MetaData Persistence using Storage Class Memory: Experiences with Flash-backed DRAM

Jithin Jose
Department of Computer Science
The Ohio State University
Columbus, OH
jose@cse.ohio-state.edu

Mohammad Banikazemi
IBM T.J Watson Research Center
Yorktown Heights, NY
mb@us.ibm.com

Wendy Belluomini
IBM Almaden Research Center
San Jose, CA
wb1@us.ibm.com

Chet Murthy
IBM Almaden Research Center
San Jose, CA
chet@us.ibm.com

Dhabaleswar K. Panda
Department of Computer Science
The Ohio State University
Columbus, OH
panda@cse.ohio-state.edu

ABSTRACT

Storage Class Memory (SCM) blends the best properties of main memory and hard disk drives. It offers non-volatility and byte addressability, and promises short access times with low cost per bit. Earlier research in this field explored designs exploiting SCM features and used either simulations or theoretical models for evaluations. In this work, we explore the design challenges for achieving non-volatility using real SCM hardware that is available now: Flash-Backed DRAM. We present performance analysis of flash-backed DRAM and describe the system issues involved in achieving true non-volatility using the system memory hierarchy which was designed assuming that data is volatile. We present software abstractions which allow applications to be redesigned easily using SCM features, without having to worry about system issues. Furthermore, we present case studies using two applications with different characteristics: an SSD-based caching layer used in enterprise storage (Flash Cache) and an in-memory database (SolidDB), and redesign them using software abstractions. Our performance evaluations reveal that SCM aware Flash Cache design could enable persistence with less than 2% degradation in performance. Similarly, redesigning SolidDB persistence layer using SCM improved the performance by a factor of two. To the best of our knowledge, this is the first work that evaluates SCM performance and demonstrates application redesign using real SCM hardware.

1. INTRODUCTION

The access time difference between memory and storage has been one of the major challenges in storage systems design. High performance systems are designed in such a way that this access time gap is hidden, through techniques such as buffering the data before it is being written to storage. Such techniques often sacrifice durability, consistency, or performance in balancing their use of memory and storage. Recent advancements in storage research have led to a new class of memory called Storage Class Memory (SCM), which combines the best properties of memory and storage. It offers non-volatility and byte-addressability with access times comparable to that of Dynamic Random Access Memory (DRAM).

It is widely believed that the evolution of SCM will have significant impact on current systems and software stacks [8]. Earlier research in this area has shown that SCM aware designs can improve the performance with high durability. These studies ([6], [18], [7], [19], [5], [16], [14]) considered different classes of software systems spanning over databases, key-value stores, file systems, and showed the possibility of performance improvements. But, these studies considered only simulations or theoretical models of SCM, because of lack of real SCM hardware.

In this work, we use available real SCM hardware - Flash Backed DRAM (FB-DRAM) - and present our experiences. We conducted detailed performance and persistence evaluations using FB-DRAM, which reveal new design challenges that were not identified in earlier studies. For instance, the presence of CPU memory hierarchy compromises memory write ordering and may even result corrupting the data in persistent memory. We discuss these challenges in detail and propose software abstractions that can be used for redesigning software systems taking advantage of SCM features.

Furthermore, we present case studies of two applications: SSD based cache called 'Flash Cache' and an in-memory data-base called 'SolidDB.' The original designs of Flash Cache did not consider persistence and, therefore the system requires battery backup in case of power failure. Similarly, the original design of SolidDB relied on hard disk drives (HDD) for persistence. We redesigned both of these applications to use FB-DRAM for persistence. Our experiment evaluations reveal that Flash Cache system was able to achieve persistence with less than 2% degradation in performance; and, SolidDB performance (latency and bandwidth)

INFLOW '13 Farmington, Pennsylvania, USA

was improved by a factor of two. For both of these applications, we verified our persistence design by pulling the power cable during the operation and verifying that the data is persistent after power is returned.

The rest of the paper is organized as follows. Section 2.1 presents some of the SCM technologies that are evolving and Section 2.2 describes FB-DRAM in detail. We list the challenges involved in using FB-DRAM and present generic software abstractions for efficient SCM aware designs in Section 3. In Section 4, we present case studies and illustrate how we redesigned applications using SCM features. The experiment results are presented in Section 5, where we present performance characteristics of FB-DRAM and application results. Finally, we conclude in Section 8.

2. BACKGROUND

2.1 Storage Class Memory (SCM)

There are several Storage Class Memory technologies that are currently being developed. Among the most prominent technologies are Phase Change Memory (PCM) [15], STT-RAM [4], M-RAM [10], R-RAM [2] and Flash-backed DRAM (FB-DRAM) [1]. In general, these memories (except for the FB-DRAM) are slower than DRAM access speed, but it is believed that in later generations, their performance will get closer to that of DRAM. In comparison with SSD technologies such as NAND Flash, they have better endurance and latency and provide higher granularity. In PCM, each cell stores information by changing the crystallization state of the phase-change material in the cell. MRAM technology uses magnetic fields to store data. Spin-Transfer Torque RAM (STT-RAM) is another emerging non-volatile memory technology that stores data as the magnetic orientation of a magnetic tunnel junction and has lesser power requirements. The basic idea behind RRAM technology is that, by applying sufficiently high voltages, an insulating dielectric can be made to conduct through a filament or conduction path. FB-DRAMs match the performance of DRAM by using DRAM as their main storage and provide persistence by using flash memory to backup (and restore) the contents of DRAM. These are different from Flash DIMMS [11], which is NAND flash in a DIMM form factor, but not on the memory bus.

The authors agree that the cost of FB-DRAM is not low enough to warrant their widespread use and hinders FB-DRAM's to be classified as SCM, since SCM's are considered to be cheaper than DRAM. But FB-DRAM provide all the desired features of SCM and these are becoming commercially available. These facts make them the best vehicle for building prototypes of systems with persistent memory.

2.2 Flash-Backed DRAM (FB-DRAM)

Flash-backed DRAM (FB-DRAM) consists of a DRAM, NAND flash, and an ultracapacitor. A high level architecture diagram of FB-DRAM is presented in Figure 1 [1]. Application read-write operations are directly acted upon the DRAM mapped memory. From application perspective, it is just another address space. The entire system is controlled by a control unit called Subsystem Control Unit (SCU).

SCU gets 'power-failure' signal through Powerfail Sideband Communications channel, when system loses power. SCU reacts to this signal by initiating backup operation, in which the entire contents of DRAM are backed up to FLASH memory. The operation is powered by the Ultracapacitor Storage Array. When the power is back, application can request to restore the data back to DRAM, in which the DRAM contents are restored to the state during power failure. Thus, FB-DRAM achieves non-volatility while providing the same access speed as that of DRAM. This qualifies FB-DRAM as Store Class Memory.

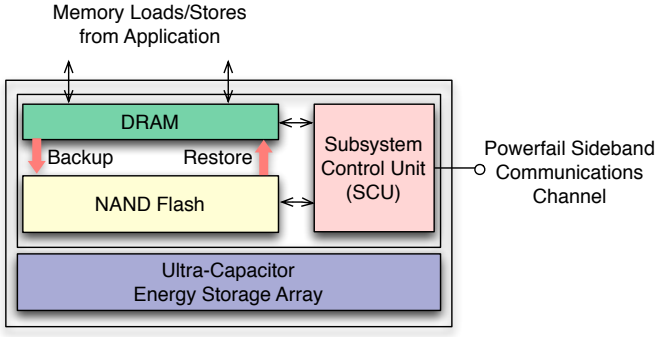


Figure 1: Flash Backed DRAM System Architecture

FB-DRAM consists of a software driver, which maps its DRAM address to a separate address space. Applications can request for this address space and can access this memory along with volatile main memory. The software driver provides access interfaces to get FB-DRAM address space pointer and do explicit backup/restore operations.

A pointer into the FB-DRAM address space pointer is used in the same way as a standard pointer. From the CPU point of view, FB-DRAM address space is indistinguishable from main memory and sits below the CPU memory hierarchy; just that the memory load/store operations are acted upon the DRAM in FB-DRAM. Applications can assume that the writes to this address space is non-volatile. However, the presence of the memory hierarchy imposes different challenges that need to be addressed for efficient and correct use of FB-DRAM, which are described in Section 3.1.

3. APPLICATION DESIGN USING FB-DRAM

Since the performance of SCM is far from being 'fast enough' to be used in CPU memory hierarchy (L1-L3 caches), systems with SCM will have both volatile and non-volatile memory types in their memory hierarchy. This leads to several challenges that need to be addressed in order to use SCM efficiently. In this section, we describe these challenges first, and then present the software abstractions that we used to redesign applications with flash-backed DRAM's.

3.1 Design Challenges

Data Loss due to CPU Cache: The slower speeds of SCM memory (FB-DRAM) mandates the presence of high speed caches in memory hierarchy. Thus, the memory store operations might not get reflected in FB-DRAM immediately. If the power loss happens at this stage, the data might get lost. Such loss in data can corrupt data structures and can cause the recovery process to hang or crash [5]. This compromises non-volatility.

Write Re-ordering: Another significant side effect of using caches in the memory hierarchy is write-reordering. Depending on the application behavior and cache replacement

algorithms, the order of writes to the memory system can be different than the actual order in which writes were issued. This can lead to inconsistency in SCM contents, which can mislead crash-recovery algorithms or cause data loss or corruption. This in turn can lead the system to inconsistent state after a system crash. Therefore, it is crucial to provide mechanisms that enforce ordering among multiple writes or groups of writes.

Writes across Cache-Line Boundaries: The smallest unit of data transfer between the main memory and caches is a cache line. If the data corresponding to a write spans across multiple cache lines, it is possible that some of these get flushed to DRAM, while others remain in cache. This can lead to data corruption, as the data in cache lines is lost.

Virtual Address of FB-DRAM mapped memory: As discussed in Section 2.2, the FB-DRAM software driver maps the DRAM address space and presents the virtual address to application layer. This virtual address will be different at each system start-up. ie, even though the data is persistent across power cycles, the address changes across power cycles. Thus, pointer references within non-volatile address space becomes invalid after a power cycle. Furthermore, application might require non-volatile memory for different data objects. So, how can the application know the new address of a particular data structure, after a power cycle?

We address these challenges in a diligent manner and design an application interface for using flash-backed DRAM's. We present the detailed design of the application interface, which abstracts the FB-DRAM access in a simple and efficient manner, in the following section.

3.2 Software Abstractions

In this section, we present the software abstractions which enable redesigning of applications without having to worry about underlying system issues and challenges in achieving volatility. We classify the abstractions as management and data access operations.

3.2.1 Management Operations

Initialize/Finalize Operations: The initialize operation initializes the FB-DRAM driver and maps the FB-DRAM to a virtual address space. It also initializes the 'Tag table' (explained below). The finalize operation does defragmentation of FB-DRAM address space, backs up data to Flash, and finally closes the FB-DRAM device. The initialize and finalize operations are directly implemented over FB-DRAM driver.

Metadata for persistent memory management: As indicated in previous section, the address of FB-DRAM mapped region might be different for each initialize call, but contents at same offset remain unchanged. Because of this variability in address, applications cannot rely on just memory address for identifying persistent data. Further, applications cannot rely on offset within the persistent address space because they cannot store offsets between power cycles. We employ 'tags' for identifying persistent memory. In this model, every persistent memory allocation request is associated with a name tag. A special table called 'Tag Table' keeps track of the entries. Tag table is kept at a pre-defined offset in FB-DRAM address space. Each entry in the table contains the name-tag, offset at which the actual data is stored, and its size. An entry is created for each successful allocation request, and are removed when the memory is freed. A snapshot of tag table is presented in Figure 2.

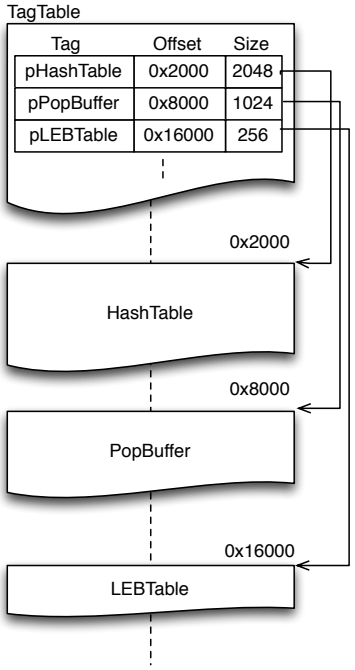


Figure 2: Tag Table

Tag table helps to easily recover data after a power cycle. Application can just request the data along with the name-tag, and the data can be identified using the offset in tag table entry. Furthermore, the table helps in defragmenting FB-DRAM address space. By a mere scan of tag table, holes in FB-DRAM memory space can be easily identified.

3.2.2 Data Access Operations

Based on the access granularity, we categorize the data access operations into *Direct Updates*, *Logging* and *Checkpointing*, and present abstractions for these operations. Each of these are explained in the following sections.

Direct Updates: Direct updates are for small updates, comparable to cache-line granularity. These updates operate directly upon the FB-DRAM memory. Such data objects are allocated directly from the non-volatile memory. The updates are ensured to be reliably reflected into FB-DRAM. As discussed in Section 3.1, we need to make sure that every memory update operation shall reach the DRAM associated with FB-DRAM, because of the presence of caches. Thus we need to flush the cache lines to memory. If the data spans across multiple cache lines, all these cachelines are flushed to memory.

We compared two approaches for ensuring reliability — 'write-through' cache, and '`sfence`+`clflush`' instructions. Our performance evaluations (Section 5) reveal that the later approach provides lesser overhead, and we select this in our implementation. As opposed to the simulation based studies, we evaluate the performance and select the flushing scheme. The flushing scheme selection is made configurable, since the flush latency can be different on different platforms.

Logging: It is quite expensive to flush the data to FB-DRAM in case of large updates. For such updates, we propose *Logging* based scheme. In this approach, actual data

is kept in main memory, but every update is logged to FB-DRAM. The logs are assumed to be much smaller than the actual update, and these are flushed to FB-DRAM using `sfence+clflush` operations. Thus, the overhead during update operation can be reduced significantly, while maintaining persistence. During the recovery operation, these logs can be replayed and the data structure can be restored to the state before power loss.

Checkpointing: Checkpointing can be employed for data structures, whose updates are large and frequent. *Checkpointing* is usually used along with *Logging*. In this scheme, checkpoints are made at regular intervals and are kept in FB-DRAM. Such checkpoints are flushed to make sure that they are reflected in FB-DRAM. All the logs prior to the checkpoint are cleared after a successful checkpoint. During recovery operation, the checkpoint is recovered and logs are replayed on this checkpoint to restore the data structure to the state before power-loss.

4. APPLICATION CASE STUDIES

We use the software abstracts presented in Section 3.2 to redesign Flash Cache and SolidDB. We present these case studies in this section.

4.1 Flash Cache

Flash Cache is an SSD-based caching layer used in enterprise storage systems (similar to the one used in IBM XIV Storage System [12]), as represented by Figure 3. A software layer called 'Shim' intercepts I/O from DRAM cache layer to the RAID layer of HDD's; and are redirected to 'Flash Cache Manager.' Flash cache manager implements caching function using a series of SSD's and takes care of garbage collection of cold/invalid flash blocks, de-staging cold data to HDD's and reliability. It keeps a hash table to map between disk and flash addresses and keeps an array called the 'LEB (Logical Erase Block) Array' and records which pages are modified with respect to data on the disk. For aggregating writes, flash cache manager keeps a set of buffers called 'Populate Buffers'. New writes/updates from the Shim as well as garbage collected data is buffered in these, before being written to SSD. Once a buffer is full, it is written to flash in an asynchronous manner.

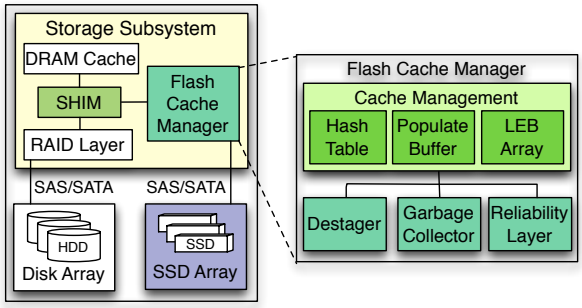


Figure 3: Flash Cache Architecture

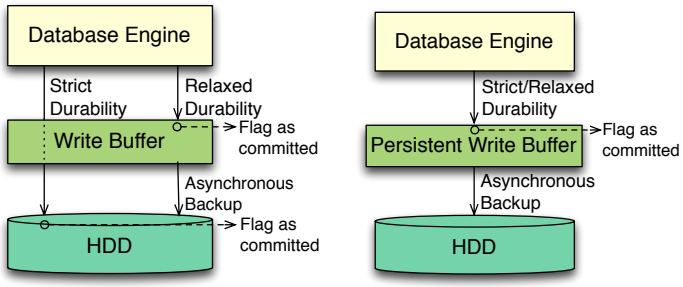
Even though the data is stored in persistent memory (SSD), metadata information - such as the Hashtable, Populate Buffers, and LEB array - is stored in the non-volatile memory (DRAM). Thus, system requires batteries to preserve data in case of power loss. We redesigned flash cache using software abstractions described in Section 3 to achieve persistence without batteries.

In Flash Cache, hashtable lookups/updates are very frequent; it gets accessed during all I/O operations. So we employed *Logging* and *Checkpointing* to make it persistent. We kept the hashtable in DRAM and every update to hashtable is logged in persistent memory. After a predefined number of log updates, the hashtable checkpoint is created in the persistent memory. Populate buffer read/write frequency is comparatively lesser. Thus, populate buffer and the associated metadata is kept in persistent memory (updates using *Direct Update*). Similarly, the LEB arrays are also kept in persistent memory. Every update to these regions is followed by a flush operation.

On recovery, first the hashtable is recovered to latest state by restoring the checkpoint and replaying the logs. Next, populate buffers and the associated metadata are recovered. Populate buffer data pages which were being filled up are identified and the valid pages are written into SSD. Buffers which were being written to SSD are re-written and those which are already written are ignored. After this, populate buffers are cleaned up and made available for aggregating future writes. LEB buffers, which are stored in persistent memory, are up-to-date with the LEB states and cache hit information. With these recovery steps, the flash cache is just up-to-date with all the pages in SSD as if no crash/power-failure had happened. In this work, we overlook dealing with replication of metadata to protect against node failure. We limit the scope to measure the impact of changes needed to make memory persistent.

4.2 SolidDB

SolidDB is an in-memory database optimized for high speed. It stores transaction logs in HDD or SSD and offers two modes of durability: strict and relaxed as shown in Figure 4(a). These modes depend on when a transaction is marked as complete. In relaxed mode, a write is marked as complete when the data is written to a *write buffer*. Obviously, if there is a system crash and the log records are not written to the disk, the committed transaction will be lost in this mode. In the strict durability mode, after creation of each log record in the write buffer, a synchronous write is issued to write it into the disk. After the disk write is complete, the transaction is marked as complete.



(a) Original Design (b) SCM Aware Design

Figure 4: SolidDB Commit Design

We enhanced the implementation of this database system such that the *write buffer* is kept in persistent memory. This

provides the same durability level as that of strict mode while achieving performance similar to that achieved with relaxed mode. Figure 4(b) illustrates this design. Every write to *write buffer* is flushed into FB-DRAM to guarantee ordering and persistence. This design uses *Logging* abstraction described in Section 3.2.2. As log records are written to the persistent write buffer, a background thread moves them to the log files stored on the main storage.

During the recovery phase, SCM memory is restored to its content before the crash. Log records from write buffer which are not transferred to main storage are identified and written to main storage. Care is given such that potential crashes during recovery are detected. For example, during the recovery process the FB-DRAM is automatically set not to save and restore the content of the memory if another crash occurs.

5. EXPERIMENT RESULTS

Experiment Platform: We used an eight core AMD Opteron node for our evaluations. It consists of 32 GB of DRAM memory, with 1.3 MHz speed and 4 GB flash backed DRAM (1.3 MHz). It is equipped with a 240 GB Sandisk SSD and 500 GB Toshiba HDD. The operating system used is Debian Linux with kernel version 2.6.32.

5.1 FB-DRAM Performance Analysis

We use Rambench benchmark [17] for FB-DRAM performance evaluation. The benchmark spawns multiple threads and each thread issues memory write operations. Memory access width is configured as 64 bits. We vary the number of threads from 1 to 16, and the bandwidth results are reported. Performance results of sequential and random write accesses are presented in Figure 5. Since the memory reads are directly operated upon the DRAM of FB-DRAM, the read performance is exactly the same as that of normal DRAM. Thus we present only the write performance results.

In Figure 5, we compare the performance of normal DRAM with that of FB-DRAM. For normal DRAM, we flush every write operation using `sfence`+`clflush` instructions. This is indicated as 'RAM (flush).' We present the FB-DRAM performance evaluations with two configurations: flush every write using `sfence`+`clflush` instructions (indicated as 'FB-DRAM (flush)', and FB-DRAM in 'write-through' mode (indicated as 'FBDRAM (WT)).'

The bandwidth increases with increase in number of threads and finally saturates. It can be observed from the results that the sequential write bandwidth of RAM (flush) saturates at around 0.25 GB/s. However, the FB-DRAM with the same configuration saturates at a lower bandwidth of about 0.21 GB/s. Results indicate higher overhead for FB-DRAM configured in write-through mode. The same pattern is observed for random write experiment as well. We choose this 'FB-DRAM (flush)' as the configuration in our application case studies, since it provides best performance for FB-DRAM. There is a slight performance difference between FB-DRAM and normal DRAM, with the fence+flush operation. We believe that this is because these DRAM's (main memory and DRAM in FB-DRAM) are from different vendors.

5.2 Flash Cache Performance Analysis

We used a synthetic benchmark to evaluate Flash Cache performance. In this benchmark, multiple threads are launched and each thread issues I/O requests. The benchmark reports the total execution time and the number of I/O operations per second. For this experiment, we configured number of I/O threads to be 8 and total number of operations to be 80,000 with read-write ratio of 1:1.

Performance results shown in Figure 6 report both execution time and the number of I/O operations per second (IOPS). 'Base Version' denotes the unmodified flash cache version. For the persistence enabled versions, we ran the experiment under two configurations - a) FB-DRAM in 'write-back' mode and every write is followed by flush operation ('FB-DRAM (flush)') and b) FB-DRAM in 'write-through' mode ('FB-DRAM (write-through)'). We did this to confirm that the performance results shown in FB-DRAM evaluation benchmarks matches even at application level. The execution time reported for Base Version, FB-DRAM (flush) and FB-DRAM (write-through) are 603, 615 and 622 seconds respectively. This indicates that the Flash Cache design using FB-DRAM is able to achieve persistence with less than 2% degradation in performance.

5.3 SolidDB Performance Analysis

We used a synthetic benchmark to measure average latency and bandwidth for a group of transactions with SolidDB. These comprise of create, update and delete operations. Since the performance of original design depends on disk drive performance, we present the performance results with different types of drives (Table 1). In particular we used a traditional hard disk drive, a FusionIO ioDrive (SLC) [9], and an experimental PCM prototype drive (SLC) [3]. These are represented as 'Standard HDD', 'FusionIO ioDrive' and 'PCM Drive', respectively. Enhanced SCM aware design is represented as 'FB-DRAM'. It can be observed that the enhanced design using FB-DRAM reduces operation latency by a factor of two when compared to the original design even with the fastest SSD's.

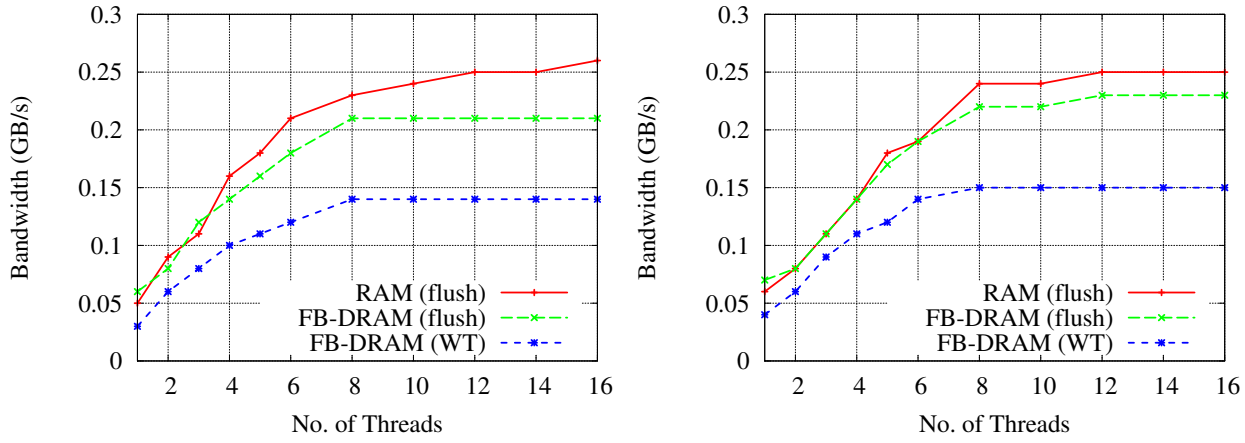
Table 1: SolidDB Update Operation Latency (μs)

	Standard HDD	PCM Drive	FusionIO ioDrive	FB-DRAM
Latency	46948.5	415.0	267.6	141.3

We also measured SolidDB throughput with same configurations (Figure 7). The performance achieved for enhanced SCM aware design in strict durability mode is almost twice that of the fastest storage devices. The performance of the system with relaxed durability is not shown in the figure as it is practically identical with that of system with Flash-backed memory and strict durability. Thus, we could attain the performance achievable in relaxed durability mode, while providing the strict durability required by many database applications.

6. VALIDATION

We validated both the applications for correctness and persistence. For persistence validation, we pulled the power cable during the operation to simulate real power failure, and verified that the data is persistent after power is returned. We conducted this experiment multiple times, and ensured that persistence and correctness are maintained.



(a) Sequential Write

(b) Random Write

Figure 5: FB-DRAM Write Performance

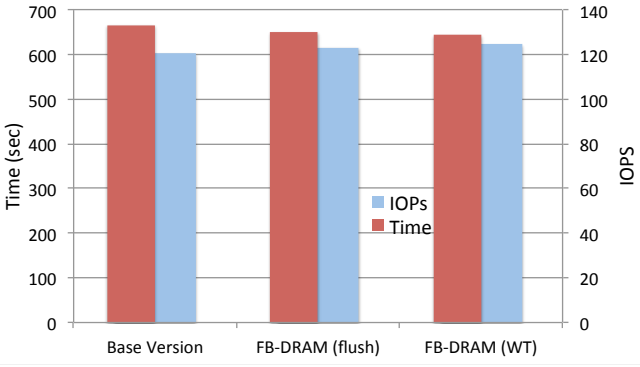


Figure 6: Flash Cache Performance Results

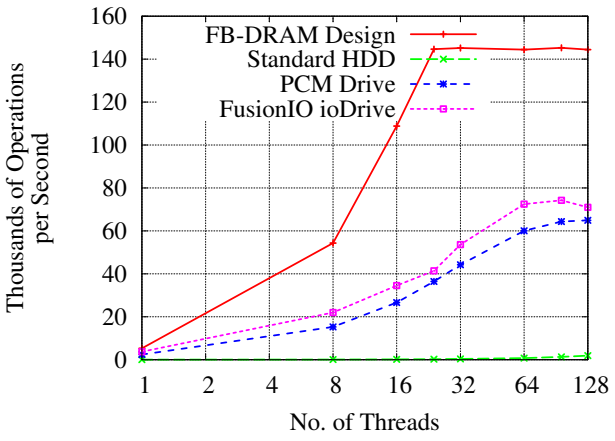


Figure 7: SolidDB Operation Throughput Results

7. RELATED WORK

Storage Class Memory and its applicability in different software stacks have been actively researched and it has resulted in valuable studies. NV-Heaps [5] is a light weight persistent object system that provides transactional semantics and provides a persistence model. BPFS [6] presents a persistent file system, which is designed around the properties of persistent byte addressable memory. Both these studies throws light into the design of efficient SCM aware applications and provides valuable insights about SCM usage. In fact, many of our design considerations are inspired by these studies. However, these studies assumed special hardware support such as epoch barriers for ordering writes, where as our design relies of fence operation for ensuring ordering.

Mnemosyne [19] is another valuable study, which presents a simple interface for programming with persistent memory. It also throws light into the issues associated with redesigning applications with SCM features. Mnemosyne requires modifications to Linux kernel for allocating and virtualizing SCM pages. Fang, et. al studied the use of SCM for high performance database logging in DBMS, and presented good insights about SCM aware application design. Several other studies (STeTSiMS [16], [14]) enables designers to explore the potential of recent SCM designs and adjust the performance without needing a detailed understanding of underlying SCM technology. Narayan et. al. studied the whole system persistence using NV-RAM [13]. Their approach was to provide persistence at the system level than at the application level. Keeping the entire system memory may lead to permanent (unrecoverable) corrupted state. Also, they flush data only during power failure and do not consider write re-ordering, which can lead to corrupt system state.

However, all these studies considered only simulations or theoretical models of SCM, because of lack of real SCM hardware. We present our experiences with redesigning applications using real SCM that is available. We unearth new design challenges, and design an initial prototype for abstracting SCM memory, and re-design applications using this.

8. CONCLUSION AND FUTURE WORK

In this paper, we presented our initial experiences with Flash-backed DRAM's for enabling persistence. We described

the system issues involved in achieving true non-volatility and propose software abstractions for redesigning software systems. We redesigned Flash Cache and SolidDB using these abstractions to enable persistence. Experiment evaluations reveal that the Flash Cache system was able to achieve persistence with less than 2% degradation in performance and the SolidDB performance was improved by a factor of two. For both these applications, we verified our persistence design by pulling the power cable during the operation and verified that the data is persistent after power is returned.

We plan to continue working along this direction. We plan to improve our prototype to support multiple applications simultaneously, and to support dynamic memory objects such as linked lists, without the need for name-tag. We aim to consider efficient memory defragmentation schemes without hindering application performance. We would also like to redesign more applications with different characteristics and enable persistence.

9. REFERENCES

[1] Agiga Tech. Finding the Perfect Memory. http://www.agiga tech.com/pdf/pdf_WhitePaper_FindingPerfectMemory.pdf.

[2] H. Akinaga and H. Shima. Resistive Random Access Memory (ReRAM) Based on Metal Oxides. *Proceedings of the IEEE*.

[3] M. Athanassoulis, B. Bhattacharjee, M. Canim, and K. A. Ross. Path processing using Solid State Storage.

[4] E. Chen, D. Apalkov, Z. Diao, A. Driskill-Smith, D. Druist, D. Lottis, V. Nikitin, X. Tang, S. Watts, S. Wang, S. Wolf, A. Ghosh, J. Lu, S. Poon, M. Stan, W. Butler, S. Gupta, C. Mewes, T. Mewes, and P. Visscher. Advances and Future Prospects of Spin-Transfer Torque Random Access Memory. *IEEE Transactions on Magnetics*, 2010.

[5] 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. *SIGARCH Comput. Archit. News*.

[6] 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 *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*.

[7] R. Fang, H.-I. Hsiao, B. He, C. Mohan, and Y. Wang. High Performance Database Logging using Storage Class Memory. In *2011 International Conference on Data Engineering*.

[8] R. F. Freitas and W. W. Wilcke. Storage-Class Memory: The Next Storage System Technology. *IBM Journal of Research and Development*, 2008.

[9] Fusion IO. Fusionio Drive Specifications. http://www.fusion io.com/load/media-docsProduct/kcb62o/Fusion_Specsheet.pdf.

[10] W. Gallagher, D. Abraham, and et. al. Recent Advances in MRAM Technology. In *VLSI Technology, 2005. (VLSI-TSA-Tech). IEEE VLSI-TSA International Symposium on*, 2005.

[11] Hewlett Packard. Flash DIMM Technology. http://www.stethos.com/flashmemory/data/paper.pdf.

[12] IBM XIV Storage System. http://www-03.ibm.com/systems/storage/disk/xiv/index.html.

[13] D. Narayanan and O. Hodson. Whole-System Persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, 2012.

[14] M. K. Qureshi, S. Gurumurthi, and B. Rajendran. *Phase Change Memory: From Devices to Systems*. Synthesis Lectures on Computer Architecture. 2011.

[15] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam. Phase-Change Random Access Memory: A Scalable Technology. *IBM Journal of Research and Development*, 2008.

[16] C. W. Smullen, IV, A. Nigam, S. Gurumurthi, and M. R. Stan. The STeTSiMS STT-RAM simulation and modeling system. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '11, 2011.

[17] T. Kaldewey, A. Blas, J. Hagen, E. Sedlar, S. Brandt. Memory Matters. In *Work in Progress in the 29th IEEE Real-Time Systems Symposium (RTSS)*.

[18] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *Proceedings of the 9th USENIX conference on File and stroage technologies*.

[19] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the 16th international conference on Architectural support for programming languages and operating systems*, ASPLOS '11.