Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, Thomas Willhalm

SOFORT. A Hybrid SCM-DRAM Storage Engine for Fast Data Recovery

Erstveröffentlichung in / First published in:

SIGMOD/PODS'14: International Conference on Management of Data, Snowbird 23.06.2014. ACM Digital Library, Art. Nr. 8. 978-1-4503-2971-2

DOI: https://doi.org/10.1145/2619228.2619236

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", Art. Nr. 8, ISBN 978-1-4503-2971-2
https://doi.org/10.1145/2619228.2619236

SOFORT: A Hybrid SCM-DRAM Storage Engine for Fast Data Recovery

Ismail Oukid
Technische Universität Dresden & SAP AG
i.oukid@sap.com

Daniel Booss
SAP AG
daniel.booss@sap.com

Wolfgang Lehner
Technische Universität Dresden
wolfgang.lehner@tu-dresden.de

Peter Bumbulis
SAP AG
peter.bumbulis@sap.com

Thomas Willhalm
Intel GmbH
thomas.willhalm@intel.com

ABSTRACT

Storage Class Memory (SCM) has the potential to significantly improve database performance. This potential has been well documented for throughput [4] and response time [25, 22]. In this paper we show that SCM has also the potential to significantly improve restart performance, a shortcoming of traditional main memory database systems. We present SOFORT, a hybrid SCM-DRAM storage engine that leverages full capabilities of SCM by doing away with a traditional log and updating the persisted data in place in small increments. We show that we can achieve restart times of a few seconds independent of instance size and transaction volume without significantly impacting transaction throughput.

1. INTRODUCTION

Availability guarantees form an important part of many service level agreements (SLAs) for production database systems [7]: minimizing database downtime has economic as well as usability benefits. Database systems crash for a variety of reasons including software bugs, hardware faults and user errors. Many of these conditions are transient: for these, restarting - after logging the error - is a reasonable approach to recovery. In these cases database restart time has a significant and direct impact on database availability.

In-memory DBMSs recover by rebuilding DRAM-based data structures from a consistent state persisted on durable media. The persisted state typically consists of a copy (checkpoint) of the database state at a particular instant in time and a log of subsequent updates. Recovery consists of reloading portions of the checkpointed state, applying subsequent updates and then undoing the effects of unfinished transactions. Database restart time not only includes the time to recover the consistent state as it existed before the crash but also the time to reload any data required by the current workload. This second component can be substantial for OLAP workloads.

Storage Class Memory (SCM) is Non-Volatile Memory (NVM) with latency characteristics close to that of DRAM and density, durability and economic characteristics comparable to existing storage media. Examples of SCM are Phase Change Memory [16], Spin Transfer Torque RAM [12], Magnetic RAM [9], and Memristors [26]. Continuing advances in NVM technology hold the promise that SCM will become a reality in the near future. Current NVM technologies provide read and write latencies within an order of magnitude of DRAM, with writes being noticeably slower than reads.

In this paper we present SOFORT, a hybrid SCM-DRAM storage engine that speeds up restarts by taking advantage of the properties of SCM to operate on the persisted data directly without having to first cache it in DRAM. SOFORT also speeds up recovery by doing away with a traditional log and updating the persisted data in place in small increments. In this paper we show that, with accepted assumptions for SCM performance, SOFORT can achieve this without compromising transactional throughput. To do so, we propose a novel programming model for persistent memory. We carefully choose which structures to put on SCM and which ones to keep in DRAM. This flexibility enables performance/restart time trade-offs since DRAM is faster than SCM and the structures that are not persisted in real time on SCM need to be rebuilt at restart time. To build SOFORT, we have designed persistent, single-level, lock-free, and concurrent data structures that are self-sufficient to recover in a consistent way relative to the state of the database.

To evaluate SOFORT, we compare it with Shore-MT on ramdisk using the TATP benchmark [2]. Our evaluation shows that SOFORT has up to 4 times higher throughput than Shore-MT on ramdisk. As we do not know yet the final latencies of SCM, we consider a range of latencies and report incurred performance variations using special hardware that enables the tuning of emulated SCM latency. We show that SOFORT stays competitive even in a high latency SCM environment, recovers in seconds and is resilient to user contention.

The paper is structured as follows: In Section 2, we discuss the design of SOFORT. Section 3 describes our persistent memory programming model. We describe the core operations of SOFORT in Section 4. We evaluate SOFORT in Section 5 and discuss related work in Section 6. Finally, Section 7 concludes the paper and outlines future work.

2. SOFORT DESIGN

SOFORT is a main-memory transactional storage engine intended for mixed OLAP and OLTP workloads. To achieve good OLAP performance, tables are stored column-wise. While SOFORT has been implemented as a column store, the same principles also apply for row stores. As in other column-stores, such as SAP HANA, data is split into a larger read-optimized, read-only *main* storage and a smaller write-optimized, read-write *delta* storage, with periodic

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", Art. Nr. 8, ISBN 978-1-4503-2971-2
https://doi.org/10.1145/2619228.2619236

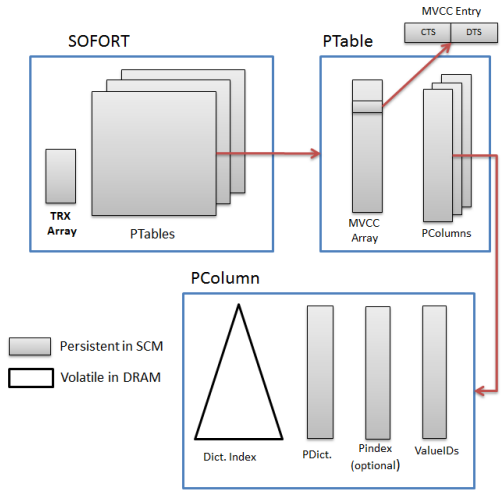


Figure 1: Overview of SOFORT.

merges from *delta* to *main* to keep the size of the *delta* bounded. Currently only the *delta* is implemented in SOFORT; implementing the *main* is straightforward as it is read-only. Given our architectural choices, we expect SOFORT to have high OLAP performance.

SOFORT uses Multi-Version Concurrency Control (MVCC) [14, 15] and dictionary encoding. In particular, every column has its own dictionary. Dictionary codes are integers called ValueIDs. Figure 1 gives an overview of SOFORT. An instance of SOFORT consists of multiple persistent tables (PTables) and a persistent array of currently running transaction objects (TRX array). In the current implementation, the size of the TRX array is fixed and bounds the maximum number of concurrent transactions. A transaction object holds information related to the state of the transaction including a transaction time-stamp (TTS). A transaction ID (TXID) is the index of a transaction in the TRX array. Each PTable encompasses several persistent columns (PColumns) and a persistent MVCC array.

There is one MVCC entry per row of a table. An MVCC entry consists of a commit time-stamp (CTS) and a deletion time-stamp (DTS) which are both initialized to ∞. Time-stamps are logical and generated starting from *sizeof*(TRX array) by a global counter. When a transaction starts, it is assigned a TTS which is equal to the current logical time. The time-stamp counter is incremented for every commit. A CTS or DTS is interpreted as a TXID if it is lower than or equal to the size of the TRX array. Based on its MVCC entry and relative to a transaction, a row is:

Visible, if the TTS is greater than or equal to the CTS and lower than the DTS. If the DTS is a transaction ID (TXID), then we compare with the commit time-stamp of the transaction pointed to by that TXID.

Invisible, if it is not visible.

Locked, if its DTS is a TXID.

A PColumn is append-only and contains a persistent dictionary array (PDict.) that maps ValueIDs to Values, supported by a dictionary index that maps Values to ValueIDs. Since not all data structures need to be persistent, we carefully choose which ones to put on SCM and which ones we keep on DRAM. In general, column structures are more bandwidth-bound while tree structures are more latency-bound. Since the dictionary index is heavily accessed and is latency-bound, we keep it on DRAM for better performance. Hence, we need to reconstruct it from the dictionary array at restart time. All the other data structures are persistent in SCM.

Figure 2 is an example of a SOFORT table of employees and their offices. It illustrates how multiple versions are managed. When a row is updated (in the example, Ingo changes office), the current version of the row is invalidated by updating the DTS of the corresponding MVCC entry, and a new version of the row is created, with its corresponding CTS equal to the DTS of the deleted row.

SOFORT is latch-free and uses atomic instructions only, except when resizing tables, where only writers are blocked. All data structures are concurrent and no centralized lock is needed. This makes SOFORT highly scalable to the number of cores and resilient to user contention. At the moment, SOFORT supports only statement level isolation.

3. PROGRAMMING MODEL

First, we discuss what persistent primitives current hardware offers. Then, we give an overview of SOFORT's memory management. Last, we detail the recovery mechanism of SOFORT.

3.1 Persistence Primitives

In our proof-of-concept, SCM is connected via PCIe. However in the future, SCM may be tightly integrated into the memory subsystem and may use the CPU's cache interface. Independent from the implementation, enforcing persistency is critical. Using the existing architecture, one could envision an implementation using non-temporal stores, CPU flushing instructions and memory barriers [3]. The primitives we use are:

CLFLUSH: Flushing Instruction. Invalidates the cache line that contains a given linear address. Writes back this cache line if it is inconsistent with memory.

MOVNT: Store Instruction. Bypasses the cache and writes directly to memory.

Modern CPUs implement complex out-of-order execution, where for example, a flushing instruction can be reordered with previous instructions, leading to the eviction of a cache line that may not take into account the new writes that should have happened before the flushing instruction. To order memory instructions, we use memory barriers. On x86 architecture, we use [3]:

SFENCE: Memory Barrier. Performs a serializing operation on all store-to-memory instructions that were issued prior to this instruction.

MFENCE: Memory Barrier. Performs a serializing operation on all memory instructions (load and store) that were issued prior to this instruction.

For example, let $persistentInt$ be a persistent integer variable. We want to persistently write 1 to this variable. To do so, we have to execute the following sequence of instructions:

```
persistentInt = 1;
MFENCE();
CLFLUSH(&persistentInt);
MFENCE();
```

The first memory barrier guarantees that value 1 has effectively been written to $persistentInt$ and that the following flushing instruction will write back the *new* value of $persistentInt$. The cache line flushing instruction invalidates the cache line where $persistentInt$ is held and writes it back to its location in SCM. This implies that persistent variables must not be split between two cache lines. The last memory barrier ensures that the cache line flushing instruction is not reordered with the next instructions. This is required to order persistent write operations. In the following sections, we use the suffix "Flush" to indicate a persistent write operation.

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", Art. Nr. 8, ISBN 978-1-4503-2971-2
https://doi.org/10.1145/2619228.2619236

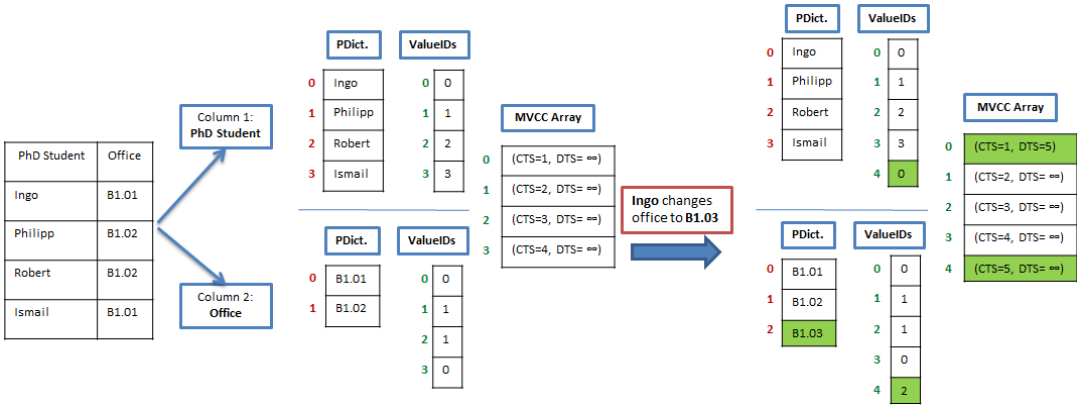


Figure 2: Example of a SOFORT PTable. The office of Ingo is updated from B1.01 to B1.03.

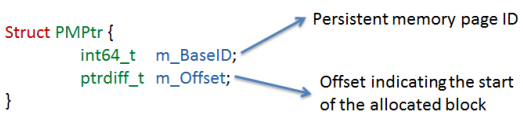


Figure 3: SOFORT Persistent Memory Pointer PMPtr.

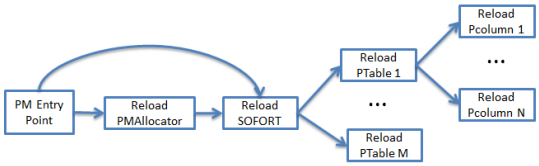


Figure 4: SOFORT Recovery Diagram.

Memory controllers on modern hardware have write buffers that can hold a write even after it has been flushed. To overcome this issue, a mechanism needs to be put in place so that the memory controller drains its buffer on power failures.

3.2 Persistent Memory Management

Persistent memory is managed using a file system. User space access to persistent memory 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 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 Memory Pointer (PMPtr). As illustrated in Figure 3, it consists of a base, which is a persistent memory page ID, and an offset that relates to the start of the allocated block. To manage persistent memory, we propose a persistent memory allocator (PMAllocator) that provides a mapping of persistent memory to virtual memory which enables the conversion of PMPtrs to regular pointers. We denote "work pointers" regular pointers that are conversions of PMPtrs. In this context, [27] have proposed "pointer swizzling at page fault time" where disk pointers are converted to virtual memory pointers at page fault time. While our pointer conversion principle is the same as in [27], we convert pointers only at restart time and not for every page fault.

Another solution would be to recover persistent memory in the same virtual address space, in which case regular pointers would remain valid. However, the operating system does not guarantee that the previous virtual address segments will remain free. For example, using the option MAP_FIXED of *mmap* allows to recover data in the same address space but will unmap any mapped files in the specified memory region, eventually leading to undesirable behaviors. Besides, using fixed addresses for data is a bad idea for security reasons.

The PMAllocator uses big persistent memory pages that are cut into smaller segments for allocation. It maps these pages to virtual memory to enable the conversion of PMPtrs to work pointers. At restart time, a new mapping to virtual memory is created, allowing to re-convert PMPtrs to new valid work pointers. The PMAllocator is also persistent. It maintains a persistent memory page counter to know how many pages it has created. It also stores meta-data at the beginning of every allocated segment. At restart time, the PMAllocator re-opens its persistent memory pages and reconstructs the mapping of its persistent memory to virtual memory.

To perform recovery, we need to keep track of a persistent memory 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. A persistent entry point can simply be two PMPtrs: the first points to SOFORT's persistent structures and the second to the PMAllocator persistent object. The entry point is kept in SCM on a small persistent memory page with a fixed ID.

3.3 Recovery Mechanism

Figure 4 shows how recovery at restart is performed from a single persistent entry point. Once the PMAllocator has recovered its objects, SOFORT recovers its tables in parallel. Every data structure has a *reload* function which checks the sanity of its state, restores work pointers from PMPtrs, and recovers from problematic situations, such as crashing in the middle of a table resize. In every transaction, updating MVCC information is the last operation to perform. Since our data structures are append-only, we do not need to do anything since MVCC entries attest the validity and visibility of every row.

The only time consuming operation left is rebuilding the dictionary indexes. We made the choice to keep them volatile for performance reasons, but we could make them persistent on SCM using a persistent map structure, such as the CDDS-Btree [23]. Therefore, recovery time depends on the size of the dictionary indexes. As demonstrated in Section 5, although we rebuild dictionary indexes, SOFORT total recovery time takes only a few seconds.

4. SOFORT ALGORITHMS

In this section, we detail some core operations of SOFORT in order to demonstrate how consistency and durability are achieved. The primary challenge is to make the operations failure atomic: regardless of crash condition, we can recover the database to a consistent state. We exclude read operations since they do not change the

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", Art. Nr. 8, ISBN 978-1-4503-2971-2
https://doi.org/10.1145/2619228.2619236

database state and thus, read operations are not a challenge for consistent recovery. Due to space constraints, we discuss only the Update operation. The Insert and the Delete operations are detailed in Appendix A.

4.1 Update

Algorithm 1 Update(TableName,Key,NewRow)

```
1:  // Get table pointer from table name
2:  ptab = find(TableName)
3:
4:  // Get RowID from key value
5:  DeleteIdx = ptab–>getLatestVisible(Key)
6:
7:  // Reserve a row and get back its index
8:  InsertIdx = ptab–>ReserveRow()
9:
10: // Update the transaction object with InsertIdx and DeleteIdx
11: UpdateTransFlush(DeleteIdx, InsertIdx)
12:
13: // Try lo lock the row
14: if !AtomicLock(MVCCArray[DeleteIdx]) then
15:     Abort
16: end if
17: // Persistently insert new row
18: ptab–>pushBackFlush(InsertIdx, NewRow)
19:
20: // Commit by updating the two MVCC entries
21: CommitUpdate(DeleteIdx, InsertIdx)
22: Flush MVCCArray[DeleteIdx]
23: Flush MVCCArray[InsertIdx]
```

The update operation is effectively a Delete operation followed by an Insert operation. Algorithm 1 describes the core steps of this operation. First, we get a pointer to the target table using a mapping from table names to table IDs (line 2). Then, we look up the indexes of the target table to get the latest visible row where the key value (Key) occurs (line 5). Afterwards, we reserve a new row in the table and get back the corresponding row index (line 8). If needed, a resize of the table is triggered. The index of the new row is computed with an atomic increment of a counter maintained by the table. No other operation will write neither to the row pointed by this index nor to the corresponding MVCC entry. Thus, the operation is latch-free and thread-safe. Afterwards, we persistently update the transaction object with the index of the row to be inserted and the index of the row to be deleted (line 11). Then, we execute an atomic compare-and-swap operation to try to lock the row to be deleted by setting its DTS to the transaction ID (line 14) and abort the transaction if the atomic operation fails (line 15). If it succeeds, we persistently append the row to to be inserted to the table (line 18). The transaction commits by fetching a commit time-stamp and assigning it to the CTS of the row to be inserted and the DTS of the row to be deleted (21). Finally, we persist the commit by flushing the updated MVCC entries (lines 22-23).

The only challenging failure scenario is a crash between lines 22 and 23 in which case the commit might or might not have been propagated to persistent memory. To address this scenario, during recovery, we rollback the update transactions that were active at the time of failure. To do so, we visit every TRX array entry and do the following for every update transaction: if the CTS of the row to insert and the DTS of the row to delete are valid time-stamps, then do nothing. Otherwise, reset both the CTS of the row to insert and the DTS of the row to delete to ∞. As shown in Appendix A, only update operations need to be rolled back.

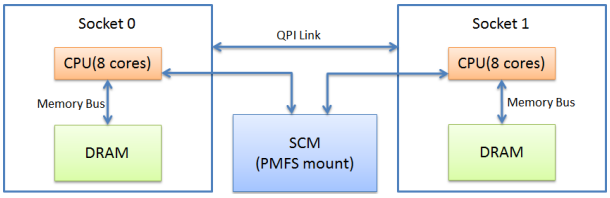


Figure 5: System Setup Overview.

5. EVALUATION

In this section, we give an overview of our system setup. Afterwards, we evaluate SOFORT in two steps. First, we evaluate SOFORT's OLTP performance using the TATP benchmark. Second, we evaluate SOFORT's restart time using a micro-benchmark.

5.1 System Setup

We assume a hybrid SCM-DRAM environment where both DRAM and SCM can be accessed using load-store semantics. We envision in the future SCM maybe fully integrated into the memory subsystem where SCM can be accessed using Load/Store semantics. In this paper, we attempted to emulate this scenario. We used a system equipped with a dual socket Intel®Xeon®E5 processor @ 2.60GHz, with 20MB of L3 cache and 8 physical cores per socket. Figure 5 gives an overview of the system setup. During all our tests, Intel HyperThreading Technology was disabled. In order to avoid NUMA effects in the performance measurements, we bind the process of the benchmark to run to a single socket. (NUMA effects could be of the same order of magnitude as the higher SCM latencies.) The DRAM latency was measured with Intel Memory Latency Checker[24] as 90ns. Since different SCM media expose different latencies, we used multiple SCM latencies during our tests.

SCM is managed by Persistent Memory File System (PMFS), a file system optimized for byte-addressable non-volatile memory [10]. Memory mapped PMFS files are not buffered in DRAM. This ensures that applications are given direct access to SCM. The PMAllocator uses 1GB PMFS files, each corresponding to a logical memory page.

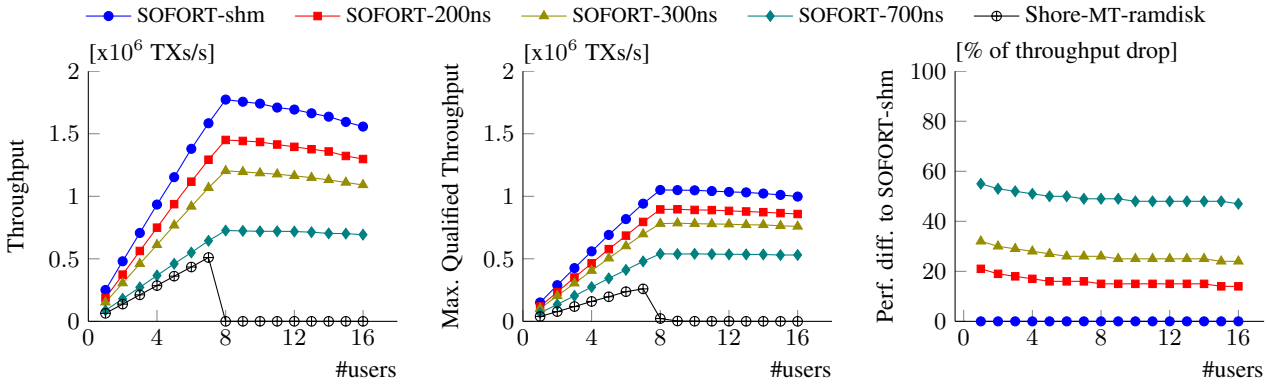
The hardware uses a custom BIOS to emulate the different higher latencies of SCM. One limitation of this emulation is that the tuned latency is read-write symmetric, while SCM is expected to have asymmetric read-write latencies, with writes slower than reads.

5.2 Throughput

To measure the throughput of SOFORT, we have implemented the TATP benchmark, a simple but realistic transactional benchmark that simulates a telecommunication application [2]. It is composed of 80% read transactions and 20% write transactions. TATP measures *Maximum Qualified Throughput*, which is the number of successful transactions per second. We run two sets of experiments to measure throughput. The first one is read only and consists of running GetSubData, one of the TATP queries. The second one is running the full TATP benchmark. In both cases, we vary the number of users from 1 to 16. We also vary the latency of SCM, from 200 ns to 700 ns. To provide an upper bound of SOFORT's performance, we also run SOFORT on shared memory, i.e. on DRAM with a latency of 90 ns. We run TATP with a scale factor of 100 which corresponds to an initial population of 1M subscribers.

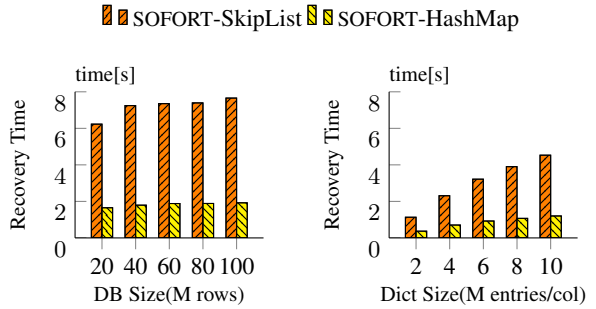
To provide a baseline for SOFORT's throughput performance, we also experiment with Shore-MT [13]. We use Shore-MT on ramdisk (a Temporary File System mount) (Shore-MT-ramdisk) to get an upper bound of Shore-MT's performance. We have also tuned the configuration of Shore-MT baseline to get the highest possible throughput.

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", Art. Nr. 8, ISBN 978-1-4503-2971-2
https://doi.org/10.1145/2619228.2619236



(a) TATP GetSubData (b) TATP Mix (c) Higher latencies impact on TATP Mix

Figure 6: TATP benchmark SF 100 throughput results. SOFORT outperforms Shore-MT and resists better to user contention. SOFORT stays competitive even in a high read-write latency (700 ns) SCM environment.



(a) Dict. size fixed(10M entries/col) (b) DB size fixed(10M rows)

Figure 7: SOFORT Restart Time. Recovery time is dominated by the rebuild time of dictionary indexes.

Figure 6 illustrates throughput results for the experiments described above. SOFORT on shared memory (SOFORT-shm) achieves up to 1.8M transaction per second for the read benchmark (Figure 6a), and up to 1.1M transaction per second for the full TATP benchmark (Figure 6b). We also notice that even in a high SCM latency environment (700 ns), SOFORT stays competitive and still outperforms Shore-MT on ramdisk. Figure 6c highlights the impact of higher SCM latencies on SOFORT's TATP throughput relative to using shared memory. We observe that for a latency of 200 ns, which is more than the double of shared memory latency (90 ns), the performance drop is only approximately 20%. This results from the hybrid design of SOFORT, where structures can be either volatile (hence, faster access) or non-volatile. Additionally, SOFORT resists user contention regardless of SCM latency.

The throughput of Shore-MT on ramdisk drops to almost zero with more than 8 users due to resource contention. However, with Dora, Shore-MT resists to a certain extent user contention [19].

5.3 Recovery

We define recovery time as the time it takes the database to recover and answer a first simple select query. To measure this time, we have implemented a micro-benchmark. The database consists of a single table of 4 integer columns. We change the database size by varying the number of rows, and the dictionaries size by varying the number of dictionary entries, i.e. the number of distinct values per column. In the following experiments, SCM latency is set to 200 ns.

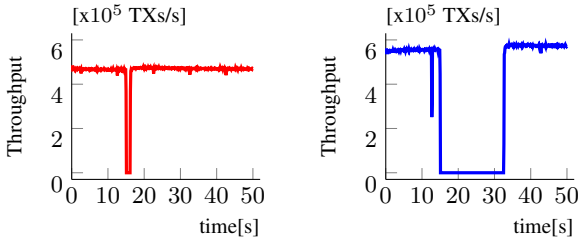
We compare two configurations of SOFORT. They differ in the data structure used for the dictionary index. One uses a lock-free skip list map (default configuration) whereas the other one uses the Intel Threading Building Blocks hash map [1]. In Figure 7b, we vary the database size while keeping dictionaries size fixed. In Figure 7a, we keep a fixed database size and vary dictionaries size. The first observation is that SOFORT-HashMap is faster to recover than SOFORT-SkipList. We also observe that recovery time increases linearly with the dictionaries size. This shows the price to pay at recovery time for keeping a volatile structure for dictionary indexes, although it enables better throughput. Besides, we notice that the restart time does not depend on the database size for a given dictionaries size.

Table 1: Breakdown of SOFORT's Restart Time. DB Size=100M rows. Dict. Size=10M entries/column.

Configuration	Dict. Indexes Rebuild.	Rest of Recov.
SOFORT-HashMap	1848 ms	74ms
SOFORT-SkipList	7599ms	58ms

Table 1 represents the breakdown of the restart time into two parts: the time spent in rebuilding the dictionary indexes and the time spent doing the rest of the recovery process. We observe that rebuilding dictionary indexes largely dominates total restart time.

To illustrate recovery time, we run TATP with 4 users on SOFORT and crash the database at second 15, as shown in Figure 8. To provide a baseline, we have implemented SOFORT on disk using memory mapping. This means that data is kept on DRAM and is backed by files on disk. Since SOFORT on disk is not crash safe, we call *sync* to flush all the dirty memory mapped pages to disk before crashing the system. Figure 8a shows SOFORT's throughput on PMFS before, during, and after recovery. SOFORT recovers in



(a) SOFORT-PMFS-200ns (b) SOFORT-Disk

Figure 8: SOFORT Recovery. TATP throughput with 4 users. The database is crashed at second 15.

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", Art. Nr. 8, ISBN 978-1-4503-2971-2
https://doi.org/10.1145/2619228.2619236

approximately 1 second and delivers right away full throughput performance since it does not need to warm up. Figure 8b shows the throughput of SOFORT on disk, where recovery takes approximately 18 seconds, even though SOFORT on disk does not have a real persistence. In conclusion, SOFORT on SCM offers a good trade-off as it achieves high OLTP throughput performance and recovers from system failures in seconds.

6. RELATED WORK

To our knowledge, we are the first to propose a SCM-based transactional column-store. Bailey et al. [4] presented *Echo*, a persistent key-value storage system with snapshot isolation. Contrary to SOFORT which assumes that DRAM and SCM are of the same memory hierarchy level, *Echo* adopts a two-level memory design with a thin layer of DRAM on top of SCM. Other works use SCM to optimize OLTP durability management [20, 11]. Their focus is on disk-based, row-store databases while our focus is on main-memory column-stores. Additionally, Chen et al. [5] discuss how B+-trees and hash joins can benefit from SCM. Venkataraman et al. [23] contributed Consistent and Durable Data Structures that leverage the non-volatility of SCM using versioning.

To manage SCM, Condit et al. [8] presented BPFS, a carefully designed, high performance transactional file system that runs on top of SCM. Nayaranan et al. [18] proposed Whole System Persistence (WSP), where data is flushed only on power failures using the residual energy of the system. However, they do not consider software failures. Recent works have also looked at how to architect SCM. Qureshi et al. [21] discussed how to architect SCM-based high performance main-memory systems and showed that a buffer of 1GB of DRAM can hide the higher latency of 32GB of SCM. Moreover, Lee et al. [17] discussed how to use Phase Change Memory as a scalable DRAM alternative. Zhao et al. [28] proposed Kiln, a persistent memory design that employs a non-volatile last level cache and SCM to offer persistent in-place updates without logging or copy-on-write. Finally, other works have proposed interfaces to ease the use of SCM as persistent memory for programmers [6, 25].

7. CONCLUSION AND OUTLOOK

In this paper, we investigated how to design a columnar transactional storage engine that leverages full capabilities of SCM. We proposed SOFORT, a log-less, single-level columnar data store with high OLTP throughput performance and fast data recovery. SOFORT relies on persistent, self-managed data structures and a novel programming model for persistent memory to achieve its goals. Besides, SOFORT exhibits competitive performance, even in high-latency SCM environments.

We believe that our approach can also benefit main-memory row-stores. For future work, we plan to extend the concurrency control mechanism of SOFORT to support arbitrarily long transactions, design persistent index structures and experiment with write-through caching policy.

Acknowledgement

Special thanks go to Thomas Kissinger, Anisoara Nica, Norman May and the SAP HANA PhD students for their helpful suggestions and discussions. We are also grateful to Porobic Danica for her help with Shore-MT.

8. REFERENCES

[1] Intel®Threading Building Blocks. https://www.threadingbuildingblocks.org/.

[2] Telecommunication Application Transaction Processing (TATP) Benchmark. http://tatpbenchmark.sourceforge.net/.

[3] Intel®Architecture Instruction Set Extensions Programming Reference. Technical report, 2014. http://software.intel.com/en-us/intel-isa-extensions.

[4] K. A. Bailey, P. Hornyack, L. Ceze, S. D. Gribble, and H. M. Levy. Exploring storage class memory with key value stores. In *INFLOW 2013*.

[5] S. Chen, P. B. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In *CIDR 2011*.

[6] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. *SIGPLAN Not.*, 47(4), Mar. 2011.

[7] R. Colledge. *SQL Server 2008 administration in action*. Manning, Greenwich, CT, 2010.

[8] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *SOSP 2009*.

[9] X. Dong, X. Wu, G. Sun, Y. Xie, H. Li, and Y. Chen. Circuit and microarchitecture evaluation of 3d stacking magnetic ram (mram) as a universal memory replacement. In *DAC 2008*.

[10] S. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, R. Sankaran, J. Jackson, and D. Subbareddy. System software for persistent memory. In *EuroSys 2014*.

[11] R. Fang, H.-I. Hsiao, B. He, C. Mohan, and Y. Wang. High performance database logging using storage class memory. In *ICDE 2011*.

[12] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, H. Nagao, and H. Kano. A novel nonvolatile memory with spin torque transfer magnetization switching: spin-ram. In *IEDM 2005*.

[13] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-mt: A scalable storage manager for the multicore era. In *EDBT 2009*.

[14] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst., 6(3), 1981.*

[15] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4), Dec. 2011.

[16] B. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger. Phase-change technology and the future of main memory. *Micro, IEEE*, 30(1), Jan 2010.

[17] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. *SIGARCH Comput. Archit. News*, 37(3), June 2009.

[18] D. Narayanan and O. Hodson. Whole-system persistence. In *ASPLOS XVII*, 2012.

[19] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *Proc. VLDB Endow.*, 3(1-2), Sept. 2010.

[20] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge. Storage management in the nvram era. *PVLDB*, 7(2), 2013.

[21] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *ISCA 2009*.

[22] B. M. Sang-Won Lee. Accelerating in-page logging with non-volatile memory. *IEEE Data Eng. Bull.*, 33, 2010.

Final edited form was published in "SIGMOD/PODS'14: International Conference on Management of Data. Snowbird 2014", Art. Nr. 8, ISBN 978-1-4503-2971-2
https://doi.org/10.1145/2619228.2619236

[23] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST 2011*.

[24] V. Viswanathan, K. Kumar, and T. Willhalm. Intel memory latency checker. Technical report. http://www.intel.com/software/mlc.

[25] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. *SIGPLAN Not., 47(4), 2011.*

[26] R. Williams. How we found the missing memristor. *Spectrum, IEEE*, Dec 2008.

[27] P. R. Wilson. Pointer swizzling at page fault time: Efficiently supporting huge address spaces on standard hardware. *SIGARCH Comput. Archit. News*, 19(4):6–13, July 1991.

[28] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *MICRO-46*, 2013.

APPENDIX

A. SOFORT ALGORITHMS

This appendix is the continuation of Section 4 in which we describe some of SOFORT's core algorithms.

A.1 Insert

Algorithm 2 illustrates the core steps of an insert operation. First, we get a pointer to the table using a mapping from table names to table IDs (line 2). Then, we reserve a new row and get back the corresponding row index (line 5). No other writers will write neither to the row pointed by this index nor to the corresponding MVCC entry. Thus, the operation is latch-free and thread-safe. If needed, a resize of the table is triggered. Afterwards, the row is persistently appended to the table (line 8). The last step is to atomically and persistently commit the transaction by updating the CTS of the new MVCC entry (lines 11-12).

If a crash happens at any step before the commit has reached SCM, the transaction is considered as aborted and the row will be recovered in an invisible state (see Section 2), i.e., without any impact on the state of the database. Therefore, there is nothing to do at restart time.

Algorithm 2 InsertRow(TableName,NewRow)

```
 1: // Get table pointer from table name
 2: ptab = find(TableName)
 3:
 4: // Reserve a row and get back its index
 5: InsertIdx = ptab–>ReserveRow()
 6:
 7: // Persistently insert row and get back
 8: ptab–>pushBackFlush(InsertIdx,NewRow)
 9:
10: // Commit by updating the CTS
11: CommitInsert(InsertIdx)
12: Flush MVCCArray[InsertIdx]
```

A.2 Delete

The delete operation is described in Algorithm 3. First, we get a pointer to the target table (line 2). By a column index lookup, we get the latest visible row where the key value (Key) occurs (line 5). The DTS of the corresponding MVCC entry indicates whether this row is being modified by another write transaction. We execute an atomic compare-and-swap operation to try to lock the row and commit the delete at the same time by setting its DTS to the transaction's commit time-stamp (line 8). If the atomic operation fails, we abort the transaction (line 9). Otherwise, the row has been successfully locked and the delete operation committed. The commit is finalized by persisting the updated MVCC entry (line 12).

If a crash occurs before locking and committing a row, the transaction is considered as aborted and there is nothing to do at recovery time.

Algorithm 3 Delete(TableName,Key)

```
 1: // Get table pointer from table name
 2: ptab = find(TableName)
 3:
 4: // Get RowID from key value
 5: DeleteIdx = ptab–>getLatestVisible(Key)
 6:
 7: // Try lo lock and commit the row
 8: if !AtomicTryLockCommit(MVCCArray[DeleteIdx]) then
 9:     Abort
10: end if
11: // Flush updated MVCC
12: Flush MVCCArray[DeleteIdx]
```