Bridging the Programming Gap Between Persistent and Volatile Memory Using WrAP*[†]

Ellis Giles
Rice University
Houston, Texas 77005
erg@rice.edu

Kshitij Doshi
Intel Corp.
Portland, OR
kshitij.a.doshi@intel.com

Peter Varman
Rice University
Houston, Texas 77005
pjb@rice.edu

ABSTRACT

Advances in memory technology are promising the availability of byte-addressable persistent memory as an integral component of future computing platforms. This change has significant implications for software that has traditionally made a sharp distinction between durable and volatile storage. In this paper we describe a software-hardware architecture, WrAP, for persistent memory that provides atomicity and durability while simultaneously ensuring that fast paths through the cache, DRAM, and persistent memory layers are not slowed down by burdensome buffering or double-copying requirements. Trace-driven simulation of transactional data structures indicate the potential for significant performance gains using the WrAP approach.

Categories and Subject Descriptors

D.4.2 [Storage Management]: [Storage Hierarchies]

General Terms

RELIABILITY, PERFORMANCE

Keywords

Atomic transactions, phase change memory, persistence, programming, storage class memory

1. INTRODUCTION

This paper examines the use of byte-addressable persistent memory (referred to as Storage Class Memory [10] or SCM) as a replacement for traditional non-volatile storage (hard disks or SSDs) in emerging data management systems. There has been a growing use of DRAM servers to accelerate these applications, allowing their performance to scale well beyond the limits of traditional

implementations. These solutions either use DRAM servers as main-memory caches (e.g. **memcached** [9]) or employ in-memory database technology to operate almost entirely from DRAM. Popular examples of the latter include SAP HANA [8], IBM solidDB [1], voltDB [3], and Neo4J [2]. However, the volatile nature of DRAM makes these main-memory systems vulnerable to system crashes. Preventing loss of data requires additional (often ad-hoc) techniques to maintain a persistent copy of the data on non-volatile storage. This incurs overheads that reduce the benefit of in-memory operation. Furthermore, recovery times following a crash or scheduled maintenance are long, as the in-memory structures need to be rebuilt from non-volatile storage.

Nascent SCM technologies [10] like Memristors, Spin-Torque MRAM (ST-MRAM), or Phase Change Memory (PCM) provide a potential solution to this problem. PCM devices can be used to create memory that resembles DRAM in form factor, speed, and access characteristics, but with the non-volatility, cost, and power consumption of storage devices like hard disks. Continuing advances in SCM technology hold the promise of overcoming problems of reliability and wear-endurance [19, 23], write latency [19, 20] and read performance [18], increasing anticipation that storage-class memory will become available in commodity computing platforms in the near future.

While exploiting main-memory provides tremendous performance advantages, these solutions do not fully address the fundamental problem of data persistence or durability; *i.e.* guaranteeing that the results of a committed transaction are not lost due to system crashes caused by software and hardware errors or power failure. If a cache server fails, performance is disrupted as the data in the cache is rebuilt from the back-end database. In addition, all updates need to be made to the slow database, rather than just updating cache, to ensure that the data are persistent in the event of a cache failure. Similar issues arise in an in-memory database implementation. To guarantee persistence, some form of logging and checkpointing to disks is bolted on to ensure that data can be recovered in case the database state in main memory is lost. A recent proposal to do away with non-volatile storage is the use of RAM-clouds [17] in which persistence is obtained by keeping multiple in-memory copies of the data on different servers. The power and energy costs of keeping large amounts of passive data continuously refreshed in DRAM, compared to the use of non-volatile storage to hold inactive portions of the database, is a significant consideration in adopting this approach. There is also need for high-speed networking to keep the copies synchronized.

SCM technology provides an ideal solution that combines the cache-line access of DRAM with the persistence of disk. This makes it possible to use fine-grained RAM algorithms and data structures, without worrying about either the need for blocking

*Partly supported by the National Science Foundation under NSF Grants CNS 0917157 and CCF 0917226.

[†]A preliminary version of this paper was presented at the Memory Architecture and Organization Workshop, Tampere, 2012.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CF'13, May 14–16, 2013, Ischia, Italy.

Copyright 2013 ACM 978-1-4503-2053-5 ...\$15.00.

these structures for disk access, or the loss of data due to a system crash. A SCM-based memory cache implementation can employ the cache for both reads as well as updates since the data in the cache is persistent. In case of failure, rebooting the system will make the cached data instantly available for use. Similarly, in-memory databases can avoid the complexity associated with adding disk-based checkpointing as a separate mechanism, since the data stored in the SCM is non-volatile.

The rest of the paper is organized as follows. Section 2 discusses the problems faced in using SCM as a persistent replacement for DRAM, and reviews existing approaches to address these problems. In Section 3 we propose a new solution WrAP (**Write Aside Persistence**), which is based on propagating updates simultaneously through the fast cache hierarchy as well as along a slower asynchronous channel to SCM. The solution avoids changes to the front-end of the well-understood cache hierarchy, and leverages the SCM controller to obtain the desired behavior in a non-intrusive manner. Section 4 provides the results of a simulation-based empirical evaluation that showing that WrAP-based atomic persistence is significantly faster than traditional software-based approaches based on undo logs. We conclude with Section 5.

2. OVERVIEW

In traditional computer systems there is a sharp distinction between accesses made to volatile and persistent storage. Applications can directly access the former using regular memory instructions (loads and stores), while accesses to durable storage are arbitrated by the file system or a database system. The arbitration allows the system to ensure that the ACID (atomicity, consistency, isolation and durability) requirements of transactions are met, by coordinating the component and concurrent activities. In addition, the interposed access provides protection of the underlying data from spurious writes by malicious or erroneous processes.

We discuss three issues that need to be addressed by software for correctly implementing persistence. The techniques needed to handle these efficiently are different when the non-volatile store is a memory-bus-based SCM device that is accessed using load and store instructions, compared to traditional disk-based stores where all accesses to disk are arbitrated by intervening system software.

- **Persistence Ordering.** Updating persistent data structures imposes additional constraints on the *ordering* of statements, due to the possibility of failure at arbitrary points in the program. For instance, setting a persistent pointer variable to the address of an uninitialized block of storage can result in an undetectable error if the system crashes between the pointer update and the initialization of the block. However, switching the order of updates, so that the block is initialized before the pointer, maintains consistency even after a failure-induced reboot. Note that persistence ordering requires that the updates must be propagated all the way to the persistent memory in the specified order. It is not sufficient to just order the global visibility of these updates as in typical memory consistency protocols. Additional hardware support may be needed to ensure this memory behavior, as discussed in Section 2.1.
- **Persistence Atomicity.** Transactional semantics require that updates to a set of related records must always occur as a group; either all the records must be updated or none of them should. Canonical examples include the updates to account balances when transferring funds between bank accounts, or coupled pointer-swizzling operations while updating dynamic data structures. Since failure may occur at

any time, the system must have some way of backing out from a partial set of updates, or must defer the updates until all values have been safely recorded in a power-fail-safe region. Traditional software systems perform transactional updates by making system calls to an underlying file system or database, which uses disk-based record logging or copy-on-write-based mechanisms to ensure that the updates are applied indivisibly, and are always recoverable once the transaction has committed.

- **Persistence Protection.** Programming bugs in a persistent memory system can be insidious. Not only does the persistent nature of the changes make it impossible to simply reboot to a consistent memory state, but subtle pointer dependencies between data structures spread over volatile and non-volatile memory regions of memory, increase the challenge of robust programming tremendously [5].

In the next section we discuss previous approaches to addressing these issues. Our approach, discussed in Section 3, is directed towards the general atomicity problem, and provides a new software-hardware approach.

2.1 Related Work

The problem of write **ordering** has been addressed in several papers [6, 22, 15, 21]. In BPFS [6], a new mechanism called *epoch barriers* was proposed for ordering updates. A cache line is tagged with an epoch number, and the cache hardware is modified to guarantee that memory updates due to write backs are always done in epoch order. In Mnemosyne [22], the ordering of writes to persistent memory is controlled by software using a combination of non-cached write modes, cache line flush instructions, and memory barriers (fence instructions). Note however, that fence instructions which only ensure global visibility will need to be enhanced to ensure that completion of the fence implies that the pending writes have also been committed to persistent memory (or at least a power-fail-safe region). A lightweight hardware mechanism called *cache line counters* is proposed in [15], which allows software to query whether all the writes belonging to a specified set have been committed to memory, and to delay dependent updates accordingly. Finally, a software primitive *flush* is proposed in [21], that is used in conjunction with a memory fence instruction to allow software to order its updates. Novel implementations of persistent logs based on ordering primitives have been proposed in [22, 7]. In this paper, we use ordering primitives sparingly, and only to ensure that the update trail of a transaction has been logged to a power-safe region of persistent memory before the transaction commits. The persistent log structures proposed in [22, 7] can be simplified and adapted here.

Mechanisms to enforce **persistent atomicity** in the literature rely on hardware support in the form of *atomic 8-byte writes*, so that single scalar variables can always be updated indivisibly by a store instruction. This basic hardware primitive is used to construct atomic update mechanisms for larger data structures. Atomic updates for tree-structured file systems was proposed in BPFS [6] using short-circuit shadow paging. Copy-on-write semantics trigger the copying of a block that need to be updated. All blocks on the path till the lowest common ancestor of the set of changed blocks are copied and updated, and the copies are finally linked into the tree by a single pointer switch in the parent of their common ancestor. In Mnemosyne [22], atomicity is handled by executing applications under the control of an underlying software transactional memory system (STM). Since the STM handles atomicity as part of its concurrency control activities, it can be leveraged to handle

persistence atomicity as well. A third approach to handle atomicity is based on the use of versioning [21]. CDDS [21] proposes the design of a persistent multi-version B-Tree that maintains several versions of the database at any instant. Reads are performed on the current database version. An update transaction creates new versions of all the data blocks it is modifying and timestamps them with the new version number. After all the updates in this transaction have been completed, the version number of the database is updated. These proposals, and others like Moneta [4], employ a traditional block-based interface for accessing persistent memory. They do not address the problem of efficient atomic updates in a load and store architecture, and require considerable software intervention in the access path. For instance, BPFS and CDDS are block-oriented designs for file systems, and rely on accessing blocks by a unique path from a common root node, while Mnemosyne is designed for use with STM systems. Whole System Persistence (WSP) [16] advocates the use of hardware that can flush the entire transient state of the system to durable storage. However, this does not obviate the need for atomicity-preserving mechanisms to handle software crashes or user-induced aborts of transactions.

Finally, **robust persistence** has been addressed in NV-Heaps [5] and in [15], and mechanisms to increase the reliability of software running on SCM-based systems have been studied recently.

3. OUR APPROACH

The central problem addressed in this paper is obtaining atomicity of a group of store operations to arbitrary addresses of byte-addressable persistent memory in the presence of unpredictable failures. Our aim is to support general transaction processing software without making too many assumptions about how the software is structured. This is necessary to allow a broad range of both legacy software and new in-memory data management applications to take advantage of SCM-based systems rather than restricting its applicability only to specialized classes of software.

An overview of the system architecture is shown in Figure 1. A range of the physical address space is occupied by SCM memory modules rather than DRAM. Addresses in this range are intercepted by the SCM memory controller that is responsible for managing the SCM devices. We will leverage the controller in the path to provide efficient atomicity. Since accesses to variables in this architecture are performed directly to main memory, rather than to a disk through a layer of operating system or database software, the mechanisms for supporting atomicity must be correspondingly lightweight and fast.

Figure 1: Physical Address Space has SCM and DRAM

The basic problem in trying to make updates atomically is shown

in Figure 2 (a). A transaction begins at time denoted by **S**, makes a series of writes to different variables, and signals its intent to commit at time **C**. The write instants are denoted by the small vertical bars in the timeline. These writes cannot be allowed to update SCM memory until the transaction commits, since it may abort midway either voluntarily or due to a system crash. These deferred writes must then be all written persistently and atomically before the transaction can end at time **E**. During the interval that the transaction is active (between **S** and **C**) it should be able to efficiently re-read the values of the updated variables. Other concurrent transactions may also, depending on the isolation guarantees made by the system (*e.g.* read uncommitted mode), be allowed to read these intermediate writes. When strong isolation modes (*e.g.* serializable or read committed) are used, concurrent transactions reading these variables will need to be delayed until time **E** when all the writes are made persistent, increasing their latency and decreasing transaction throughput.

Figure 2(b) shows how the shadow copying approach (essentially an undo log) would operate in this situation. Before a location in persistent memory can be updated, a copy of the old value is made and saved in non-volatile memory. This is a synchronous copy operation that needs to be completed before the write can proceed. The transaction commit time **C** is delayed in this case, because of the synchronous copy operations required during its execution. Finally, Figure 2 (c) shows the desirable way that we would like the system to operate. Writes should be allowed to proceed asynchronously to persistent memory without stalling to make synchronous copies or creating a storm of deferred writes at commit time. The stall at the end is to force any remaining writes out to persistent memory.

Figure 2: (a) Write Storm (b) Copy-On-Write (c) Ideal

3.1 WrAP Architecture

We introduce the notion of a **wrap** as an abstract conduit through which a thread funnels its writes to persistent memory. A wrap ensures the atomicity and durability of the transaction that it shepherds. However, the wrap is **not** directly involved in communicating the values of persistent variables between writers and readers. Reads and writes proceed independently through the system cache hierarchy, with minimal interference from wrap operations. The interaction between the cache hierarchy and the wrap occurs only at the back-end, at the SCM memory controller. All of these interactions will occur in background or asynchronous mode that are off the critical execution path. This contrasts with most proposals [6, 13, 21, 22, 15] in which cache and logging operations are coordinated at the processor or cache.

A wrap has several different functions:

- Acts as a lightweight **firewall** that prevents arbitrary writes to persistent memory. Changes to protected areas of persistent memory are only possible through a wrap operation. Like a file system that protects a storage device from arbitrary updates, all changes to persistent memory are orchestrated by a wrap.
- Provides an **ordered log** of all updates to persistent memory made by transactions, permitting rollback or recovery in case of process or system failures.
- Provides a **non-intrusive interface** for interaction between the cache system and persistent memory while permitting relatively independent operations.

Figure 3 shows a high-level view of the WrAP architecture, that is responsible for providing various wrap services. It is made up of several components: a **victim persistence cache** (VPC); a Log area of SCM that is used to keep a **log of update** operations; and an asynchronous channel used to propagate log records to persistent memory.

Figure 3: WrAP Architecture

A thread will do a **wrapped write** when it wants to update persistent storage in an atomic manner¹. At the start of an atomic region, the thread **opens** a wrap and obtains a **token**, which is used to uniquely identify this wrap. Writes within the atomic region result in *two* actions: a wrap record is created to log this update (similar to a redo log record) and write it to a reserved area in the Log structure allocated by the wrap. Simultaneously, a normal *store* instruction to the persistent memory address is issued. At the end of the atomic region the thread **closes** the wrap.

A log record is a **key** and **value** pair, consisting of the memory address that the transaction is updating and the value being written. Log records are write-once records used only for logging purposes. Hence, they are not constrained by memory consistency requirements and do not benefit by caching. In addition, while the underlying writes may be to scattered persistent memory addresses, the log records of an atomic region will all be stored contiguously in a bucket associated with this wrap. This makes them ideal candidates for using the non-cached write-combining modes present in

¹Programmer or compiler support is needed to identify writes that must be wrapped.

many modern processors (referred to as non-temporal writes [11]). This mode bypasses the cache on stores and uses a write buffer to combine writes in a cache line before flushing the buffer to memory, greatly speeding up sequential writes. When the transaction commits, a single persistent fence operation is needed to make sure that any remaining log records have been written out to the corresponding bucket.

The normal store instruction that is issued concurrently gets written to the cache as usual. This cached item is used to communicate the value of the variable to any reads (loads) made to it. The read may be from the same thread that did the write or from a different thread that is allowed to do so by the isolation policy in effect. Note that if the value had not been written to the cache, then these reads would involve a slow, software-arbitrated lookup of the Log in order to satisfy the read. While such arbitrated approaches are common when using disk-based logging, they are not compatible with the load and store characteristics of SCM accesses. Writing only to the cache is clearly insufficient and will lead to the situation described in Figure 2(a). The updated cache lines must be prevented from updating the persistent memory locations until the transaction commits, and all the updated values must then be flushed to persistent memory before the transaction ends.

As noted above, persistent memory locations *should not* be updated until the transaction commits. The log write does not update the actual memory locations referenced by the thread. However, there is no guarantee that as part of its normal activity, the cache hierarchy will not evict such a cached write to persistent memory before the transaction commits. One approach is to simply mark these updates as clean, so the cache lines are never written back to memory. If the cache reclaims one of these cache lines for a different memory block, it simply overwrites its contents. In this case, future reads to the variable will need to be trapped and the latest value returned from its saved value in the Log. We do not consider this approach further in this paper, but plan to investigate it more completely in our continuing work.

To address the problem of premature cache evictions, the wrap controller implements and manages a **victim persistence cache** (VPC) to hold persistent memory entries that are evicted from the last-level cache. The VPC will serve as the backing store for these evicted variables until the SCM controller writes them to their persistent memory locations. Unlike the usual spillage of dirty cache lines which results in updating main memory, these cached values will **not** be written to their persistent memory locations. Instead, they will be saved in the VPC which acts as a logical extension of the cache hierarchy. The VPC may be implemented in volatile DRAM memory, since its entries need not be persistent. The maximum size of the VPC depends on the number of *live* persistent variables that overflow the last-level cache. The number of these variables is bounded by the number of distinct variables in currently open wraps *i.e.* variables in open transactions that have not yet committed. To keep the VPC from growing too large, items that have been safely committed to persistent memory should be deleted from the VPC. Once deleted, the next read of that variable will result in a cache read miss, and will be serviced in the normal way by reading that variable from its persistent memory location. Deletions of entries from the VPC can be done as part of the background operation by the wrap controller, when it copies values from the log records to their actual memory locations. Different designs of the VPC are possible: these range from pure hardware-controlled solutions to software cache implementations. The latter would be triggered by a cache miss exception that would return the value from the VPC rather than from persistent memory.

The use of write coalescing in a DRAM buffer in front of PCM

to reduce the number of writes and improve reliability was used in [20]. In contrast, the proposed VPC stops writes from percolating to the backing SCM, and is used to service memory requests made by the processor.

3.2 Background Operations

As mentioned earlier, the actual persistent memory locations referenced by a write operation (called *home locations*) are not updated immediately. A copy is made in the cache in order to facilitate normal program functioning, and a log record carries the new value to the log bucket associated with the wrap.

A possible organization of the Log is shown in Figure 4. When a wrap is opened, it is allocated a bucket in the Log area. A bucket implements a Key-Value store to hold the log records being written in that atomic region. The figure shows four buckets A, B, C and D. Of these, C and D are buckets belonging to wraps that are currently open. Buckets A and B belong to wraps that have already closed. No new records will be added to a closed wrap. When a wrap closes, it is added to a Log, which is a circular First-In-First-Out queue of closed wraps. Each entry in the Log points to the bucket of a closed wrap. When a wrap closes, its bucket is atomically added to the tail of the Log queue by appending a pointer and incrementing the tail indicator. Methods to implement a robust Log in the presence of failures are presented in [7, 22], and we can easily adapt those ideas for our log structure as well. As discussed below, the entries in the Log are periodically processed and deleted after the associated updates are made persistent. Note that a transaction is allowed to complete only after its bucket has been added to the Log.

Figure 4: Log structures maintained by WRAP

The update to the home locations will be made independently by a **COPY** module that is part of the wrap controller. The module operates as a background task that is periodically invoked to trim the log. It operates on the Log entries in order from the head towards the tail. This module will perform an update operation, one wrap at a time, in the order that they closed. Its task is to update each persistent memory variable referenced in the wrap’s bucket, by copying the new value to the address noted in the log record.

The frequency of invocation of the COPY operation is constrained by the space available in the VPC. If too many items belonging to closed transactions remain in the VPC it may overflow. In our design, these items will be deleted when the COPY module updates the corresponding persistent memory location from the log. We note that overflowing the VPC should be a very rare event with proper sizing. In any case, it is not a fatal issue since the system can always switch to a degraded mode where items not found in

the cache hierarchy are first searched for among the buffered log records before returning the value stored in the SCM. Clearly, this has a significant performance impact, but is provided only as a safe exception path should the rare event actually materialize.

A couple of issues regarding the COPY operation should be noted. First, it is possible that a variable that is being copied from a log record to its home location, has since been updated by another transaction. In fact this more recent transaction may still be live or may itself have completed. In either case the VPC entry, which contains the most recent value for that variable, should not be blindly deleted. A check to make sure that this is the last transaction that wrote to the variable is necessary, and the item should be deleted only if the copying is being done by the most recent transaction. An alternate strategy is to check that the value of the item being copied as part of the COPY operation is the same as that in the VPC. In this case, the item in the VPC can be safely deleted, even if it is not the last transaction that wrote it. This can happen if two transactions wrote the same value to the variable. In this case, the premature deletion of the entry in VPC is unnecessary, but can cause no harm.

The second issue is that of failure occurring during a COPY operation. A system failure that occurs before all the locations have been updated will leave the log record and its bucket unchanged, so that on reboot the recovery module can redo all the updates in the bucket without problem.

3.3 Restart and Recovery

Finally, we discuss the requirements for recovery from system failure, using the **RESTORE** module. On system reboot **RESTORE** is invoked. Its task is to make sure that all pending updates found in the Log are applied to their actual home locations before the restart operation completes. This is a necessary step, since it is possible that the system crash occurred while a COPY operation was in progress that could have left the variables of that wrap in an inconsistent state. Since the pending updates recorded in the Log entries collectively leave the system in a consistent state, this is also sufficient to ensure the system is restarted from a correct initial state. The **RESTORE** module also flushes all entries in the VPC and reinitializes the structure. In fact, since the VPC can be implemented in volatile DRAM its contents may have been lost in the system crash anyway.

Note that partially written buckets that were not attached to the Log at the time of system crash can be safely discarded, since their transactions are treated as not having completed. Of course, none of the variables that these transactions wrote have had their home locations updated either. Finally, employing a robust, yet lightweight, implementation of the Log structure (using any number of torn writes detection techniques mentioned in the literature) ensures that a failure that occurs during the update of the Log while a bucket is being added, can be detected by the **RESTORE** module.

4. EVALUATION

In this section we present a performance evaluation of the WrAP approach to persistence. We present the results of a micro benchmark that performs variable numbers of transactional operations (Insert, Find, Delete) on several data structures implemented in persistent memory. We compare the performance in three different scenarios. In the first scenario, the benchmark is applied in non-transactional mode using PCM as the backing store; in the second, transaction semantics for each operation are enforced using a traditional implementation based on an Undo log. The third implementation uses the WrAP approach. As expected, adding transactional semantics slows down program execution significantly. However, using WrAP there is a significant performance improvement over

a traditional transactional implementation. In Section 4.1 we describe the experimental methodology used; empirical results are presented in Section 4.2.

4.1 Performance Model

A benchmark program was created which can perform a variable number of random Insert / Find / Delete operations on a configurable data structure while tagging a group of operations as a WrAP transaction. The data structure can be initialized with any number of elements and the number of operations in a transaction and the number of transactions can be varied. The procedure to generate the traces is outlined in Figure 5. A trace of the memory accesses is obtained using Pin tools [14] to instrument the benchmark program. For the evaluation we need to distinguish accesses to persistent memory from accesses to volatile DRAM. Furthermore, there needs to be a mechanism to recognize transaction boundaries by examining the trace.

We use the C++ Standard Template Library (STL) for the implementations of several data structures that are allocated in persistent memory. To track these persistent addresses, we instrument the STL memory allocator to obtain a trace of the ranges of virtual memory locations it allocates to the calling program. The memory trace generated by the Pin tool is then analyzed offline, and addresses of the allocator and benchmark are reconciled; the persistent memory locations so identified are flagged for the simulation. To identify transaction boundaries, the benchmark program writes to a specific known memory location immediately before beginning a transaction and a different known address immediately before the transaction ends. In addition, a program start location is saved to ignore initializations such as shared library loading captured by Pin. These memory locations are saved to combine with the virtual memory accesses and CPU delays generated by the Pin analysis along with the persistent memory address ranges from the STL memory allocator. The final trace file contains a sequence of memory access records with several fields: the number of instructions since the previous memory access, memory address, and a set of identifiers indicating whether the address is persistent or volatile, the operation is a read or a write, and whether it is the start or end of a transaction. In this evaluation we only consider non-overlapping transactions.

Figure 5: Trace Generation for STL Benchmarks

The memory simulation is done using a process based simulator Yacsim [12]. The system model consists of a single processor that executes the benchmark program. A single-level, direct-mapped processor cache is modeled, and all reads and writes (write allocate policy) are fielded by the cache. Reads and writes that do not occur within a transaction are handled as normal memory reads and writes, and accessed from the cache. A cache miss takes a variable amount of time depending on whether the access is to a volatile DRAM or persistent memory location. Evictions from the cache are either written to the victim cache or are written to their home location (which may be either in DRAM or persistent memory), depending on whether the access belongs to a transaction or

not. Access within a transaction (wrapped reads and writes) are handled by the WrAP protocol; in addition to accessing the cache, a wrapped write operation creates a log record that is pushed to an asynchronous queue to be flushed to the persistent memory log bucket. Cache misses for wrapped reads are served by the victim cache. System parameters (from [13]) normalized to a cache hit time are: DRAM accesses 100 cycles, and persistent memory read and write accesses are $T_r = 440$ and $T_w = 1200$ cycles respectively.

The persistent memory controller arbitrates between flushes of transactional log records for transactional writes, and servicing transactional reads and non-transactional writes. It gives priority to non-log accesses. The current implementation does not take advantage of the sequential nature of the log writes, whereby an entire cache line could be flushed in a single write operation. This would improve the performance of the WrAP scheme; the traditional implementation however does not naturally take advantage of blocked operations, since the flushing of the Undo log records need to be synchronously made persistent before the transaction can proceed.

4.2 Experimental Results

The first study of our simulation model was to verify predictions of the expected behavior of the WrAP architecture compared to an Undo log implementation. A persistent write operation in a cold cache using an Undo log approach incurs the following delays: a read followed by two writes which must complete in persistent memory to ensure atomicity and consistency. The WrAP approach requires the same read delay, but can place the store in a write back buffer that can drain asynchronously. With a store buffer size of one, the average time to perform a persistent write with T_c CPU time between writes, is given by: $T_U = T_r + T_w + \max(T_c, T_w)$ for an Undo log implementation and $T_W = T_r + \max(T_c, T_w)$ for a WrAP implementation.

To validate the simulator, we produced a synthetic trace that performs ten WrAP store operations on sequential cache lines. The traces were then run through the simulation for varying CPU delays from 0 to 10,000 cycles. Figure 6(a) shows the simulation time versus the CPU delay for three scenarios assuming a cold cache. The base case is for an application that simply ignores all transactional semantics and performs writes directly to the cache. This is compared to both WrAP and Undo log implementations that guarantee failure atomicity. Since the non-transactional approach does no logging (neither an undo or redo log) nor flushing to memory, it has the fastest simulation time. However, as the CPU delay (the time between writes in a transaction) increases, we see that the WrAP architecture performs as well as a system that provides no persistent memory consistency.

In a cold cache implementation the cache is empty at the start. In the hot cache implementation, the trace preloads the cache by accessing all elements that will be future WrAP stores in the simulation. The speedup of WrAP is calculated as T_U/T_W . For T_c between 0 and T_w , the speedup is $1 + T_w/(T_r + T_w)$. Using the values for T_r (400) and T_w (1200), the maximum speedup for a cold cache is 1.75. For a hot cache, where all the WrAP data is already in the cache, reads are a cache hit for both WrAP and the Undo log implementations. However, the latter performs two stores to memory that must complete before the transaction can continue. The WrAP architecture allows for stores to lazily write to the log in the background, overlapping the reads and other processing. This can produce a speedup of slightly greater than 2.0. Figure 6(b) shows the speedup obtained for both cold and hot caches. A cold cache with no hits and little delay between consecutive write operations within a WrAP will have a speedup close to 1.75. As

(a)

(b)

(c)

Figure 6: Comparison of WrAP and Undo Log for Simple Benchmark (a) Execution Time vs CPU delay (b) Speedup vs CPU Delay (c) Speedup vs Cache Size

the time between persistent stores within a WrAP increases, the speedup drops off as the persistent writes in an Undo log approach have more opportunity to overlap operations. Finally, the computation becomes heavily processor bound and both implementations require the same time.

Figure 7: Comparison of execution time for WrAP and Undo Log for STL-Based Benchmarks

Continuing with the simple trace, we verified that larger caches provide for an increasing speedup of WrAP compared to an Undo log implementation. Figure 6(c) shows speedup increasing for increasing cache size. For large cache size, the expected 1.75 speedup is reached. In addition, it is important to note that larger transactions will incur more cache misses, which slow the speedup.

After verification of the expected behavior using the simple sequential benchmark, we generated numerous traces using the method described in Section 4.1. To generate the traces, for each of the four benchmark data structures Set, MultiSet, B Tree, and B+ Tree, we varied the transaction size from one to ten. Each transaction consists of an *insert*, *find*, and *delete* of randomly generated data values into the data structure. The structure is initialized to contain 100 random elements, and the number remains constant through the trace. The traces were simulated for various CPU delays and cache sizes to generate the performance results.

Figure 8: Comparison of WrAP and Undo Log for STL-Based Benchmark: Execution time vs Transaction size

Figure 7 shows the time required by different methods on the benchmark memory traces. We see that WrAP performs nearly as well as a method that does not guarantee transactional atomicity and significantly better than the use of an Undo log. Figure 8 compares the speedup of the WrAP architecture over an Undo log for

Figure 9: Comparison of WrAP and Undo Log for STL-Based Benchmark: Number of store operations vs Transaction size

each of the benchmarks as the size of the transaction is increased. The cache size is set to 8192 blocks and the CPU delay is one cycle between instructions. The speedup for different data structures varies between 1.25 and 1.85. The B Tree benchmark takes full advantage of the cache as the transaction set size increases, thereby pushing the speedup above that of a purely cold cache. Figure 9 shows the number of persistent store operations for each of the data structures as a function of the transaction size. As can be seen, a B Tree implementation has almost double the number of persistent store operations per WrAP transactions for the benchmark with transaction size equal to five.

Figure 10: Execution time for B-Tree vs Transaction size

Figure 10 shows B Tree simulation time for various benchmark transaction set sizes for each of the WrAP, Undo log, and non-transactional approaches. The simulation was performed with a cache size of 8192 blocks and a CPU delay of one cycle between instructions. Even on small set sizes the WrAP method performs almost as well as a non-transactional persistent memory system.

Figure 11 also shows how each benchmark performs with increasing cache size. In this set of simulations the transaction size is set to five and each of the benchmarks and models are simulated and compared using the appropriate memory trace. With an increasing set size, the increased number of persistent stores contained within a WrAP can impact the speedup. As shown in the models, more transactions, more cache hits, and less time between persistent stores within a WrAP all increase the speedup.

Finally, for the benchmark memory traces we compare the WrAP Log Buffer and Victim Cache sizes. We simulated various cache

Figure 11: Benchmark Performance versus Cache size

sizes from 128 to 8192 blocks for each benchmark with five Insert, Find, and Delete operations per WrAP transaction and a CPU delay of one cycle between each CPU instruction, where consecutive memory operations have numerous instructions of one cycle each between them.

Figure 12(a) shows the maximum WrAP Log Buffer size for each of the benchmarks. The maximum log buffer size occurs when the cache size is the largest in the group of simulations, as that is the point where the operations are performing the fastest with no cache misses and the slow, persistent memory might not keep up with a fast cache. The buffer only grows to a size of 50 in the maximum case for a B Tree implementation that is conducting hundreds of persistent stores per transaction. The victim cache size was also examined in Figure 12(b). A large victim cache is seen when the memory cache is smaller, as more data is evicted from the cache into the victim cache. In the simulations, the WrAP architecture flushes all wrap log records and all entries in the victim cache to persistent memory before closing a WrAP transaction. The victim cache flush is a pessimistic assumption, since these can occur in the background without hurting correctness.

5. CONCLUSION

In this paper we presented an approach to using SCM in a memory-bus-based load and store architecture. In this situation, uncontrolled writes to persistent memory locations can leave the system in an inconsistent state if there is a failure. Traditional transaction systems have dealt with the problem in the context of block-based I/O accesses to disks and more recently to SSD storage. However, it is only now that these issues have begun to be addressed for cache-line-granularity accesses to fast persistent memory systems. Conventional approaches are unsuitable because they interpose a software layer between the transaction and the memory system, thereby failing to fully exploit the advantages of fast, byte-addressable persistence.

We presented a new approach to this problem and discussed the issues arising in its implementation. Our idea is to propagate the writes within an atomic sections along two paths: the fast path through the cache hierarchy and a slow path used for background updating of persistent memory, and recovery from system crashes. We propose a novel last-level persistent victim cache to prevent premature spilling of cache contents to persistent memory locations, while simultaneously avoiding costly look-up operations along the critical path. Preliminary evaluation using synthetic micro benchmarks of transactional operations on STL data structures (sets, multi sets and B Trees) shows significant performance gains

(a)

(b)

Figure 12: STL-Based Benchmarks (a) WrAP Log Size (b) VPC Size

over traditional techniques for persistence based on the Undo log. We have shown in many instances that the WrAP architecture can perform as fast as a system that provides no consistency. The gains are expected to increase with transactions that have greater variable reuse, and with concurrent transactions that access shared data using weak consistency semantics, an increasingly common mode of access in many current database applications. In continuing work we are evaluating the performance of the approach for different design alternatives, hardware and software tradeoffs, structure sizes, and different application domains.

6. REFERENCES

- [1] Ibm solidDB. In <http://www-01.ibm.com/software/data/soliddb/soliddb/>.
- [2] Neo4J:. In <http://neo4j.com>.
- [3] VoltDB:. In <http://voltdb.com>.
- [4] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 385–395. IEEE Computer Society, 2010.
- [5] J. Coburn, A. Caulfield, A. Akel, L. Frupp, R. Gupta, R. Jhala, and S. Swanson. Nv-heaps: Making persistent objects fast and safe with next generation, non-volatile memories. In *Proceedings of 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 105–118. ACM Press, 2011.
- [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 22nd ACM Symposium on Operating Systems Principles*. ACM Press, 2009.
- [7] R. Fang, H. Hsiao, B. He, C. Mohan, and Y. Wang. High performance database logging using storage class memory. In *Proceedings of 27th International Conference on Data Engineering*, pages 1221–1231. ACM Press, 2011.
- [8] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database: data management for modern business applications. *SIGMOD Rec.*, 40(4):45–51, Jan. 2012.
- [9] B. Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5–, Aug. 2004.
- [10] R. Freitas and W. Wilcke. Storage class memory, the next storage system technology. *IBM Journal of Research and Development*, 52(4/5), 2008.
- [11] Intel. Intel 64 and IA-32 architectures software developer manual. <http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>.
- [12] J. R. Jump. Yacsim reference manual. <http://www.owl.net.rice.edu/elec428/yacsim/yacsim.man.ps>.
- [13] B. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting PCM as a scalable DRAM alternative. In *Proceedings of 36th International Symposium on Computer Architecture*, pages 4–13. ACM Press, 2009.
- [14] C.-K. Luk and et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM PLDI*, 2005.
- [15] J. Moraru, D. Andersen, M. Kmainisky, N. Binkert, N. Tolia, R. Munz, and P. Ranganathan. Persistent, protected and cached: Building blocks for main memory data stores. In *CMU Parallel Data Lab Technical Report, CMU-PDL-11-114*, Dec. 2011.
- [16] D. Narayanan and O. Hodson. Whole-system persistence. In *Proceedings of 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 401–410. ACM Press, 2012.
- [17] J. Ousterhout. The case for RAMCloud. *Commun. ACM*, 54(7):121–130, July 2011.
- [18] M. K. Qureshi, M. Franceschini, and L. Lastras. Improving read performance of phase change memories via write cancellation and write pausing. In *Proceedings of 16th International Symposium on High Performance Computer Architecture*, pages 1–11. IEEE Press, 2010.
- [19] M. K. Qureshi, J. Kordis, M. Franceschini, M. Srinivasan, V. Lastras, and B. Abali. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *Proceedings of 42nd International Symposium on Microarchitecture*. ACM Press, 2009.
- [20] M. K. Qureshi, V. Srinivasa, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of 36th International Symposium on Computer Architecture*, pages 24–33. ACM Press, 2009.
- [21] S. Venkatraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for

- non-volatile byte addressable memory. In *Proceedings of 9th Usenix Conference on File and Storage Technologies*, pages 61–76. ACM Press, 2011.
- [22] H. Volos, A. J. Tack, and M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 91–104. ACM Press, 2011.
- [23] P. Zhao, B. Zhao, J. Yang, and Y. Zhang. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of 36th International Symposium on Computer Architecture*, pages 14–23. ACM Press, 2009.