Instant Recovery for Main-Memory Databases

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

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

Thomas Kissinger
Technische Universität Dresden
thomas.kissinger@tu-dresden.de

Thomas Willhalm
Intel GmbH
thomas.willhalm@intel.com

Peter Bumbulis
SAP SE
peter.bumbulis@sap.com

ABSTRACT

With the emergence of new hardware technologies, new opportunities arise and existing database architectures have to be rethought to fully exploit them. In particular, recovery mechanisms of current main-memory database systems are tuned to efficiently work on block-oriented, high-latency storage devices. These devices create a bottleneck during transaction processing. In this paper, we investigate the opportunities given by the upcoming Storage Class Memory (SCM) technology for database system recovery mechanisms. In contrast to traditional block-oriented devices, SCM is byte-addressable and offers a latency close to that of DRAM. We propose a novel main-memory database architecture that directly operates in SCM, eliminates the need for logging mechanisms, and provides a way to trade recovery time with the overall query performance. We implemented these concepts in our prototype SOFORT. Our evaluation shows that we are able to achieve instant recovery of the DBMS while removing the need for transaction rollbacks after failure.

1. INTRODUCTION

The shift from traditional disk-centric database system architectures to main-memory-centric architectures has brought a major technological disruption with different prevalent physical designs (e.g., column stores and main-memory-optimized index structures) and corresponding query processing models. More and more emerging hardware technologies and architectures (e.g., Hardware Transactional Memory (HTM), Remote Direct Access Memory (RDMA), and heterogeneous cores) force us to rethink existing database architectures [19] to let the DBMS adapt and exploit the opportunities coming from the hardware side.

In this paper, we focus on the impact of Storage Class Memory (SCM) on the recovery mechanisms of a database system. Due to the block-oriented nature of today's non-volatile devices (e.g., hard disk or flash), both traditional disk-based and modern main-memory database systems rely on additional log and snapshot components to efficiently guarantee consistency and durability. SCM, however, is byte-addressable and offers a latency close to that of DRAM. Thus, we propose a novel main-memory database architecture, as depicted in Figure 1, that directly operates on SCM, eliminates the need for an explicit I/O and logging component and provides a vehicle to trade recovery performance with the overall query performance.

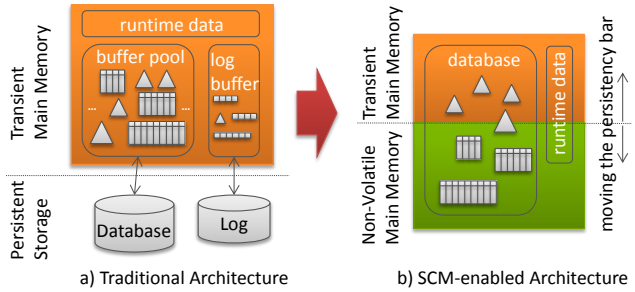


Figure 1: Database architecture in the presence of SCM.

Using a hybrid memory model consisting of both traditional RAM and persistent SCM allows for a radical change and a novel system architecture in different perspectives: First, there is no need to copy data out of the storage system but directly modifying the data stored in SCM becomes possible. Second, the use of concurrent and persistent data structures in combination with an adequate concurrency scheme (e.g., Multi-Version Concurrency Control) allows for completely dropping the logging infrastructure. Third, it can be dynamically decided where to store certain database objects as long as their loss can be tolerated, e.g., index structures may be stored in transient RAM and reconstructed within the recovery procedure, potentially based on recent workload patterns. Moreover, system runtime information can easily and efficiently be stored on the SCM side to improve system startup time. Finally, the system may retain and continue running transactions in case of system failures, due to instant and point-in-time recovery. For large main-memory database systems running in business-critical environments (e.g., online shops), this is a major advancement compared to traditional log-based systems.

In this paper and the accompanying demonstration proposal, we outline the basic concepts of our SOFORT[1] system, in which we implement and evaluate our design principles. As we show, our system allows placing database objects either in DRAM or in SCM. This yields the opportunity to balance additional runtime overhead due to the higher latency of SCM with the time to fully recover from

[1] "SOFORT" in German means: "instantly".

This article is published under a Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/), which permits distribution and reproduction in any medium as well allowing derivative works, provided that you attribute the original work to the author(s) and CIDR 2015.

7th Biennial Conference on Innovative Data Systems Research (CIDR '15)
January 4–7, 2015, Asilomar, California, USA.
.

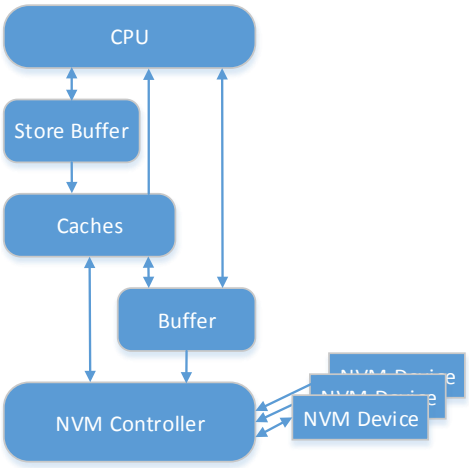


Figure 2: Simplified architectural model.

a system crash. At one extreme, the system stores all data in SCM, allowing to recover instantly and continue running transactions immediately after recovery. At the other extreme where only primary data is stored in SCM, the recovery procedure asynchronously recomputes, step-by-step, secondary data structures out of the primary SCM data and stores it (for faster read/write access) in transient RAM. Obviously any configuration in between these extremes allows tuning the system for the specific needs of an application scenario.

We demonstrate the overall system concept and report experimental results of a working prototype running on an SCM simulator directly working on DRAM and SCM representations of the corresponding data set. SOFORT is organized as a column store with a traditional separation of the primary data set into a read-optimized static data store and a write-optimized dynamic data store that buffers incoming changes to the data set and is periodically merged into a new version of the static data store. To ensure physical data integrity, persistent data structures used in SCM are carefully designed to recover, in case of failures, in a consistent state relative to the rest of the database.

In the remainder of this paper, Section 2 presents specific background with respect to Storage Class Memory and the used hardware simulator, while Section 3 outlines our prototype SOFORT that was developed according to our outlined architectural principles. Thereafter, we detail the recovery mechanism in Section 4. We conduct the evaluation in Section 5 and review related work in Section 6. Finally, Section 7 concludes this paper.

2. BACKGROUND

In this section we first give a brief description of SCM and describe the SCM hardware simulator that we use to evaluate our system. Thereafter, we introduce a series of microbenchmarks that helps characterize the performance of SCM and from which we infer our system design decisions.

2.1 Storage Class Memory

Storage Class Memory (SCM) is Non-Volatile Memory (NVM) that exhibits latency characteristics close to that of DRAM with density, durability, and economic characteristics comparable to existing storage media. Current SCM technologies provide an asynchronous latency within an order of magnitude of DRAM, with writes noticeably slower than reads. Promising SCM technologies include Phase Change Memory [15], Spin Transfer Torque RAM [12], Magnetic RAM [9], and Memristors [25]. Continuing advances in NVM technology hold the promise that SCM will become reality in the closer future.

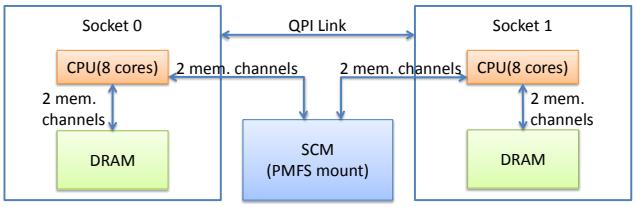


Figure 3: Hardware simulator overview.

Persistence Primitives

To maintain the invariants required to provide ACID guarantees, database systems must reason about when and in what order writes are persisted (made durable). However, memory subsystems may reorder writes to improve performance and existing processors provide no explicit mechanism to prevent this. To guarantee that a write to location A is persisted before a write to location B, one must explicitly persist the write to location A before writing to location B: there is no MFENCE counterpart for persistence[2]. As in the work by Bhandari et al. [6] we assume an architectural model as shown in Figure 2. In this model NVM is connected via an NVM controller. A store buffer is used to improve performance by hiding memory write latency and a buffer is used to allow non-temporal writes to completely bypass the cache. Ensuring that a write is persisted requires first ensuring that the write reaches the controller and then ensuring that the contents of the controller buffers are presented to NVM. Bhandari et al. [6] outline three ways in which writes can be made visible to the controller: stores followed by explicit flushes of the affected cache lines (using CLFLUSH), non-temporal writes, and setting the caching policy of persisted memory to write through. Which approach performs better depends on the algorithm being implemented. Currently the only way of ensuring that the controller buffers are presented to NVM is via a mechanism like ADR [3] that forces the contents of controller buffers on power failure. Future processors will provide additional mechanisms: for example, Intel processors will provide a PCOMMIT [4] instruction to explicitly force the contents of the controller buffers to NVM.

Hardware Simulator

In this paper we assume a hybrid architecture that is equipped with both DRAM and SCM, where the software can allocate memory in SCM through the memory interface. For our implementation we used Persistent Memory File System (PMFS) [10], a SCM-aware file system, as the SCM interface, but the results are independent of the specific API as long as it provides routines to perform the following:

(1) Allocate an SCM segment of a given size, returning an identifier.
(2) Map existing persistent SCM memory into the virtual address space using an identifier.
(3) Free persistent SCM memory using an identifier.

We do assume that the virtual address can change when persistent memory is mapped into the virtual address space of the database process.

For our evaluation, we use the same kind of NVM Evaluation Platform (NVMEP) as in Dulloor et al. [10] and Oukid et al. [17]. Using a special BIOS, the NVMEP simulates a different latency by

[2]MFENCE only ensures that all CPUs have a consistent global view of memory.

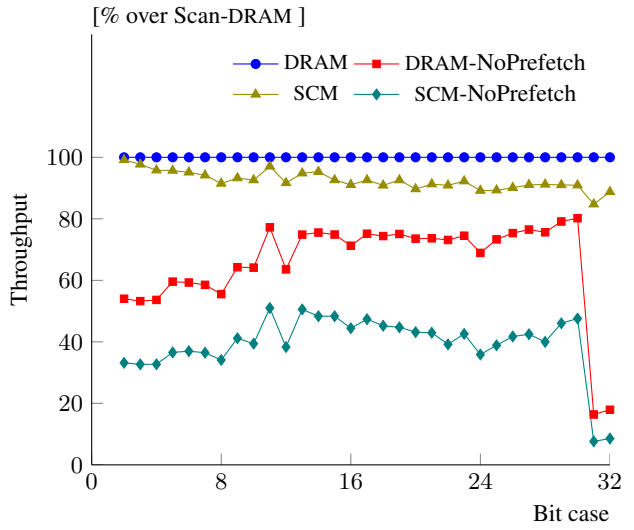


Figure 4: SIMD Scan performance on DRAM and SCM with/without prefetching.

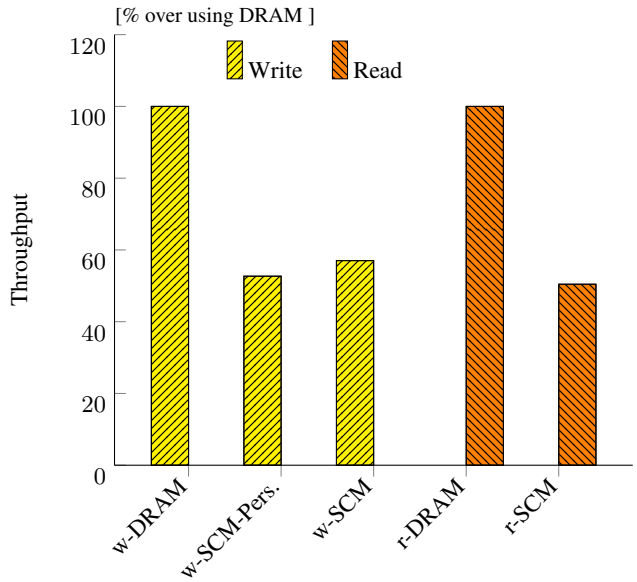


Figure 5: Skip List read/write performance on DRAM and SCM.

means of a microcode patch. The platform is equipped with two Intel Xeon E5 processors running at 2.60 GHz. Each processor has 8 physical cores with 64KB L1 and 256KB L2 data cache each, and a shared 20MB last-level cache. During all of our tests, Hyperthreading Technology was disabled.

Figure 3 gives an overview of the NVMEP. Conceptually, local DRAM is attached to each socket, whereas SCM is attached to both sockets. The NVMEP emulates this architecture by assigning two of the four memory channels on each socket as DRAM. The memory on the remaining two memory channels emulates a uniform SCM region by interleaving memory on a cache-line level. The microcode patch adds latency to all memory accesses in this memory region, which therefore emulates SCM. and allows varying its latency. We chose to set the latency to 200 ns in our experiments, which is more than two times the latency of DRAM (90 ns). From an application's perspective, however, PMFS manages all emulated SCM, which is therefore not visible as DRAM. To separate the impact of Non-Uniform Memory Access (NUMA) from the impact of SCM, we use only the cores and DRAM of the first socket for all of our tests.

2.2 Microbenchmarks

To better understand the implications of SCM on database performance, we have conducted a series of microbenchmarks. In the following we describe some of these microbenchmarks and present our main findings. All microbenchmarks are single-threaded.

Full Column Scan

In the first microbenchmark, we evaluate the performance of a full column Single Instruction Multiple Data range scan operator (SIMD Scan) on DRAM and on SCM [24]. The SIMD Scan operates on bit-packed compressed columns, where the bit case indicates the number of bits used to represent a value. Figure 4 shows that for an SCM latency of 200 ns, which is more than two times higher than the 90 ns DRAM latency, the average performance penalty for the different bit cases is 8% compared with using DRAM. In addition, the average performance penalty for the lower bit cases (1–16) is only 5%. This is due to hardware prefetching that hides the higher latency of SCM when detecting a sequential access pattern. With hardware prefetching disabled for both SCM and DRAM [1], we observe that the average performance penalty of using SCM rises to 41% compared with using DRAM. If we compare similar memory technologies, disabling prefetching incurs a slowdown of 34% for DRAM and 56% for SCM.

We conclude that the higher latency of SCM has little impact on the performance of the SIMD Scan.

Skip List on SCM

However, not all data structures perform the same in SCM and in DRAM. In this microbenchmark, we evaluate the read and write performance of a skip list in DRAM and in SCM. We experiment with a transient and a persistent skip list; the transient skip list is simply a traditional skip list without modifications to its implementation while the persistent skip list makes its changes persistent immediately using persistence primitives, such as, flushing instructions and memory barriers. The skip list is first populated with one million of 2×4 bytes integer key-value pairs before the microbenchmark is run, then, we execute 10 million either read or insert operations. Figure 5 illustrates the results of this experiment. The prefixes *w* and *r* on the labels of the figure's *y* axis are abbreviations for *write* and *read*, respectively. We observe that there is a 49% penalty for reads when using SCM instead of DRAM, while for writes the penalty is 43% and 47% for a transient skip list and a persistent skip list, respectively. This is due to the fact that the memory access pattern is random and therefore unpredictable for hardware prefetchers.

Effect of Higher SCM Latencies

Finally, we conduct a study of SCM latency effects on the two previous microbenchmarks. We restrict the experiment for the SIMD Scan to bit case 16. Figure 6 summarizes the performance results. We observe that the higher the latency of SCM gets, the more the skip list performance deteriorates. The performance penalty at an SCM latency of 700 ns is 82% and 76% for reads and writes compared with using DRAM, respectively.

As for the SIMD Scan, its performance decreases by only 8% for an SCM latency of 200 ns compared with using DRAM. However, its performance significantly deteriorates in the presence of high SCM latencies, because the hardware prefetcher is not able to

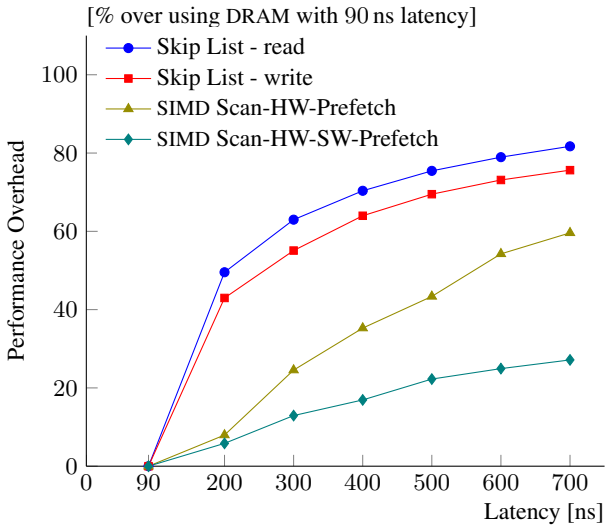


Figure 6: Impact of higher SCM latencies on the SIMD Scan and Skip List performance over using DRAM.

prefetch data at the right time. Indeed, one of the problems that the hardware prefetcher has to solve is *when* to prefetch data; if the data is prefetched too early, then it might be evicted from the cache before it is used by the CPU, and if the data is prefetched too late, then prefetching will not hide the whole memory latency. As the hardware prefetcher is calibrated for the latency of DRAM, it prefetches data too late when the latency of SCM is too high making it unable to hide the whole memory latency. This is a limitation of our hardware simulator and can be fixed in real hardware by simply having the hardware prefetcher calibrated for SCM.

To support our analysis, we have augmented the SIMD Scan with customized software prefetching for each SCM latency we experiment with. In Figure 6 we can clearly see that the overhead caused by the higher latencies of SCM is significantly decreased for the SIMD Scan; Using software prefetching decreases the overhead, at an SCM latency of 700 ns, from 60% to 27% compared with using DRAM.

Summary

In general, from the previous microbenchmark results, we draw the conclusion that latency sensitive data structures like indexes do not perform well in SCM, while operators with sequential data access patterns perform nearly equally in DRAM and SCM. Besides, software prefetching can be leveraged to hide the effects of high SCM latencies if the data access pattern of the workload is predictable. While static data structures can be easily made persistent on SCM, dynamic data structures require considerable changes to ensure their durability and post-recovery integrity.

3. SOFORT ARCHITECTURE

SOFORT is a single-level [3] transactional storage engine designed to leverage the full capabilities of SCM [17]. It is a hybrid system that takes advantage of both DRAM and SCM to achieve fast recovery while exhibiting good OLTP and OLAP performance. SOFORT stores data column-wise and employs dictionary encoding to speed up query execution, as processing integers is faster than processing other types of data, and to enable compression, such as using directory tokens for different cardinalities with different bit cases [23], for an optimal use of SCM and DRAM.

[3] single-level means that there is no separation between the working copy and the persistent copy of the data.

3.1 Data Placement

In general there are two ways of using SCM: (1) as DRAM replacement due to its better scalability and economic properties, or (2) as DRAM extension, and additionally leveraging its non-volatility property. We chose the latter way for SOFORT to take advantage of both technologies, namely SCM and DRAM. Leveraging the non-volatility property of SCM requires to redesign the database data structures using persistence primitives [4]:

(1) Store fences are used to enforce the ordering of stores.
(2) Non-temporal stores (Streaming SIMD Extensions (SSE) non-temporal instructions) allow to bypass the CPU cache and write directly to memory.
(3) Cache lines are flushed with CLFLUSH or, in the future, with the new optimized CLFLUSHOPT.
(4) A new instruction, named PCOMMIT, will flush the buffers in the memory controller. .

These primitives enable data structures to make their changes durable immediately and to recover after failure to a state of integrity. An example of a persistent data structure is the versioned B-tree proposed by Venkataraman et al. [20]. Our prototype SOFORT implements its own persistent data structures, such as arrays and columns, to guarantee physical integrity after system failure.

The design of SOFORT relies on a Twin-store main-memory database architecture, with a larger static store and a smaller dynamic store. The two stores are periodically merged to do garbage collection and keep the dynamic part small. We envision that the static part will be entirely kept in SCM and is instantly recovered after failure at a negligible cost. Traditional (non-persistent) data structures, such as indexes, can also be used for the static part in SCM since they are only changed during the merge process, which creates a new version of each such data object. Flushing all cache levels at the end of the merge process is sufficient to ensure that the static part is persistent in SCM.

The microbenchmark results presented in Section 2.2 support this vision; column scans and other operators with predictable memory access patterns perform nearly equally on SCM and on DRAM. As for index lookups, although latency sensitive, they still fulfill their main purpose on the static store, which is speeding up query processing. Indeed, even with half the performance, an index lookup remains orders of magnitude faster than a scan of a large column. Hence, we expect the static part on SCM to exhibit similar performance as on DRAM. However, the performance of OLTP queries handled by the dynamic part highly depends on the performance of its dynamic data structures. These dynamic data structures require considerable changes to ensure their durability and post-recovery integrity, which adds an extra overhead to the overhead caused by the higher latency of SCM.

Given the low latency of DRAM and the non-volatility property but high latency of SCM, SOFORT is designed to take advantage of both technologies. To do so, SOFORT seeks a trade-off between query performance and recovery performance by keeping primary data structures (i.e., columns and dictionaries) in SCM for faster recovery, and secondary data structures (e.g., indexes) in DRAM for better transaction performance.

3.2 Concurrency Control

Multi-Version Concurrency Control (MVCC) [14, 13] is a natural design decision for SOFORT, as versioning simplifies rollbacks, that is, makes them completely unnecessary, if the physical integrity of

persistent data structures can be ensured as it is done in SOFORT. Besides, it simplifies the design of persistent columns, as handling integers incurs less complexity in recovery procedures than handling complex data structures that have their own constructors and allocate memory, which makes memory leaks much more difficult to handle in case of failures. A detailed description of the concurrency control mechanism of SOFORT is available in [17].

Durability management relies on the fact that SOFORT is a single-level store, i.e., it does not differentiate between transient main memory and persistent storage for primary data structures. In other words, the working copy of the data *is the same* as the durable copy of the data. Consequently, SOFORT does not require a traditional transaction log to achieve durability, as changes are applied directly to the primary, durable data.

4. RECOVERY MECHANISM

In this section, we detail the recovery mechanism of SOFORT. First, we describe the basic recovery procedure. Afterwards, we explain how SOFORT retains and continues unfinished transactions at recovery. In the next subsection, we show how our basic recovery procedure can be extended to achieve instant recovery. Finally, we argue about how to trade between throughput performance and recovery time.

4.1 Basic Recovery Procedure

SOFORT distinguishes between two types of data structures: persistent ones and transient ones that are recovered in a different way. The persistent data structures are recovered by "reload" routines that check their physical integrity and allow them to recover from problematic situations. For instance, if the system fails while an array is being resized, the "reload" routine of the array will detect that and decide whether to continue to resize or to roll back to the previous size, depending on how far in the resize process the failure occurred. More technical details on our persistent memory management, such as persistent memory allocation and persistent pointers, can be found in [17]. The transient data structures on the other hand are fully rebuilt from the persistent data structures within the recovery procedure. This implies that primary data structures whose loss at failure causes a loss of information (e.g., the column store and the dictionary array) cannot be transient. Our basic recovery procedure follows these steps:

(1) Recover memory management information and rebuild the mapping from persistent memory to virtual memory. No data has to be fetched from slow storage media such as hard disks or SSDs during the whole recovery process.
(2) Recover persistent data structures, i.e., instant reload followed by integrity checking and self-contained repairs if needed.
(3) Examine the transaction control block array and continue unfinished transactions.
(4) Rebuild transient data structures on DRAM.

As explained in Section 3, we envision that the static part of the store will be kept in SCM and is recovered at a negligible cost. Regarding the dynamic part of the store, the "recovery" of persistent data structures executes instantly, independently of the data size, while rebuilding transient data structures is the only time consuming step in the procedure, which reflects a potential source of recovery overhead. However, in Sections 4.3 and 4.4, we show how to achieve instant and guaranteed recovery, even in presence of transient data structures, and we quantify the impact of this technique on usual query processing throughput.

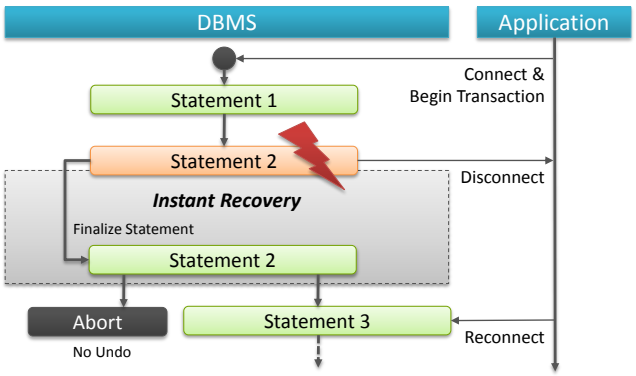


Figure 7: Transaction flow during a system failure.

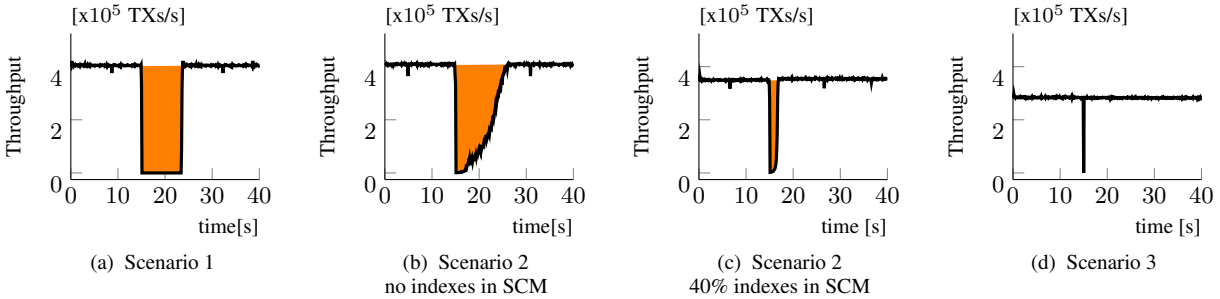
4.2 Continuing Unfinished Transactions

SOFORT does not need to roll back unfinished transactions after a database failure. Instead, it allows the user to continue executing open transactions. To achieve this, SOFORT executes a transaction statement by statement as usual. However, the difference to traditional systems is that SOFORT makes the changes of each statement durable, in SCM, before executing the next one. The changes, which are durable, are not visible to other transactions and will only become atomically visible when the transaction commits by updating the MVCC commit time-stamps. Figure 7 shows what happens in case of a system failure during the processing of a statement of a transaction. Every executed statement is guaranteed to have its incurred changes persisted. If the system crashes before a statement is finished, the latter is re-executed or finalized respectively right after recovery. To do so, the transaction list, an array of currently running transaction objects, is stored in SCM. It keeps redo information of currently executing statements, and classical MVCC information of the other already executed statements of the running transaction. Moreover, storing the transaction list in SCM does not affect performance as accessing it represents a negligible part of the total transaction execution time in addition to the fact that it is present most of the time in the CPU cache due to its small size.

With no transaction rollbacks after failure and instant recovery, the application or user will not notice that the system has crashed and will continue executing his transaction as if nothing happened. If the application does not reconnect to the DBMS after a crash, its transactions are aborted after a timeout. Since MVCC hides uncommitted changes from other transactions, there is no need to spend time on undoing the changes – they are removed during the next merge process. This approach is highly beneficial especially for long-running interactive transactions as they occur in e-commerce applications, which we will showcase in the attached demo proposal.

4.3 Instant Recovery

To achieve instant recovery, SOFORT starts accepting new transactions right after "reloading" the persistent data structures of the dynamic part of the store (step 2 in the procedure). This is possible because all of the primary data are persistent in SCM allowing us to process transactions without the vanished transient data structures. However, since the latter are responsible for speeding up query processing, the transaction throughput could be low at this point in time. SOFORT rebuilds transient data structures progressively and in parallel to normal transaction processing. The partially reconstructed transient data structures are leveraged to speed up query execution. Hence, transaction throughput during recovery continuously increases until it reaches, once all data structures have been



(a) Scenario 1

(b) Scenario 2 no indexes in SCM

(c) Scenario 2 40% indexes in SCM

(d) Scenario 3

Figure 8: Different SOFORT recovery schemes. TATP scale factor 500, 4 users. The database is crashed at second 15.

completely rebuilt, its maximum observed before failure. In the following section we discuss how to trade between runtime transaction throughput and guaranteed recovery performance.

4.4 Balancing Query Performance with Recovery Performance

As we have shown in Section 2.2, latency-sensitive data structures, such as indexes, perform almost two times better if they are located in DRAM, because of its low latency. However, storing secondary index structures in DRAM causes a significant drop in transaction throughput immediately after recovery, since they have to be rebuilt before the DBMS can leverage such indexes for optimized query processing. SOFORT tries to shorten the period of low throughput by employing a progressive rebuild mechanism for the transient secondary data structures. For example, the system keeps track of how far the dictionary has been indexed and, based on this information, it either scans the remaining part of the dictionary array or does a lookup on the partially rebuilt dictionary index. Nevertheless, operational database systems often require certain guarantees in terms of minimum transaction throughput per second. To enable SOFORT to meet such constraints, we allow placing a certain amount of transient data structures (or all in the extreme scenario) in SCM to store them in a persistent way. Besides, the rebuilding of the transient data structures can be done in a specific order, based on the transactions which need to be continued at the same time with the recovery process. By varying the amount of transient and persistent data structures, we are able to balance between a high transaction throughput during usual query processing and a guaranteed minimum throughput during recovery.

5. EVALUATION

We evaluate recovery performance with respect to three metrics: (1) *Recovery area* is the maximum number of transactions that could have been executed if the database had not failed. This corresponds to the colored area in Figure 8. Hence, the smaller the area of the colored region the better; (2) *Recovery response time* is the average query response time during the recovery process, and (3) *Recovery delta* is the time it takes to achieve the pre-failure throughput. Currently only the dynamic part is implemented in SOFORT.

We use the Telecom Application Transaction Processing (TATP) benchmark, a simple but realistic OLTP benchmark that simulates a telecommunication application [2]. We run the benchmark with 4 users and a scale factor of 500, which corresponds to an initial population of 5 million subscribers and a database size of roughly 10GB. Then, we crash the database and monitor the throughput variation during the recovery process by sampling it every 100 ms. Figure 8 summarizes the experiment results. We explore three scenarios: Scenario 1: all indexes are transient and SOFORT waits until the completion of their reconstruction before accepting queries; this corresponds to what we have proposed in [17]. Scenario 2: SOFORT accepts queries right after recovery where it uses the primary, persistent data structures to answer queries while taking advantage of the indexes that are being rebuilt in the background, as explained in Section 4.4. This scenario includes two sub-scenarios: in Figure 8b, all indexes are transient while in Figure 8c 40% of the indexes are persistent in SCM. Scenario 3: all indexes are persistent in SCM and there are no transient data structures to reconstruct.

Figure 8a shows the results for scenario 1. We observe that the throughput drops to zero at crash time and it takes SOFORT approximately 8 s to start answering queries again. The throughput is back to its maximum observed before failure immediately after SOFORT starts processing queries again. The recovery delta is then 8 s. The recovery time is spent rebuilding the transient data structures, which means that recovery time is proportional to the transient data structures size. Hence, recovery may take much longer for instances with larger transient data structures.

Figure 8b shows the results for scenario 2 where all indexes are transient. We observe that it takes only a few milliseconds after failure for SOFORT to start answering the first queries. The throughput gradually increases as secondary data structures are rebuilt to reach its maximum observed before failure after 8 s, which is identical to the recovery delta observed in scenario 1. Also, the recovery area has decreased by 16% compared with scenario 1. However, the recovery delta is still a function of the size of the transient data structures. This approach is optimal when throughput is to be favored. Figure 8c shows the case where 40% of the indexes are persistent in SCM. We notice that both the recovery area and the recovery delta are significantly improved; the recovery area is decreased by 82% compared with scenario 1 and the recovery delta is less than 2 s. However, this does not come for free as throughput during usual query processing is decreased by 14% compared with the two previous cases. This approach is optimal when we seek a trade-off between throughput and recovery performance.

Figure 8d shows the results for scenario 3. We notice that the throughput during normal processing is 30% lower than for scenario 1 for which both SCM and DRAM are leveraged. However, recovery is instant at maximum throughput observed before the crash. Indeed, the recovery area has completely vanished – it has decreased by 99.8% compared with scenario 1 and the recovery delta is only a few milliseconds. Besides, recovery is instant independently of data size, including secondary data structures, since recovering persistent objects takes no time. This scenario is optimal when the system must provide guaranteed throughput right after failure.

The distribution of the data structures over SCM and DRAM used in these scenarios are just one of many possible distributions. The more secondary data structures we put on DRAM, the higher the performance during usual query processing and the lower the perfor-

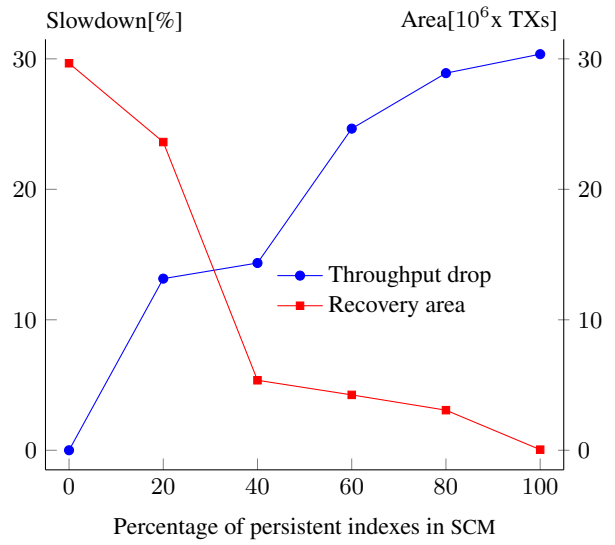


Figure 9: Impact of persistent indexes in SCM on throughput during usual query processing and recovery area. TATP scale factor 500, 4 users.

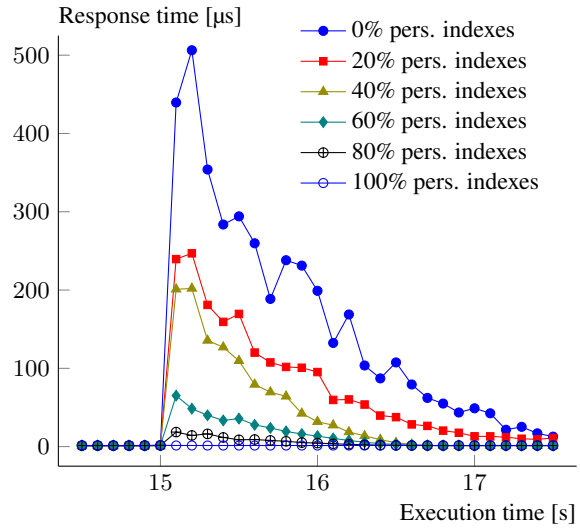


Figure 10: SOFORT average recovery response time. TATP scale factor 500, 4 users.

mance of recovery. In contrast, the more secondary data structures in SCM, the lower the performance during usual query processing and the better the recovery performance. This is due to the fact that current secondary data structures, such as B-trees and skip lists, are not a good fit for SCM as they are latency sensitive. On the other hand, data structures that exhibit more data locality, such as arrays and columns, are less latency sensitive and thus show good performance in SCM.

In the second experiment, we run TATP with a scale factor of 500, vary the percentage of secondary data structures in SCM and evaluate both throughput and recovery area. Figure 9 summarizes the results of the experiment. We notice that the more secondary data structures we put in SCM, the smaller the recovery area, and the lower the throughput during usual query processing. The throughput drop is limited to 30% for the extreme case where all secondary data structures are persistent in SCM. We also observe that both throughput and recovery area curves are not linear; it is due to the fact that not all secondary data structures are equally important for the TATP workload, which also holds for other workloads. Therefore, taking advantage of specific characteristics of a workload may lead to an optimal trade-off between throughput and recovery by for example, putting the most used secondary data structures in DRAM and the less used ones in SCM.

We have also evaluated the average response time during the recovery process, as shown in Figure 10. In this experiment, we also vary the percentage of secondary data structures in SCM and run TATP with a scale factor of 500. We observe that the more secondary data structures we put in SCM, the lower the recovery response time. Besides, the difference in response time between the two extreme cases, all secondary data structures in DRAM and all in SCM, is huge; for the former case, response time reached a peak of 506 µs while for the latter case it never exceeds 2 µs. In conclusion, depending on the scenario and context, we have a trade-off between throughput, response time guarantees, and recovery performance.

6. RELATED WORK

To our knowledge, with SOFORT, we are the first to propose a hybrid SCM-DRAM single-level transactional system. Bailey et al. [5] 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 have used SCM to optimize OLTP durability management. Pelley et al. [18] propose to use in-place updates and group commits to optimize OLTP durability management and speed up recovery based on row store, disk-based OLTP database architectures. In their work, SCM is handled more as block device by paging than as byte-addressable memory. They also assume a DRAM cache for SCM, while SOFORT treats SCM as of the same level of DRAM, and goes further into adopting a single-level memory design, more inspired from column store, main-memory architectures than disk-based OLTP architectures. Fang et al. [11] propose an SCM-based new logging technique implemented on IBM SolidDB, which is a disk-based row store. SCM is used only to store the log. Our approach is radically different since we use SCM as memory and storage at the same time and we do not need a transaction log. Wang et al. [22] proposed a new SCM-based scalable distributed logging approach. All these works aim to improve existing systems based on classical database architectures, while we propose a green field approach that makes the most out of SCM.

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. [16] proposed Whole System Persistence (WSP), where data is flushed only on power failures using the residual energy of the system. However, contrary to SOFORT, they do not consider software failures, which are prominent. Zhao et al. [26] proposed Kiln, a persistent memory design that employs SCM to offer persistent in-place updates without logging or copy-on-write. While they assume a non-volatile last level cache, SOFORT makes no such hardware assumption.

Chen et al. [7] discuss how B+-trees and hash joins can benefit from SCM. Venkataraman et al. [20] contributed Consistent and Durable Data Structures that leverage the non-volatility property of SCM using versioning. They propose a versioned B-tree that is persisted in SCM in real time and that can be recovered in a consistent state after failure. These works are orthogonal, though very related to ours. Indeed, there is a need to redesign classical data structures to make them persistent and come up with new algorithms that will enable the full power of SCM.

7. CONCLUSION

In this paper, we have investigated how SCM can be leveraged next to DRAM to achieve instant recovery and remove the need for transaction rollbacks after failure by operating directly on the primary copy of the data. We have used our prototype SOFORT, which is a log-less, single-level, hybrid SCM-DRAM storage engine, to implement our proposed approach. We showed that instant recovery can be reached by two methods: by using persistent, primary data and leveraging secondary data structures that are being rebuilt in the background to answer queries, or by putting all secondary data structures in SCM. The former approach offers a high throughput during usual query processing but a lower throughput and a larger response time during recovery, while the latter approach offers a lower throughput during usual query processing than the former case but a small response time and a guaranteed throughput during recovery. There are many other distributions of secondary data structures over SCM and DRAM between these two extreme cases. In particular, workload analysis and partial reconstruction of transient index structures may lead to an optimal distribution.

For future work, we plan to investigate new recovery schemes for the hybrid SCM-DRAM cases by using more efficiently transient secondary data structures that have to be rebuilt during recovery. In that context, techniques such as Self-Managing Indexes (SMIX) have the potential to enhance the recovery performance [21]. We also plan to investigate how to achieve high availability with minimal cost for throughput and recovery time. Indeed, our proposed instant recovery is not sufficient by itself since it addresses only software failures. We plan to extend it to support hardware failures by investigating, among others, replication between different nodes. Finally, as shown in this paper, current indexing data structures do not perform well on SCM. We plan to investigate alternative indexing structures that are a better match for SCM.

8. ACKNOWLEDGMENT

We would like to gratefully thank Nica Anisoara for her helpful suggestions, fruitful discussions and thorough reviews. We also would like to thank Michael Rudolf, Ingo Müller, Marcus Paradies, Iraklis Psaroudakis, Norman May, as well as all the SAP HANA PhD students for their helpful reviews and suggestions. Finally, we would like to acknowledge the help of the SAP HANA development team, in particular Daniel Booss.

9. REFERENCES

[1] Disclosure of H/W prefetcher control on some Intel®processors. https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-\intel-processors.

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

[3] Intel Xeon Processor C5500/C3500 Series-based Platform for Storage Applications. Technical report, 2010. http://download.intel.com/design/intarch/prodbref/323306.pdf.

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

[5] 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*.

[6] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm. Implications of CPU caching on byte-addressable non-volatile memory programming. Technical Report HPL-2012-236, HP Laboratories, Dec. 2012.

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

[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 *ACM 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 *ACM/IEEE 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 *IEEE 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 *IEEE IEDM 2005*.

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

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

[15] 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. *IEEE Micro*, 2010.

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

[17] I. Oukid, D. Booss, W. Lehner, P. Bumbulis, and T. Willhalm. Sofort: A hybrid scm-dram storage engine for fast data recovery. In *DaMoN, in ACM SIGMOD 2014*.

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

[19] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB 2007*.

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

[21] H. Voigt, T. Kissinger, and W. Lehner. Smix: Self-managing indexes for dynamic workloads. In *SSDBM 2013*.

[22] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *PVLDB*, 7(10), 2014.

[23] T. Willhalm, I. Oukid, I. Müller, and F. Färber. vectorizing database column scans with complex predicates. In *ADMS, in VLDB 2014*.

[24] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units. *PVLDB 2009*, 2(1), 2009.

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

[26] 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 *IEEE/ACM MICRO-46*, 2013.

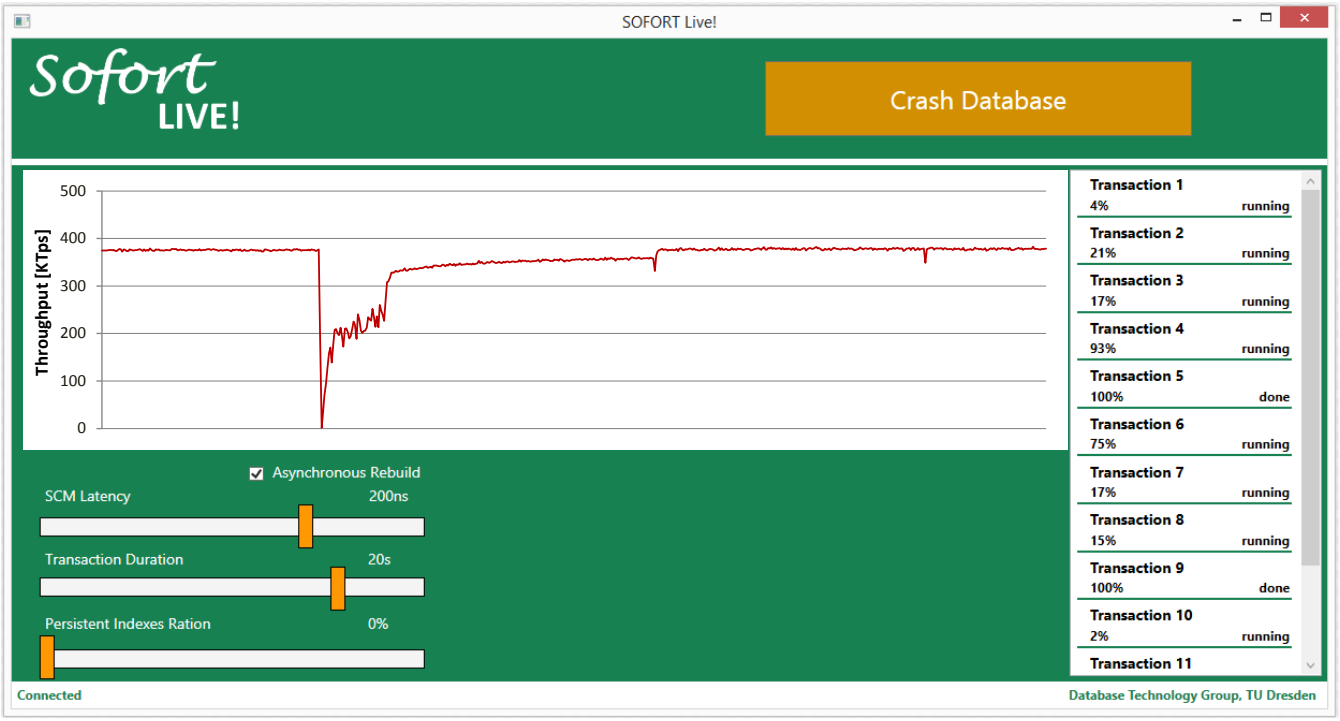


Figure 11: User interface of SOFORT demonstration proposal.

APPENDIX

A. DEMONSTRATION OF SOFORT

This paper is accompanied with a demonstration proposal of 10 minutes. It encompasses all the techniques presented in this paper that have been implemented in our prototype SOFORT. To help the audience understand better these techniques, we have built a front-end application, as depicted in Figure 11, that illustrates how we can trade between recovery performance and throughput performance. SOFORT will run on our SCM hardware simulator which is described in Section 2.1. In particular, we demonstrate:

Live recovery. We demonstrate how SOFORT recovers from random crash scenarios without losing data or corrupting it. In particular, we demonstrate how the persistent data structures of SOFORT can recover and execute a self-contained repair if needed. The demonstration includes sending *Linux* kill signals to a running instance of SOFORT during the execution of a benchmark and then recovering from the crash and continuing the execution of the benchmark as if the crash did not happen. This is due to SOFORT persisting the changes of every executed statement in real time. Put another way, every already executed statement is guaranteed to have persisted its changes.

Removing the need for transaction rollbacks after recovery. As a second step, we demonstrate the feasibility of removing the need for undoing running transactions after failure. To do so, we run a micro-benchmark consisting of long running transactions on SOFORT and then we crash the system. Recovery will show that we crashed in the middle of several long transactions but none have been rolled back. Even better, we resume execution exactly at the statement of the transaction when we crashed SOFORT.

Recovery performance/Query performance trade-offs. This is the main part of the demonstration. We use the TATP benchmark. The user interface includes a tuner to vary the number of the secondary data structures that are persistent in SCM and a tuner to vary the number of users. We vary the number of persisted secondary data structures and observe the effect on throughput. Then, for every configuration, we crash the database and observe recovery performance in terms of recovery area, recovery delta, throughput during recovery and response time. This will illustrate how DRAM favors query performance while SCM favors recovery performance.

Different SCM latency effects on SOFORT. Finally, we will show the effect of different SCM latencies on query performance and recovery performance. The hardware simulator we use allows to tune SCM latency via its microcode patch. We will vary SCM latency within a range and illustrate how higher latencies affect the overall system performance and how SOFORT manages to stay competitive even in the presence of very high SCM latencies.