Atomic Persistence for SCM with a Non-intrusive Backend Controller[*]

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

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

Peter Varman
Rice University
Houston, Texas
pjv@rice.edu

ABSTRACT

Non-volatile byte-addressable memory has the potential to revolutionize system architecture by providing instruction-grained direct access to vast amounts of persistent data. We describe a non-intrusive memory controller that uses back-end operations for achieving lightweight failure atomicity. By moving synchronous persistent memory operations to the background, the performance overheads are minimized.

Our solution avoids costly software intervention by decoupling isolation and concurrency-driven atomicity from failure atomicity and durability, and does not require changes to the front-end cache hierarchy. Two implementation alternatives – one using a hardware structure, and the other extending the memory controller with a firmware managed volatile space – are described. Our results show the performance is significantly better than traditional approaches.

1. INTRODUCTION

This paper examines the use of byte-addressable persistent memory (also called Storage Class Memory [1] or SCM) as a replacement for traditional non-volatile storage (hard disks or SSDs) in high performance data processing systems. Real-time analytics and transaction processing using in-memory (DRAM) databases is gaining popularity [2]. Fast DRAM access enables high throughputs and low transaction latency, while the fine-grained cache-line accessibility of DRAM promotes the use of sophisticated parallel algorithms and data structures together with fine-grained work division and thread coordination. However, the volatile nature of DRAM makes these systems vulnerable to system crashes. Adding persistence incurs run-time overheads for application-specific checkpointing to non-volatile storage, as well as long recovery times following a crash or scheduled maintenance to rebuild the necessary in-memory structures.

Emerging high-capacity persistent memory [3] provides an opportunity to have the best of both worlds: persistence along with direct access to memory at the granularity of bytes, words or cache lines using a CPU's load-store instructions. Ideally, a program simply reads or writes locations in SCM just as it would in volatile memory, with the assurance that its SCM writes can be made durable simply by flushing data from CPU caches. An in-memory application can benefit from using SCM in this way since its fine-grained accessibility facilitates popular applications involving data structure operations like graph traversals, pointer chasing, searching, and indexing, while its durability removes the need to cover memory updates with backing store operations.

SCM makes it possible to eliminate having to manage multiple representations of data: one for CPU operations, and another for bulk transfer to a high-capacity storage tier. As a result, algorithms and data structures can be designed specifically for fine-grained random-access memory without worrying about the need for packing and unpacking these structures for block devices. With direct processor access to non-volatile SCM, there is no need to add complex disk-based checkpointing and logging as separate mechanisms, or to endure time-consuming rebuilding of memory structures from disk-formatted data following a system reboot. Additionally, the large SCM size reduces the frequency of memory preemptions allowing the use of simpler non-blocking synchronization in managing memory.

Since SCM is nonvolatile it raises the issues of consistency faced by all storage systems. A machine restart following an inopportune power failure can leave data in persistent memory in an inconsistent state. However, the ground rules for solving the consistency challenges change significantly when using persistent memory. First, the complex intervention techniques that databases [4] and operating systems [5] use to protect integrity of disk-based data are not suitable for the fine-grained, frequent, and fast flowing updates to data in SCM. Second, where as in a traditional disk-based system evictions of modified data to disks are performed and thus controlled by software, SCM interacts directly with the processor cache hierarchy and is therefore exposed to uncontrolled cache evictions from CPUs. To address these two problems, this paper describes a light-weight solution for ensuring that groups of logically related updates to physically scattered locations in SCM are performed atomically, even in the presence of machine failures. The solution consists of a novel backend controller and a software library that non-intrusively provide failure atomicity without costly software intervention or changes to the well-established front-end processor cache hierarchy.

The rest of the paper is organized as follows. In Section 2

[*]Supported by NSF Grant CCF 1439075 and Intel SSG.

we describe the atomic update problem for SCM. Our solution using the backend controller supported by a software library are described in Section 3. Two implementations of the controller using hardware-friendly structures and using embedded firmware are described in Section 4. Evaluation methodology and results are presented in Section 5.

2. OVERVIEW

A sequence of updates flowing from CPU caches to non-volatile memory locations can be interrupted at any time due to hardware or software failures. This gives rise to two complementary ordering problems that need to be solved without excluding the benefits of the processor cache hierarchy.

(1) **Delayed update problem** : Default memory store semantics supported by processors make no guarantees of when the value written to the target location of a store instruction will actually be reflected in the backend device (typically DRAM). This is not a problem if these are regular DRAM variables for which value propagation is the goal, since this can be effected through the cache hierarchy without actually updating memory. For persistent memory variables, cache flush instructions guarantee eventual update at persistent memory home locations provided there is no failure, and memory fences (such as SFENCE) ensure that stores are visible to other CPUs before the writers continue. However neither mechanism can ensure that a software thread is able to wait for completion of a write to durable medium before issuing an instruction that depends on such a guarantee. To address this gap, processor manufacturers are providing *persistent fence* instructions (*e.g.* Intel's recent PCOMMIT instruction) whose retirement guarantees that prior stores are successfully committed in a power-safe domain.

(2) **Early update problem**: Since caches evict modified cache lines autonomously, software must explicitly flush modifications before undertaking dependent operations, to preserve the desired order of updates to persistent memory.

The two ordering problems with respect to a single store may be addressed by making sensitive stores synchronous. A store can be made synchronous (deterministic) with respect to later instructions by writing to its cacheline, and then following it with flush and persistent-fence instructions. However, even if a programmer were to make individual stores synchronous in this manner, it is not easy to guarantee atomicity of write sequences: that would require resuming execution of the transaction precisely from the point of failure necessitating all volatile state to have been check pointed as well [6]. To achieve resilient operation, a programmer must consider employing additional metadata (e.g., a journal) also in persistent memory, to track which sequences completed and which did not, and use that metadata for effecting recovery. Conventional storage systems today don‘t face a similarly severe coordination problem since blocks on a durable medium are accessed indirectly through system software, which effectively coordinates the sequencing of updates and manages recovery from torn updates.

Algorithm 1 shows an example (on the left) in which three variables need to be either all updated or none updated in persistent memory. (Comments on the right may be ignored until section 3). The programmer wants to make such an update sequence atomic across a potential machine restart, while allowing the values of the three variables to be freely communicated via the CPU cache hierarchy for performance. Without additional support, an example of the early update problem arises if algorithm 1 modifies *x*, and *y* in CPU caches, the new values of *x* and *y* are evicted into persistent memory from the caches, but an untimely machine crash prevents the algorithm from completing and writing a new value of *z* into persistent memory. Conversely the delayed update problem arises if algorithm 1 completes, but a machine crash precludes the writing back of the new values of any (or all) of the three variables into persistent memory. Imagine that the writes in algorithm 1 were flushed from the CPU cache and persistent-fenced with respect to writes that followed; the programmer would need to keep track of the progress of the execution in a persistent journal to correct for a machine crash that interrupts the above sequence – in analogy with a database or filesystem based approach for achieving transactional updates to data and/or system state.

Algorithm 1: Programmer annotated failure atomic region

```
// x, y and z are persistent variables
atomic_begin   | id = OpenWrap();
  x = 1;       | wrapStore(id, &x, 1);
  ........
  y = 2;       | wrapStore(id, &y, 2);
  ........
  z = x;       | wrapStore(id, &z, x);
  ........
atomic_end     | CloseWrap(id);
```

In the next section we describe a novel hardware solution which liberates programmers from having to manage failure atomicity in persistent memory.

3. OUR APPROACH

Figure 1 shows the logical organization of the hardware. A backend controller intercepts cache misses and evictions to persistent memory as shown in the left of the figure. Evicted cache lines are held by the controller in a (volatile) *victim cache* and prevented from directly updating SCM. Instead, the home locations of persistent variables are updated *asynchronously* from *log records* streamed to a persistent log area using compact, write-combined streaming stores. Victim cache entries are deleted only after their updates have been reflected in persistent memory by log retirement. In the meanwhile, processor misses for these cache lines are served from the victim cache. The log itself is continuously pruned as its records are copied to persistent memory. We describe a simple protocol which guarantees that the combination of victim cache and SCM never returns stale values, and that log records do not have to be searched to locate the most recent update. On a power failure it is safe to lose the contents of the victim cache. The log records will be used on restart to bring persistent memory to a consistent and up-to-date state.

The hardware controller described above is complemented by a software library that bridges the programmer and the controller interfaces. Algorithm 1 shows a pair of library calls **OpenWrap** and **CloseWrap**, which are created from the programmer-annotated atomic_begin and atomic_end directives. The region of code that is bracketed by OpenWrap and CloseWrap is a failure atomic region with all-or-nothing store semantics. We will refer to such a region as a *wrap* and the backend controller of Figure 1 as the *SCM controller*.

When a wrap is *opened*, OpenWrap allocates a *log bucket* in a non-volatile memory area known to the SCM controller. This is a sequential byte array that implements a redo log for the wrap. The redo log is a sequence of log records, one for each persistent memory store operation done within the wrap. A log record holds the address being updated and the updated value. Stores to persistent memory variables are "wrapped" by a call to the **wrapStore** library function as indicated on the right in Algorithm 1. The wrapStore operation (a) writes to the specified persistent memory address (such as the addresses of x, y, or z) using a normal write-back instruction (e.g., an x86 MOV) and (b) appends a log record containing the pair, [address, new-value], to the wrap's log bucket. The processor does not need to wait for the log records to be written to persistent memory at this time. The new values written by the normal write instructions are communicated to later loads (such as the assignment z = x in Algorithm 1) via the cache hierarchy just as non-persistent DRAM variables. However, unlike regular DRAM variables, the flow back of the updates to their persistent memory home locations is through the logged records and *not by write backs* resulting from cache evictions. Instead, write backs of these variables are intercepted and stored in the victim cache until the updates are successfully retired to persistent memory from the log records.

The log records are written to the log bucket using write-combining streaming stores (e.g., using MOVNT in x86 architecture) that bypass the cache. At the point of CloseWrap, any pending log record writes together with an end-of-log marker are flushed to the log bucket using a persistent fence (like the x86 instruction sequence SFENCE, PCOMMIT). A sentinel bit in each log bucket record ensures that any torn writes in the log bucket sequence can be detected, without additional fences to correctly order the write of the end-of-log marker. This log flush is the *only* one synchronous operation required per wrap in our protocol. Since the log bucket is a sequential array, write-combining is very effective in reducing the time required to record the updates, in contrast to directly updating the scattered persistent memory addresses of the variables being updated.

Transactional stores of single variables whose atomicity is guaranteed by the hardware, are optimized by fusing the library operations OpenWrap, WrapStore, and CloseWrap into an anonymous wrap that writes a singleton log record. Further, any streaming (non-temporal) writes to SCM are considered as controller bypassing writes.

Different processors may safely update different fields in the same cacheline; SCM updates are only made from logged field values, while the victim cache behaves as an extension of the normal cache hierarchy. This avoids the complexities that arise when, for instance, two variables that share the same cache line are updated by separate transactions, one of which completes but the other aborts.

Wraps may nest if the programmer chooses, and the default semantic is to flatten the individual wraps contained within the outermost wrap, so that the outermost OpenWrap-CloseWrap pair defines the single failure-atomic sequence.

Since the wrap library API calls are done independently of the cache operations, a protocol is necessary to coordinate the retirement of log records and the operations of the victim cache, while ensuring recoverability from unexpected failures. In the next section we describe such a solution along with two different implementations for the controller.

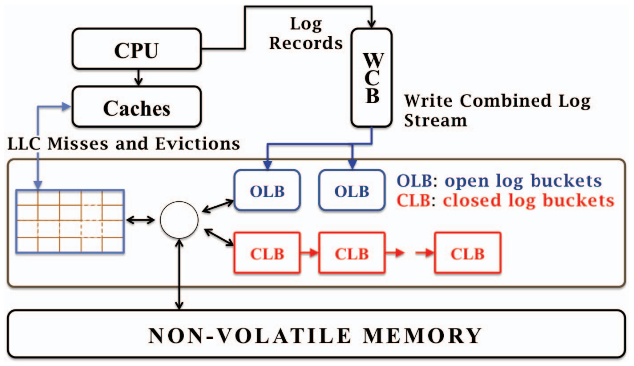

Figure 1: **SCM Controller: Intercepts cache misses and evictions and log records. Log records must be in SCM while victim cache can be volatile.**

3.1 SCM Controller Algorithm

The SCM controller ensures that the victim cache appears as a transparent additional caching level; behind-the scenes it enforces safety in the retirement of updates to SCM and continuously prunes and retires the log. We first describe the high-level operation of the SCM controller in Algorithm 2. We follow this with two implementations (controller hardware, or controller firmware based) in Section 4.

The SCM controller handles two operations originating from the processor cache: **CacheEvict** in response to a cache eviction and **CacheMiss** in response to a cache miss. It also supports the operations **HandleOpenWrap** and **HandleCloseWrap** invoked from the Wrap software OpenWrap and CloseWrap functions. The **RetireWrap** controller routine is invoked internally by the controller to update home locations of closed wrap buckets.

Each open wrap has a wrap id (a small recycled integer). The controller keeps track of the wraps that have opened but not yet retired in the set variable **openWraps**, which is updated in *HandleOpenWrap* and *RetireWrap* controller functions. When a wrap closes, its log bucket is appended to the tail of a FIFO queue of closed log buckets pending retirement. The log records in these buckets are retired asynchronously with backend writes of updated data values into their persistent memory home locations (see *RetireWrap*).

A block B that is evicted from the processor cache is placed in the victim cache and tagged with its *dependence set* DS_B. DS_B is the set of wraps that were opened before B was evicted that have not yet retired. DS_B is initialized with the value of *openWraps* at the time of its eviction; as wraps retire they are removed from DS_B. The dependence set is used to determine when it is safe to delete a block from the victim cache.

Block B is deleted from the victim cache when its dependence set D_B becomes empty. Suppose B was evicted at time t and was last written by wrap w. Then either $w \notin DS_B$ (w retired before t) or $w \in DS_B$. If $w \notin DS_B$, then all updates made to B (by w or earlier closing wraps) have already been written to SCM and B can be safely deleted from the victim cache. On the other hand, if $w \in DS_B$, then B may hold updates of a currently open wrap that have not yet been reflected in SCM. Since we do not know which case holds, we conservatively assume that $w \in DS_B$ and keep B in the victim cache until DS_B becomes empty.

Algorithm 2: Controller Algorithm

```
HandleOpenWrap (wrapId w)
    openWraps = openWraps ∪ {w};
HandleCloseWrap (wrapId w)
    Add bucket w to end of closedBuckets list;
CacheEvict (block address B, data d)
    $DS_B$ = openWraps;
    Add $(B, DS_B)$ to VictimCache;
CacheMiss (block address B)
    if (block B not in VictimCache)
        Forward memory request to SCM;
    else
        Return value of block B from VictimCache;
RetireWrap (wrapId w)
    for each address $a \in bucket(w)$
        Store $val(a)$ in SCM address $a$;
    for each block $B \in VictimCache$
        $DS_B = DS_B - \{w\}$;
        if ($DS_B == \Phi$)
            Delete block B from the VictimCache;
    openWraps = openWraps - {w};
```

We illustrate the algorithm using the example operation sequence of Table 1. The *HandleOpenWrap* calls at $t = 1$ and 3 cause wrap ids 1 and 3 to be added to *openWraps*. When A is evicted at $t = 2$, its dependence set DS_A is set to $\{1\}$, the current *openWraps*. Similarly, when B is evicted at $t = 4$, it is tagged with $DS_B = \{1,3\}$. A cache miss for A at $t = 5$ is serviced from the victim cache but not deleted from it. Since it is possible for A to be subsequently replaced silently in the processor cache, the controller retains A in the victim cache. When A is again evicted at $t = 6$ it is tagged with an updated dependence set DS_A=$\{1,3\}$. The previous version of A can be removed at this time. When wrap 3 closes, its log bucket is moved to the retire queue. At $t = 9$ log bucket 3 is retired and removed from *openWraps* and the dependence sets of both A and B, resulting in DS_A= DS_B= $\{1\}$. When log bucket 1 is retired at $t = 10$ both A and B are left with empty dependence sets, and thus both can be safely deleted from the victim cache.

This behavior of the SCM controller is summarized by the following invariant.

Property 1: if a persistent memory variable is found in the processor cache hierarchy then its value is that of its latest update. If it is not present in the cache hierarchy but is present in the victim cache then this is its latest value. If it is neither in the processor caches nor in the victim cache, then its persistent memory home location holds its latest value. Consequently, the controller does not have to search the log records at run time to find the value of a variable.

T	Operation	openWraps	Victim Cache
1	openWrap(1)	$\{1\}$	Φ
2	*CacheEvict*(A)	$\{1\}$	$\{A\}$
3	openWrap(3)	$\{1,3\}$	$\{A\}$
4	*CacheEvict*(B)	$\{1,3\}$	$\{A,B\}$
5	*CacheMiss*(A)	$\{1,3\}$	$\{A,B\}$
6	*CacheEvict*(A)	$\{1,3\}$	$\{A,B\}$
7	CloseWrap(3)	$\{1,3\}$	$\{A,B\}$
8	CloseWrap(1)	$\{1,3\}$	$\{A,B\}$
9	RetireLogBucket(3)	$\{1\}$	$\{A,B\}$
10	RetireLogBucket(1)	Φ	Φ

Table 1: Example Trace

3.2 Handling Non-transactional Stores

We now describe how to deal with mixed workloads consisting of both transactional writes and stores that are not part of any transaction. It is possible to simply treat the latter as singleton store transactions as discussed earlier, but this approach incurs unnecessary overheads for wrapping, logging and replay. Instead we alter the SCM controller protocol to write these variables that spill into the victim cache to SCM. For these variables the controller simply acts as a pass through with delay. The controller must infer which entries in the victim cache can be discarded and which need to be written back to SCM. We make a reasonable assumption that a cache line does not contain both transactional and ordinary variables. This allows the protocol to operate at the granularity of cache lines rather than individual words.

The modified protocol operates as follows. As each log record is written to SCM during log retirement, we look up the victim cache for the address being written; if the corresponding block is found in the victim cache it is marked as *clean*. The update of the dependence sets proceeds normally without any change. However when the dependence set of a block becomes empty, the block in the victim cache is simply marked as *free* but its contents are not deleted. When the controller needs to insert a block into the victim cache it looks for a free block as before. If the free block is also marked as clean the block can be overwritten by the incoming data; otherwise the current contents must be written back to SCM before the block can be overwritten. The updated block is then marked as *dirty*. In some situations (if an eviction occurs after the log retirement of the last update) some transactional data may be written twice to SCM, once from the log and once by writeback from the victim cache. However, the correctness invariant for transactional data still holds; further, in the absence of failure, all non-transactional data evicted into the victim cache will be eventually updated in SCM.

3.3 Isolation Modes and Persistent Order

In [7] a memory model was proposed that distinguished consistency orderings in volatile memory and persistence orderings in non-volatile memory. The former refers to the permissible orders for loads and stores during program execution, while persistence ordering refers to the order in which persistent memory locations are actually updated. The state of persistent memory following crash recovery reflects the persistent memory order. In our design, persistence ordering is defined by the order in which wraps are closed.

Algorithm 3: Allowed memory ordering depends on the isolation mode. Persistence ordering depends on the order in which wraps close.

```
// x and y are persistent variables initially all 0

T1                          T2
atomic_begin;               atomic_begin;
  x = 3;                      x = 5;
atomic_end;                   y = x;
                            atomic_end;
```

The permissible orderings of reads and writes in a multithreaded program are constrained by the transaction isolation mode (*e.g.* serializable, repeatable reads, read committed, read uncommitted) chosen for the application. Two transactions may serialize differently, depending on the chosen isolation mode. With the exception of read uncommitted, a transaction will hold all write locks for the duration of the transaction. Hence, program ordering seen by the threads and the persistence ordering enforced by the controller will be equivalent for these modes. For instance, in Algorithm 3 under the first three isolation modes the permitted outputs after both T1 and T2 have executed are $(x = 5, y = 5)$ or $(x = 3, y = 5)$. Since write locks are released only on transaction commit, persistence order will match execution order. In the case of uncommitted reads an additional possible output is $(x = 3, y = 3)$. In this situation execution ordering may differ from persistence ordering $(x = 5, y = 3)$ if T2 closes later than T1. The weak isolation mode trades off predictability for performance.

4. CONTROLLER IMPLEMENTATION

Algorithm 2 may be implemented as a hardware structure or as firmware extensions to a memory controller. We describe both alternatives below. Details can be found in [8].

Hardware Implementation: In the hardware implementation *openWraps* is maintained in a bit vector. Its size is a limit for the number of wraps simultaneously open at any time, and it is usually in the range of the number of CPUs. When a wrap opens, a bit in *openWraps* is set, and when it retires, the bit is cleared. The victim cache is implemented as a modest set-associative structure in which each cache block maintains, in addition to the standard tag and data fields, an additional field for the block's *dependence set* whose width is that of *openWraps*. When the log bucket for a wrap is retired, the wrap id is broadcast to the victim cache and each block clears the retired id bit from its dependence set. If the bit vector representing the dependence set of a victim cache block becomes 0, the block can be deleted and reassigned.

Ordinarily, background log retirements prevent the victim cache from becoming saturated. To decrease the chance of overflow, more space-efficient structures like a cuckoo hash table may be employed. Alternatively, we propose storing the entries overflowing the victim cache in a DRAM area managed by the controller firmware.

The above implementation of the SCM controller uses hardware-friendly components like associative search, broadcast, and highly parallel operations. An alternative approach is to use software-based search implemented by controller firmware using a DRAM victim cache as described next. The firmware can be used as an alternate implementation for the victim cache or may only serve as an overflow structure.

Firmware Implementation: A hash-based key-value store (KVS) implements the victim cache. The key is the persistent memory address of the cache block and the value is the block data. The dependence set of a block is not stored in the KV store but in a separate FIFO queue (DFIFO). On a *CacheEvict*, an entry is allocated at the tail of DFIFO and tagged with the current *openWraps*. The data is inserted into the KVS and a pointer to the KVS location is added to the new DFIFO entry. If there is an older version of the same block in the KVS we keep both versions in the firmware implementation. This enables quick deletion from the KVS when the dependence set for this block becomes null.

When a wrap retires it must be deleted from the dependence sets of all cached blocks. Without a broadcast, this would require a time-consuming scan of all the blocks every time. We avoid this by exploiting an inclusiveness property of dependence sets. Specifically, DFIFO becomes null in order from the head towards its tail. That is, if an entry in DFIFO has a null dependence set then so does every other entry that lies between it and the head. We combine this observation with a lazy update of the dependence sets to allow a constant time amortized deletion operation. We keep a set of retired wraps and apply them only to the head entry of DFIFO. We advance to the next entry in DFIFO only when the dependence set of the head entry becomes null and the corresponding entry in the victim cache is deleted [8].

A wrap id *w* can be recycled when no entry in DFIFO has *w* in its dependence set. This can be efficiently checked during the walk of DFIFO. To avoid unnecessary stalls due to the lazy release of wrap ids, it is is sufficient to alternate between two versions for each wrap id to match the behavior of an immediate release of the wrap id. Finally, a long running transaction can be handled by extending the KVS using system memory. The following claim summarizes the behavior of the KVS and DFIFO structure.

Property 2: The amortized time to delete a victim cache entry is $O(1)$. The time to insert or lookup a block in the victim cache is constant with high probability due to the hash implementation of KVS.

5. EVALUATION

We now present an evaluation of our approach. The section proceeds in three parts. First, in subsections 5.1 and 5.2, we describe how we extend and use memory simulation techniques from recent literature to work with SCM. Next in subsection 5.3, we use a number of memory and transaction intensive unit tests to evaluate the benefits of log-write streaming, reduced number of PCOMMITs, and background transfer of values to SCM home locations. We compare the WrAP approach with an undo-log approach, and with two unsafe update methods: ***cached*** and ***non-atomic*** for additional contrast. In sections 5.4 and 5.5, we turn to using two macro performance tests: the first of these employs the Graph 500 benchmarking harness, and the second exercises a B+ tree extension to the C++ standard template library (STL). For the various tests we do in sections 5.2-5.5, we vary the clock rate of SCM between 1/2 that of DDR3 and 1/8th of DDR3, and select a relatively pessimistic 1/4th, to evaluate the potential slowdown of this technique compared to the two unsafe methods. The evaluations reported in this section will show that the WrAP approach closely matches and even exceeds the performance of an unsafe method that commits transactions by streaming writes directly into the SCM (bypassing the caches) asynchronously. While the result may seem anomalous, it is explained by the ability of the WrAP method to benefit from sequential (cache combining) writes to the log buckets.

5.1 Methodology

To evaluate our design we adopted a simulation methodology similar to that used in Kiln [9] using the McSimA+ simulator [10]. This choice is prompted by the current absence of readily available hardware platforms having SCM and ISA support for persistent barrier instructions like PCOMMIT [11]. McSimA+ is a PIN [12] based simulator which decouples execution from simulation. McSimA+ models the out-of-order processor micro-architecture at the cycle level and includes a detailed initialization and timing configuration that has been finely tuned and tested to match the Intel Nehalem micro-architecture.

We made several extensions to the McSimA+ based setup:

1. Added PIN support for streaming (non-temporal) stores, PCOMMIT, and tick cycle rates. The McSimA+ simulation and PIN based execution are decoupled implementations; hence we added mechanisms for feeding various events, statistics, and time (tick) indications from one to the other. Due to large difference in the way that "time" advances in the two environments, to emulate aligning effects (such as PCOMMITs), an artificial cycle count is passed from PIN into the simulator to keep the latter from advancing further than the former's notion of time.

2. Supplemented it with new statistics gathering routines, such as numbers of Wrap open, close, write, read operations, numbers of PCOMMIT and sfence operations, numbers of SCM cache line writes and numbers of DRAM writes.

3. Extended the simulator to support WrAP primitives: OpenWrap, CloseWrap, and WrapStore; cache misses from the simulated environment are handled by changes inside the simulator to implement the SCM controller logic and an additional path in the simulator for streaming stores.

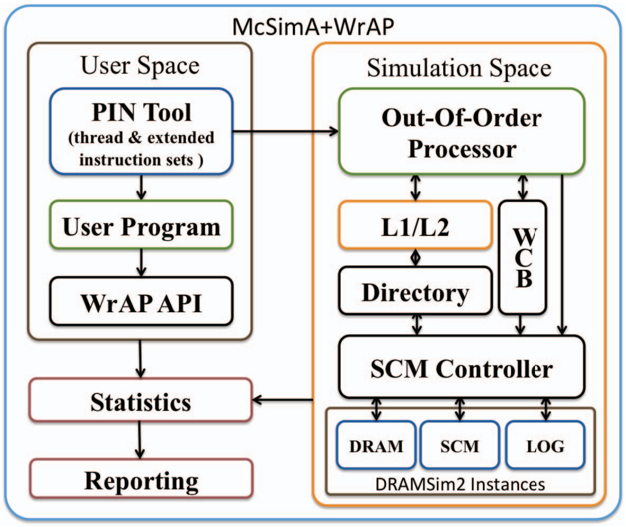

Figure 2: SCM Controller simulation is performed using extensions to the McSimA+ simulator and DRAMSim2 [13]. Applications are executed in user-space and PIN based traces from an extended McSimA+ are sent to an Out-of-Order simulation core with extensions for the SCM controller and DRAMSim2 for memory accesses.

4. Added support for DRAMSim2 [13], a cycle-accurate memory system and DRAM memory controller model library. This allows for finely tuned memory configuration parameters and easily adding channels and configurations for SCM.

5. Added write-combining and internal buffering (for write requests) pending in the SCM controller.

For the above modifications, the memory controller implementation of the combined McSimA+ and DRAMSim2 was extended as follows and as shown in Figure 2. The extended controller includes three memory channels as outlined in Figure 2: one for DRAM, one for SCM, and one for the persistent SCM Log area. A configuration option is provided to allow for the SCM Log area to be included in the main SCM area, with or without a separate channel. Each channel is simulated by an instance of DRAMSim2 and can be configured from a separate initialization file to model different types of SCM or have a slowdown factor cycle time as a multiple of DRAM cycle times. Our implementation of the SCM controller supports up to 128 open wraps using long long variables.

McSimA+ decouples the execution environment from the simulation environment. The rate of incoming transactions in determined in effect, by the rate at which the PIN instrumented program runs. However, we need to (a) capture timing events and (b) run McSimA+ and PIN-instrumented programs in alignment when there are saturation effects (buffer exhaustion, for example) or synchronization (due to PCOMMIT) effects. To achieve this coupling, we supplemented the simulation with a number of functions that collect and return fine-grained instrumentation from within the simulation of memory operations, so that we can both assess and control the execution side (loading) of the simulation environment with incoming transactions.

It is instructive to compare the wrap method advanced in this paper with three variations, listed below, for performing update transactions in place in SCM:

- **Undo-log:** Under this variant, the first store to each location is preceded by creating an undo-log entry (describing the location and its previous value) into a transaction log, and writing the undo-log entry synchronously (i.e., flushing it followed by a PCOMMIT), before the home SCM location is updated with the new value.
- **Non-atomic:** This method provides durability without guaranteeing atomicity. Each store to a persistent variable pushes the new value into SCM (e.g., using CLWB), but does not perform a PCOMMIT except at the end of the update transaction as a whole. Using this variant helps in delineating the impact of logging from that of storing the data values into SCM home locations and persisting them.
- **Cached / No persistence:** In cache mode, transaction begin and end calls are treaded as NO-OPs, and a store simply copies data to SCM addresses in the cache hierarchy. Obviously this method is extremely unsafe and it represents an extreme point where we would expect highest performance but no failure safety.

The WrAP API is extended to allow the above variations to be specified at run time. A preset environment variable chooses the appropriate variant in the implementation.

5.2 Simulator and Memory Model

In this subsection we describe the adaptation for streaming stores and evaluate the accuracy of simulation. The base machine model used for all evaluations is an Intel Nehalem equivalent, dual core CPU, with a simulated clock speed of 2.8 GHz. It is simulated together with a simulated 8 GB DDR3-1066 MHZ PC3-10600 Micron DRAM. We created a DRAMSim2 configuration that supported the timing parameters of the Micron based PC3-10600 DRAM DIMMS.

We tested the benefit of non-temporal streaming stores and write-combining, a key feature exercised by the solution described in this paper. To test the integration of these capabilities into DRAMSim2 in Figure 2, we created a large array (2 GB) that defeats being cached in much smaller L3 cache of the processor. Into this array we issue bursts of stores, testing both random and sequential address patterns. The sequential order tests the write-combining buffer implementation and its sensitivity to depth.

We execute a group of streaming stores to DRAM and vary the size of the group, executing them both on actual hardware and on the model. On actual hardware, we execute 20,000 groups of stores with a maximum rate of 1000 groups per second and obtain average time (in microseconds) to perform a group of stores. From that we derive the average time per store for that group size. The same test, on the McSimA+ and DRAMSim2 model, yields data for the same operations on the simulation model. A few iterations are sufficient for generating stable average under simulation.

In both simulation and on actual hardware, we experiment with group sizes ranging from 2 to 1024, performing the

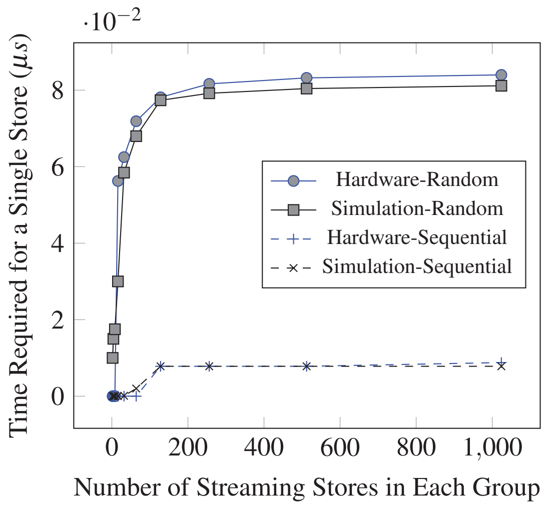

Figure 3: Hardware and Simulation Average Streaming Store Times to DRAM

sequential and random stores to the (2 GB) array. The results are shown in Figure 3. As the number of stores in a group increases, the average time required to perform a store increases. Very quickly however, the buffering to support write combining and for queuing up pending stores in the memory controller exhausts, and we get to a stable rate in which the rate of stores is a function of the bandwidth for stores available from the DRAM memory system (i.e., not from the short term ability of caches to absorb bursts of writes). When stores are performed sequentially, the write-combining effect allows for stores to happen faster since they can be combined in buffers. Our modifications to McSimA+ and the timing parameters captured for DRAMSim2 exhibit results that closely match measurements on actual hardware.

Let DRAM scale factor refer to a ratio of two speeds: the average rate at which a DDR3 DIMM can perform a media operation such as a read or a write, divided by the average rate at which an SCM module clocked at the same frequency can perform the same operation. A variety of factors are expected to change the DRAM scale factor - the type of operation (read or write) and the recent history of writes and reads to a given range to name a couple; and their effects are expected to change with SCM technology, endurance management, and internal caching within SCM modules (which may change over time and across product SKUs). Conversations with Intel architects about the recently announced Intel 3DXP^{TM} SCM technology lead us to assume that DRAM scale factor is likely to range between 1 and 4.

We next repeated the test of Figure 3 for SCM. We did so by using the simulation method, and choosing for SCM, three different DRAM scale factors of 2, 4, and 8 - in other words, by testing for three cases where per unit time, SCM accomplishes respectively 1/2, 1/4th, and 1/8th the number of read or write operations as DDR3. In this case, the stores in each group are directed to simulated SCM memory areas created by the pmalloc (persistent memory alloc) calls. The results are shown in Figure 4. The results, as one can expect, confirm that SCM simulated by DRAMSim2 saturates at the proportionally higher amount of time per store, in line with the simulated DRAM scale factor.

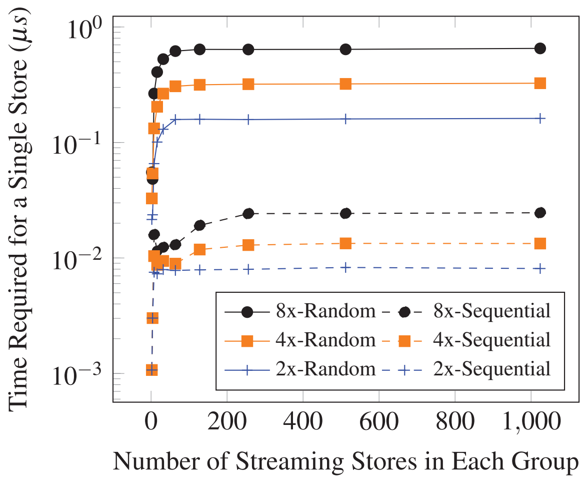

Figure 4: Average Streaming Store Times to Simulated SCM Channel with Scale Factor

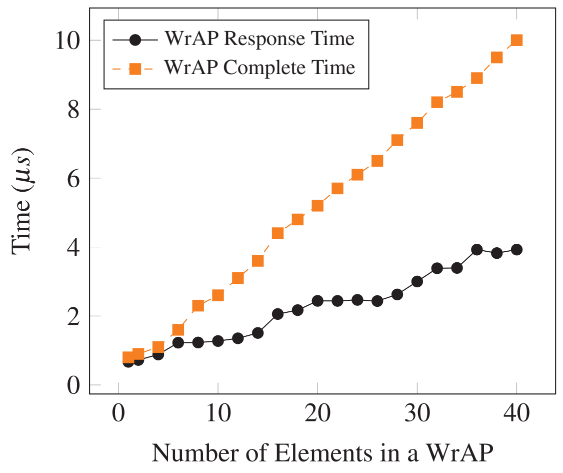

Figure 5: Average response time and completion time for an update transaction. Each transaction writes a fixed number of elements at random locations in a hash table. SCM speed has 4xDRAM slowdown.

5.3 Unit operations tests

With the evaluation setup calibrated and verified as described in subsections 5.1- 5.2, we next evaluate in this subsection the SCM controller and its integration with the simulator (that is, its cache hierarchy, processor, and write-combining buffers). First, we set the SCM speed at a 4x slowdown over DRAM (i.e., we set DRAM scale factor = 4). We create an 8 MB array (so that it can fit easily into the cache hierarchy), and initialized it with random values. We then streamed update transactions at a rate of 1,000 txps. Each transaction updated a parameterized number of randomly chosen locations in the array - the number of elements updated per transaction was varied.

We record the average response time and completion time of a transaction. The response time is the difference between the time at which the CloseWrap operation completes and the time at which the transaction arrives. The completion time is the time at which the updates in the transaction have been retired and made persistent in their SCM home locations. The retirement is done by the SCM controller in the background but from the viewpoint of the transaction it has reliably completed following the close of the wrap, since the log records of the updates have been safely recorded in persistent memory.

Figure 5 shows the average response time and completion time for an update transaction in microseconds, as a function of the number of locations updated in a transaction. It shows that the "background processing time" - that is, the retirement duration in the background SCM controller increases almost linearly with the number of elements in the transaction. This is understandable as retirement consists of reading from the logs and writing to possibly scattered home SCM locations. However, several optimizations are possible: first, a measure of efficiency is available on reading the logs, since each read of a cacheline from the log records furnishes multiple sequential logged values, whose SCM home locations are then updated. Additionally, it is possible to overlap the next read of the log record with the SCM writes to home locations.

Figure 5 also shows that the foreground response time for the transaction increases gradually and the increase proceeds in a series of discrete steps as the number of locations increases. This is because a transaction only needs to write the home locations directly in the cache hierarchy (whose time is negligible), while the time for log record writes are reduced due to write combining. The step changes in the figure reflect this effect.

We next repeat the tests of Figure 5 for the four variant implementations: wrap, undo-log, non-atomic, and cached. We execute them for 100 transactions of size 20 (i.e., twenty updates in SCM per transaction). Figure 6 shows the total numbers of different low level operations - home location writes driven into SCM, log reads and writes (not loads and stores but the numbers of cacheline operations), and PCOMMITs. Non-Atomic simply stores 2000 elements to main SCM locations with one PCOMMIT for each transaction, for a total of 100 PCOMMITs. Undo Log has a synchronous write to the log for each element so has an additional 2000 log writes, with each write to SCM and log requiring a PCOMMIT. For the wrap transaction only a log write with a PCOMMIT and writes to the cache hierarchy are needed. Log writes can be write-combined in a single transaction, so we see less than half the number of log writes performed. In the background, the SCM Controller reads the Log entries and writes them to the 2000 home SCM locations. In the foreground, the WrAP method has a much lighter overhead when compared to other persistence methods. Figure 7 does not include any statistics for the cached method since the array into which the writes are performed is contained within the processor L3 cache.

For the same tests described by Figure 6, Figure 7 compares the transaction rates of different methods for varying SCM speeds. The experiment also uses a burst of 100 transactions of 20 random writes of cached data per transaction. The number of writes that WrAP needs to perform for the foreground operation is less than half those of non-atomic; the reason for this is that the foreground writes for wrap are

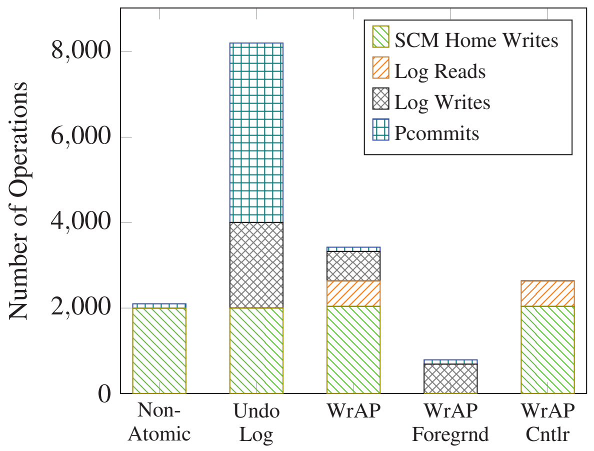

Figure 6: Number of SCM Writes, Log Read and Writes and PCOMMITs for different methods. Numbers shown are for 100 wraps of 20 elements each.

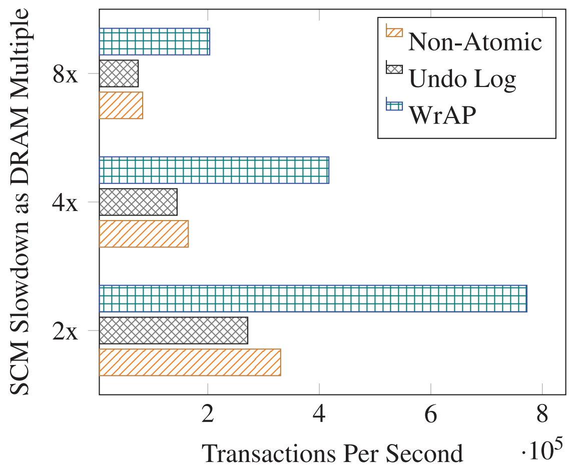

Figure 7: Transactions per Second of 20 elements each for burst traffic with various SCM Speeds

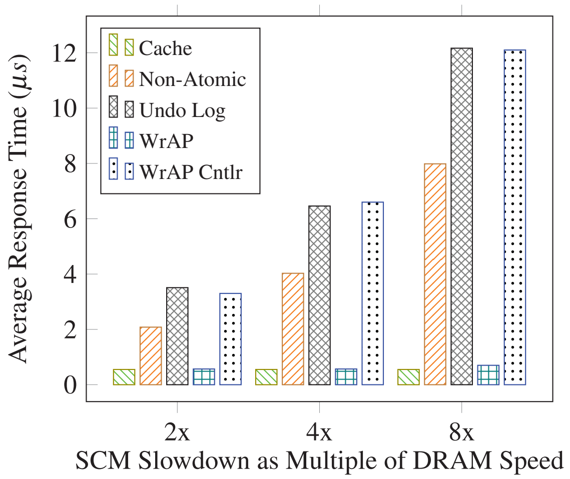

Figure 8: Average User Response Time for Transactions of 20 Writes Arriving at 10,000 txps

just the sequential, write-combining operations into the log, while the non-atomic method is forced to perform writes to scattered locations within each transaction. Even though the undo-log method performs more writes and PCOMMITs than the non-atomic method, its transaction rate does not lag the non-atomic method appreciably. This is because in the implementation we created, the log is on a separate channel from that used for data update operations into SCM.

As log writes are write-combined and go into the store buffer, a much faster response time is realized. This is depicted in Figure 8, which shows the transaction response time (i.e., the front-end completion time) at the different DRAM scaling factors, across the different methods employed. In the experiment of Figure 8, transactions arrive at an average rate of 10,000 per second, with each transaction performing 20 random writes of cached data in the array. The figure plots the average response time of a transaction. Unlike that in Figure 7, the measurement in Figure 8 is rate limited; thus where Figure 7 reports the maximum throughput for a burst of transactions, Figure 8 focuses on the average user time between the start and end of each transaction.

We also measure the background SCM Controller time and plot it (rightmost bars) alongside the response times. Significantly, the foreground response time for WrAP is comparable to that of the cache-mode (data updates are simply spilled into the cache) and not affected to any marked degree by the time to stream log writes and the overhead of a single PCOMMIT per transaction. The foreground or response time for WrAP is noticeably smaller than that of Non-Atomic and Undo-log methods. Figure 8 shows that the time it takes for the SCM controller to retire each WrAP transaction compares with the time it takes for the Undo-log approach - each having to update the same number of randomly scattered data locations in the SCM. Thus, the time to read the log to retrieve the data that is to be written out to SCM is not a determining factor.

Next, in Figure 9, we speed up the transactions considerably. As the rate increases, and transactions begin arriving fast enough, the background log retirements start to catch up with the speed of non-atomic method because the log reads begin to be streamed and read-ahead within the SCM controller, and overlap with the SCM controller's data writes into the SCM. Referring back to Figure 8 where we exercised the controller under a long-term sustainable rate of transactions, the controller was able to overlap the foreground operation of writing to the log with the background operation of reading from the log and writing to the home locations in SCM. But in Figure 9, the controller is driven into saturation, where the background path begins to dominate. Thus Figure 9 shows two ranges of operation for the wrap approach: before the onset of saturation (left), the write combining characteristic of wrap makes it faster than the (unsafe) Non-Atomic method, but later, both methods are rate limited - the rate at which the wrap controller can retire the log by writing to the SCM home locations now matches the rate at which the non-atomic method can write the same SCM home locations. At first the response time of Wrap is about the same as operating at in-cache speeds, while Non-Atomic and Undo-log are much slower, and then, as transaction arrival rate is increased, all methods (except cached mode) eventually run at speed of writing into SCM.

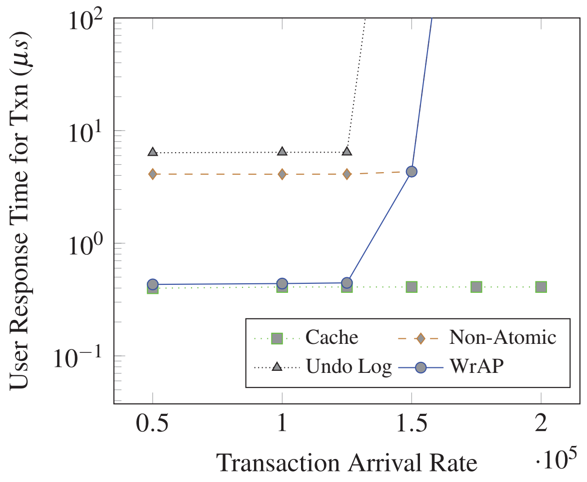

Figure 9: Average Response Time for Varying Transaction Arrival Rates of 20 Writes Each

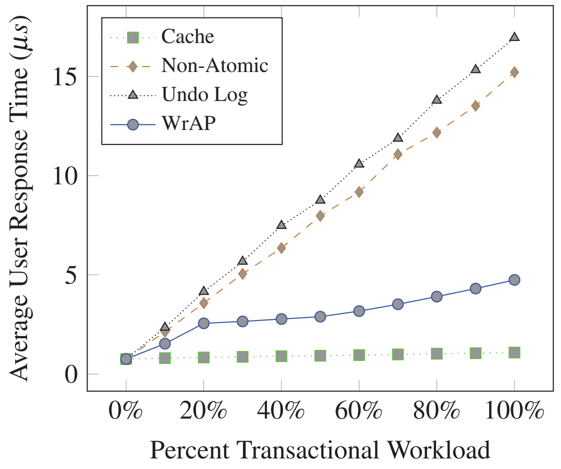

Figure 10: Average Response Time for Mixed Transactional and Non-Transactional Workload in Microseconds

Mixed Workload: We altered the controller protocol to handle both transactional and non-transactional traffic as described in Section 3.2. We created a workload of mixed writes in multiple groups of writes to SCM. Each group has 50 writes, with a varying percentage of transactional and non-transactional traffic. We measured the overall time required to perform the group of writes. The requests were performed at maximum speed, with no time between groups of writes. The results for varying the traffic type are displayed in Figure 10. When all of the traffic is non transactional, all methods perform the same. As the share of transactional writes goes up, so does the overall time. However, the rate of increase for Wrap is much slower than Undo Log and even Non-Atomic which does not guarantee atomicity.

In a stress test for the wrap controller, memory hierarchy, and victim cache, we reduced the cache sizes to just 2 sets for the L1 and 64 sets for L2/3 caches. The time required with the tiny caches was much higher as most of the operations are misses and evictions into the victim cache are increased, but the trends remained the same.

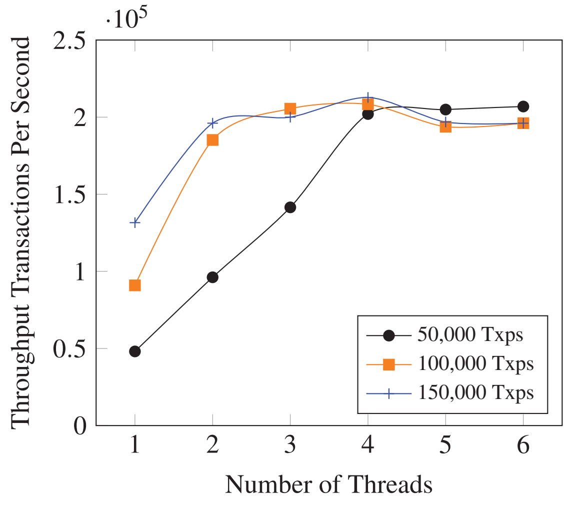

Figure 11: Throughput for WrAP for Varying Number of Threads and Different Arrival Rates

In Figure 11, we show the measured throughput of WrAP for transactions of groups of 20 writes at different arrival rates for varying numbers of threads. We create a thread that performs a number of transactions at the specified arrival rate and record the overall time and throughput. As the number of threads increases, the overall throughput is increased until the maximum memory bandwidth is reached. When the transaction arrival rate is high, the memory bandwidth limitation is reached with fewer threads.

5.4 Graph 500

In this subsection, we cover the first of our non-micro-benchmark based evaluations. The Graph 500 benchmark is designed to model real-world data intensive supercomputing workloads for High Performance Computing [14]. It creates a very large graph in memory and performs a series of searches for nodes in the graph.

We modified the reference C implementation of the benchmark so that the graph database and search trees are placed in persistent memory by using the pmalloc routine. Next, we identified the lines of C code where it was necessary to insert the WrAP calls, and performed the necessary modifications - which resulted in about 90 lines of source code changes to atomically persist the constructed graph in SCM. We instrumented the tests to obtain the generation time for edge lists, time for creating the graph, and search times; we obtained this detail alongside the performance statistics.

We executed the Graph 500 benchmark under different scale sizes for the benchmark, using a DRAM scale factor of 4. Figure 12(a) shows the node list generation time as a function of the graph scale. The node lists are in sequentially contiguous locations in memory, which makes the writes non-random during the node list generation. Figure 12(b) shows how the time to construct the graph varies with graph scale; and it is easily noted that WrAP enjoys a large speedup over the undo-log method; this is because during graph construction, node lists are read and copied to memory locations in non-sequential locations in SCM. Figure 12(b) also shows that WrAP compares well with cached- and non-atomic methods. Figure 12(c) plots the overall results and

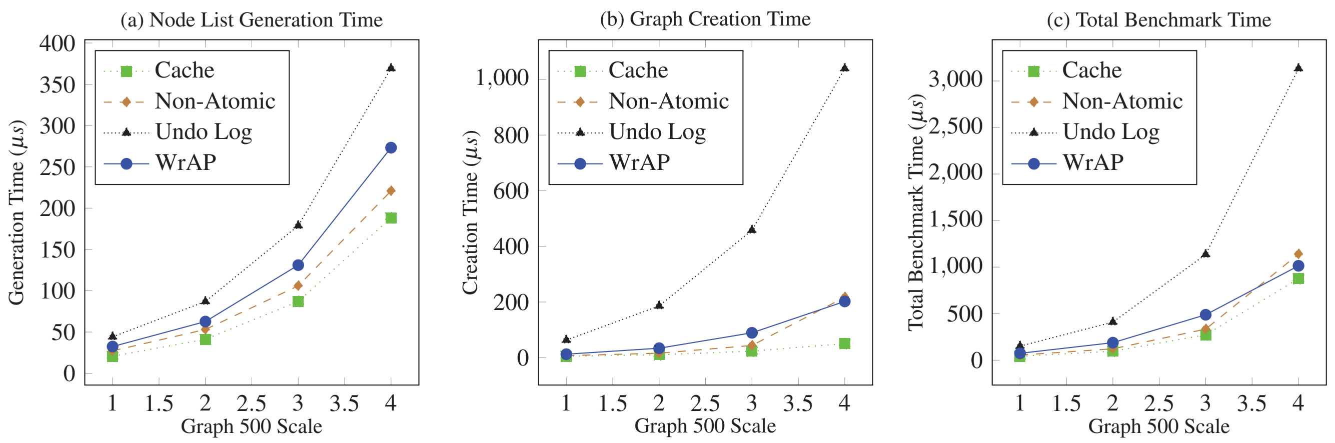

Figure 12: Graph 500 Benchmark Times for Various Persistence Methods Using SCM 4x Slowdown over DRAM Speeds

they include graph searching. Graph searching consists in part of atomically building the search trees. Here the WrAP method achieves roughly three fold speed up over the undo-log method. Interestingly, it operates at close to the speed of the cache method, and outperforms the non-atomic method since its writes to the log are write-combined and thus the PCOMMITs at the end of each transaction need to wait for a fraction of the time that the scattered home location updates require in the non-atomic method.

5.5 B+Tree

In this subsection, we describe the second of our non-micro-benchmark based evaluations. We modified the STX B+Tree, an in-main-memory B+Tree implementation with a Standard Template Library for C++ compatible interface [15], to use persistent memory. We created a persistent memory STL Allocator, one that utilized the pmalloc routine to return memory locations in the SCM region, when allocating space for new nodes in the B+Tree. Very small changes across roughly 50 lines of C++ implementation of B+ Tree and targeting primitive types and typecasting changes, were needed for this modification.

Figure 13 shows the average response time of the B+Tree for transactions of 20 inserts of new elements into the tree at a rate of 100,000 transactions per second. Response times for Wrap, Undo Log, Cache, and Non-Atomic implementations for varying SCM speeds are shown. Each of the 20 inserts in the transaction can cause multiple other operations in the B+Tree to write to SCM to rebalance or move nodes; thus the various insert or delete calls need to be treated as failure-atomic transactions against the B+ tree structure. Wrap performs nearly as well as the two unsafe methods (non-atomic, cached) while gaining the persistence protection in case of system failure. The undo-log method performs between two- and three-fold slower in comparison.

Figure 14 shows the relationship between the size of the number of elements inserted in each transaction and the response time seen by the callers. Transactions are performed at the rate of 100,000 per second of increasing size for each of the methods and the SCM slowdown factor is fixed at 4x DRAM. We again show that Wrap has a response time close to Non-Atomic and in-cache response times.

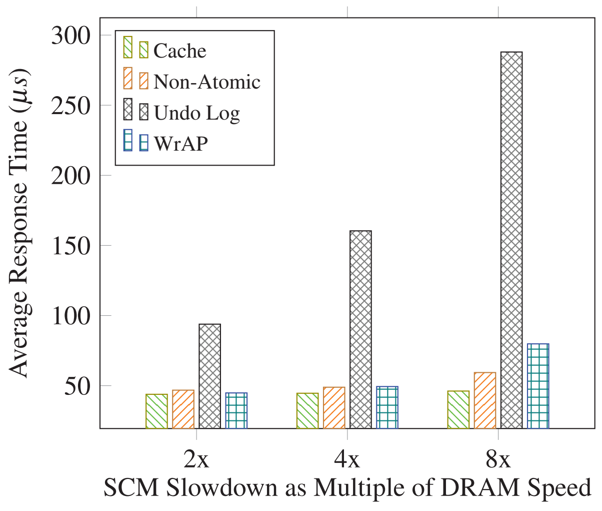

Figure 13: Average Response Time to Insert 20 Elements into a Persistent SCM B+Tree for Various SCM Times

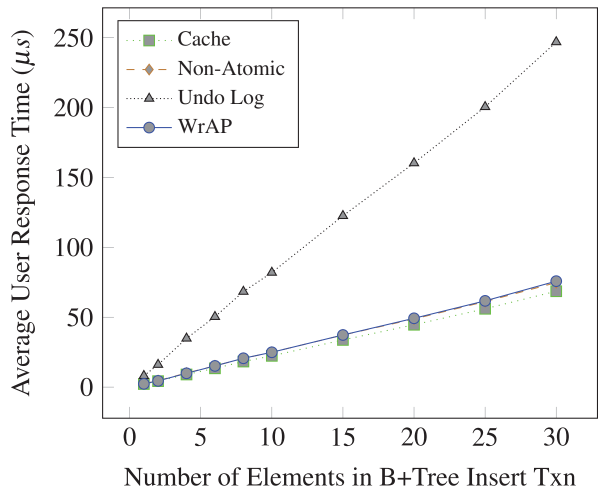

Figure 14: Average Response Time to Insert Elements to a Persistent SCM B+Tree for SCM 4x Slowdown

6. RELATED WORK

The fundamental issue of obtaining atomic updates of an arbitrary sequence of writes to persistent memory lies at the boundary between two well-studied fields: transaction processing systems and transactional memory. Transaction processing systems provide a superset of the failure atomicity and persistence problems considered here: that is, they guarantee all four ACID properties – atomicity, consistency, isolation, and durability, with sophisticated software-managed solutions [4]. *Isolation* shields an executing transaction from changes of state caused by concurrent updates in other uncommitted transactions, while *consistency* ensures that visible snapshots of a database always provide consistent views of the data. By contrast, transactional memory (TM) systems deal with main memory resident (volatile) data, but are only concerned with the issues of *atomicity* and *isolation* and do *not* treat *durability* –i.e., guaranteeing persistence in the presence of failures, as a goal. If a transaction aborts then the isolation mechanisms that prevent partial views from becoming visible are sufficient to allow rollback. Providing failure atomicity for persistence in memory thus straddles these two domains: it deals with devices accessed like main memory – orders-of-magnitude faster than disk, which prevents expensive software intervention; yet, it needs to handle arbitrary system failure and transaction rollbacks while maintaining the consistency of persistent storage (requirements C and D). Addressing the two issues simultaneously requires care: tight coupling between isolation and consistency in persistent memory shoehorns application developers into using a single concurrency control mechanism just to obtain persistence.

Mnemosyne [16] uses software transactional memory, or STM, to provide atomicity in persistent memory and supplements it with write ahead logging to achieve failure atomic commits into durable memory, an approach that meets ACID requirements within a single framework. Alternative software approaches to achieve ACID guarantees with logging in non-volatile memory and lock-based serialization include [17, 18]. ATLAS [17] combines concurrency control with persistence in an integrated framework. A custom atomic doubly linked list structure is used in [18] to minimize flushing overheads for its write-ahead log records. Our approach is to keep the two issues independent, so that application developers can blend any concurrency control policies and methods they choose, with means for achieving atomic and durable update of state in an SCM device.

Changes to the front-end cache for ordering cache evictions were proposed in [9,19,20]. BPFS [19] proposed *epoch barriers* and a copy-on-write approach to achieve controlled flushing of before and after values in caches. When combined with hardware-supported atomic writes to 8-bye words, this provides an effective mechanism for atomic updates of an important albeit restricted class of block-based tree-structured data structures, by using pointer flipping. The *flush* software primitive proposed in [20] and its timestamp-based multiversioning facilitates software control of update order. Ordering updates alone cannot guarantee atomicity of a sequence of updates, without a snapshot of the entire micro architectural state at the point of a failure [6]. A non-volatile victim cache to provide transactional buffering was proposed in [9], with the added property of not requiring logging; by comparison, our approach achieves efficiency through non-temporal write-combining streaming of log records. The design in [9] tracks pre- and post- transactional states for cache lines in both volatile and persistent caches and atomically moves them to durable state on transaction commits; however, our approach does not require changes to the front-end cache controller.

Research into new data structures such as in NV-heaps [21], which uses logging and copying, provide support for ACID components in software applications using persistent memory. CDDS [20] provides a versioning method that first copies data and then uses sequences of fences and flushes to provide transaction support. BPFS [19] and NV-heaps [21] require changes to the system architecture to support the atomicity and consistency of data. These are significant requirements since they change the front-end architecture with additional cache hardware and policies. Whole-system persistence described in [6] allows for in memory databases, but utilizes a flush-on-fail and not-flush-on-commit strategy that relies on batteries to power non-persistent memories on system failure. Research in persistent file systems built on SCM is also a promising area that might quickly enable software applications to take advantage of SCM. SCMFS uses sequences of **mfence** and **clflush** operations to perform ordering and flushing of load and store instructions and requires garbage collection [22]. BPFS uses copy on write write techniques along with hardware changes to provide atomic updates to persistent storage [19]. Ensuring atomicity of **sync** was discussed in [23].

Memory controller designs for persistent memory have been proposed in [24–27]. Adding a small DRAM buffer in front of SCM to improve latency and to coalesce writes was proposed in [24]. The use of a volatile victim cache to prevent uncontrolled cache evictions from reaching persistent memory was described in [26]. However the controller in that work assumes sequential executions of wraps; it also required complicated comparison of victim cache and log data entries during pruning, and log searching to handle overflows. In contrast the victim cache design here supports concurrent wrap executions and handles log pruning and victim cache space management in an integrated manner. FIRM [27] describes techniques to differentiate persistent and non-persistent memory traffic, and presents scheduling algorithms to maximize system throughput and fairness. Low-level memory scheduling to improve efficiency of persistent memory access was studied in [25]. Except for [26], none of these works deal with the issues of atomicity or durability of write sequences. An analysis of consistency models for persistent memory was considered in [7]. Finally a proposal for a software-only wrap library was presented in [28]. In contrast, this paper proposes a hardware controller design to non-intrusively provide support for atomicity at the back-end.

7. SUMMARY

In this paper we presented the design of a controller to provide support for atomicity of persistent memory transactions. Our approach does not require changes to existing processor caches, avoids synchronous cacheline write-backs on completions, and only needs to coordinate log retirement with deletion of entries in a volatile victim cache. The controller works in conjunction with a software library. The software API provides primitives for the programmer to mark the start and end of transactions and to mark the updates to persistent memory within the atomic region. Such wrapped updates result in the writing of a log record to the controller along with an ordinary store to the cache.

The SCM controller fields evictions and misses from the processor caches. It implements a transparent victim cache to hold evictions until the values are persisted to SCM via coupled log retirement operations. The log records of closed wraps are retired asynchronously to the home locations in SCM by backend operations of the SCM controller.

Two efficient implementations were described: one based on a hardware set-associative victim cache, the other with a DRAM-based data structure managed by firmware. The design does not defer visibility of updates, and thereby permits free and immediate propagation of updated values through processor caches. Simulation results show significant performance benefits over traditional approaches.

8. REFERENCES

[1] R. Freitas and W. Wilcke, "Storage class memory, the next storage system technology," *IBM Journal of Research and Development*, vol. 52, no. 4/5, 2008.

[2] 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.*, vol. 40, pp. 45–51, Jan. 2012.

[3] Intel, "Intel 3d xpoint," in *http://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html*.

[4] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Trans. Database Syst.*, vol. 17, pp. 94–162, Mar. 1992.

[5] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis and evolution of journaling file systems," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, (Berkeley, CA, USA), pp. 8–8, USENIX Association, 2005.

[6] 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, (New York, NY, USA), pp. 401–410, ACM, 2012.

[7] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, (Piscataway, NJ, USA), pp. 265–276, IEEE Press, 2014.

[8] L. Pu, K. Doshi, E. Giles, and P. Varman, "Non-intrusive persistence with a backend NVM controller," *Computer Architecture Letters*, vol. PP, no. 99, 2015.

[9] 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 *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, (New York, NY, USA), pp. 421–432, ACM, 2013.

[10] J. H. Ahn, S. Li, O. Seongil, and N. P. Jouppi, "Mcsima+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pp. 74–85, IEEE, 2013.

[11] I. Corporation, "Intel architecture instruction set extensions programming reference," October 2014. http://software.intel.com/.

[12] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM Sigplan Notices*, vol. 40, pp. 190–200, ACM, 2005.

[13] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, 2011.

[14] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray Users Group (CUG)*, 2010.

[15] T. Bingmann, "Stx b+ tree c++," in *http://panthema.net/2007/stx-btree/*.

[16] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, (New York, NY, USA), pp. 91–104, ACM, 2011.

[17] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "ATLAS: Leveraging locks for non-volatile memory consistency," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '14, (New York, NY, USA), pp. 433–452, ACM, 2014.

[18] A. Chatzistergiou, M. Cintra, and S. D. Viglas, "REWIND: Recovery write-ahead system for in-memory non-volatile data-structures," *Proceedings of the VLDB Endowment*, vol. 8, no. 5, pp. 497–508, 2015.

[19] 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*, SOSP '09, (New York, NY, USA), pp. 133–146, ACM, 2009.

[20] 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*, pp. 61–76, ACM Press, 2011.

[21] 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," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, (New York, NY, USA), pp. 105–118, ACM, 2011.

[22] X. Wu and A. L. N. Reddy, "Scmfs: a file system for storage class memory," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, (New York, NY, USA), pp. 39:1–39:11, ACM, 2011.

[23] S. Park, T. Kelly, and K. Shen, "Failure-atomic msync(): A simple and efficient mechanism for preserving the integrity of durable data," in *Proceedings of the 8th Eurosys*, pp. 225–238, 2013.

[24] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, (New York, NY, USA), pp. 24–33, ACM, 2009.

[25] P. Zhou, Y. Du, Y. Zhang, and J. Yang, "Fine-grained QoS scheduling for PCM-based main memory systems," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–12, IEEE, 2010.

[26] E. Giles, K. Doshi, and P. Varman, "Bridging the programming gap between persistent and volatile memory using wrap," in *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, (New York, NY, USA), pp. 30:1–30:10, ACM, 2013.

[27] J. Zhao, O. Mutlu, and Y. Xie, "FIRM: Fair and high-performance memory control for persistent memory systems," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, (Washington, DC, USA), pp. 153–165, IEEE Computer Society, 2014.

[28] E. Giles, K. Doshi, and P. Varman, "Softwrap: A lightweight framework for transactional support of storage class memory," in *Proc. 31st Symposium on Mass Storage Systems and Technologies*, MSST '15, IEEE, 2015.