrid transactional and analytical database system from the ground up that leverages SCM as persistent main memory. Our pathfinding work led us to identify the challenges and opportunities brought by SCM for database systems. As a result, we devised a set of building blocks, including an SCM allocator, a hybrid SCM-DRAM persistent and concurrent B^+-Tree, an adaptation of MVCC for SCM, novel database recovery techniques, and a testing tool for SCM-based software. Armed with these building blocks, we designed and implemented SOFORT, a hybrid SCM-DRAM transactional engine that keeps primary data in SCM and directly operates on it. We showed how SOFORT's architecture enables near-instant recovery, allows to continue unfinished transactions after failure, removes traditional write-ahead logging from the critical path of transactions, and therefore achieves low transaction latencies. These building blocks can be used to build more complex systems, exemplified by SOFORT, paving the way for future database systems on SCM.

8 Acknowledgments

I would like to warmly thank Prof. Wolfgang Lehner for his mentorship and guidance which were essential to the success of this thesis. Special thanks also go to all current and past

members of the SAP HANA Campus, the SAP HANA development team, and the Intel onsite team at SAP for their help and support throughout my PhD thesis.

References

[An17] Andrei, Mihnea; Lemke, Christian; Radestock, Günter; Schulze, Robert; Thiel, Carsten; Blanco, Rolando; Meghlan, Akanksha; Sharique, Muhammad; Seifert, Sebastian; Vishnoi, Surendra; Booss, Daniel; Peh, Thomas; Schreter, Ivan; Thesing, Werner; Wagle, Mehul; Willhalm, Thomas: SAP HANA Adoption of Non-volatile Memory. Proc. VLDB Endow., 10(12):1754–1765, August 2017.

[CGN11] Chen, Shimin; Gibbons, Phillip B; Nath, Suman: Rethinking Database Algorithms for Phase Change Memory. In: CIDR. 2011.

[CJ15] Chen, Shimin; Jin, Qin: Persistent B+-trees in Non-volatile Main Memory. Proc. VLDB Endow., 8(7):786–797, February 2015.

[NS07] Nethercote, Nicholas; Seward, Julian: Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. SIGPLAN Not., 42(6):89–100, June 2007.

[OL17] Oukid, Ismail; Lehner, Wolfgang: Data Structure Engineering For Byte-Addressable Non-Volatile Memory. In: Proceedings of the 2017 ACM International Conference on Management of Data. SIGMOD '17, ACM, New York, NY, USA, pp. 1759–1764, 2017.

[Ou14] Oukid, Ismail; Booss, Daniel; Lehner, Wolfgang; Bumbulis, Peter; Willhalm, Thomas: SOFORT: A Hybrid SCM-DRAM Storage Engine for Fast Data Recovery. In: Proceedings of the Tenth International Workshop on Data Management on New Hardware. DaMoN '14, ACM, New York, NY, USA, pp. 8:1–8:7, 2014.

[Ou15] Oukid, Ismail; Lehner, Wolfgang; Kissinger, Thomas; Willhalm, Thomas; Bumbulis, Peter: Instant Recovery for Main-Memory Databases. In: CIDR. 2015.

[Ou16a] Oukid, Ismail; Booss, Daniel; Lespinasse, Adrien; Lehner, Wolfgang: On Testing Persistent-memory-based Software. In: Proceedings of the 12th International Workshop on Data Management on New Hardware. DaMoN '16, ACM, New York, NY, USA, pp. 5:1–5:7, 2016.

[Ou16b] Oukid, Ismail; Lasperas, Johan; Nica, Anisoara; Willhalm, Thomas; Lehner, Wolfgang: FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In: Proceedings of the 2016 International Conference on Management of Data. SIGMOD '16, ACM, New York, NY, USA, pp. 371–386, 2016.

[Ou17a] Oukid, Ismail; Booss, Daniel; Lespinasse, Adrien; Lehner, Wolfgang; Willhalm, Thomas; Gomes, Grégoire: Memory Management Techniques for Large-scale Persistent-main-memory Systems. Proc. VLDB Endow., 10(11):1166–1177, August 2017.

[Ou17b] Oukid, Ismail; Nica, Anisoara; Dos Santos Bossle, Daniel; Lehner, Wolfgang; Bumbulis, Peter; Willhalm, Thomas: Adaptive Recovery for SCM-Enabled Databases. In: ADMS@VLDB. 2017.

[Ya16] Yang, J.; Wei, Q.; Wang, C.; Chen, C.; Yong, K. L.; He, B.: NV-Tree: A Consistent and Workload-Adaptive Tree Structure for Non-Volatile Memory. IEEE Transactions on Computers, 65(7):2169–2183, July 2016.