Software Support for Atomicity and Persistence in Non-volatile Memory

Ellis Giles
Rice University
erg@rice.edu

Kshitij Doshi
Intel Corp.
kshitij.a.doshi@intel.com

Peter Varman
Rice University
pjv@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 framework 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.

1. INTRODUCTION
This paper examines the use of byte-addressable persistent memory (also called Storage Class Memory [10] or SCM) as a replacement for traditional non-volatile storage (hard disks or SSDs) in emerging data management applications. A growing number of applications use in-memory database technology that operate almost entirely from DRAM (e.g. SAP HANA [9], IBM solidDB [1], voltDB [3], and Neo4J [2]) for access efficiency and to meet throughput requirements [19]. However the volatile nature of DRAM makes these systems vulnerable to system crashes, often requiring ad-hoc checkpointing techniques to maintain a persistent copy of the data on non-volatile storage. This incurs both run-time overheads, and long recovery times following a crash or scheduled maintenance to rebuild the necessary in-memory structures.

SCM technologies [10] like Memristors or Phase Change Memory (PCM) provide a potential solution to this problem. SCM technology combines the cache-line granularity 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 these structures for disk access, or the loss of data due to a system crash. In-memory databases can avoid the complexity associated with adding disk-based checkpointing and logging as a separate mechanism, since the data stored in the SCM is non-volatile.

Obtaining transactional semantics for critical code sections that must be persisted atomically is normally provided by software by the file system or database management system. While this software intervention is reasonable when the backing device is slow block-based storage on the I/O bus, these approaches are onerous when the persistent store is accessed at main memory speeds using processor load and store instructions. In this paper we examine the problem of providing transactional support in a system using SCM for persistence, and describe a novel solution. We had examined the problem previously in our own previous work [8, 11]. However that solution required the use of architectural support for its implementation. These mechanisms improved over earlier hardware-based solutions [7, 16] by providing both ordering and atomicity and did not require changes to well-established cache front ends. Nonetheless, the need for changes to existing processor architectures limits the near-term application of the solution with existing system architectures. Therefore, in this paper we present a software-only approach to provide lightweight transactional guarantees. Unlike other software approaches, a specific aim of our solution is to provide maximum flexibility to the programmer by not limiting the types of isolation and synchronization mechanisms permitted (e.g. transactional memory in [23] or version-based concurrency [16]) nor to limit its use to specific applications (e.g. tree-structured file systems [7]).

In Section 2 we elaborate on the problem and describe our approach on Section 3. Our evaluation framework and early simulation results are presented in Section 3.3.

2. PROBLEM OVERVIEW
Consider a simple, single-threaded software program that might perform updates to account balances such as transferring money between two accounts or accruing monthly interest. The two functions below show separate in-memory account update operations for transfer and interest calculation.

```
Algorithm 1: Transactional Routines

transferFunds(fromAccount, toAccount);
begin
    Trans_Begin
    toAccount += amount;
    fromAccount -= amount;
    Trans_Commit

addMonthlyInterest(balances, numAccts);
begin
    Trans_Begin
    for (int i=0; i < numAccts; i++)
        balances[i] *= (1+rate/12);
    Trans_Commit
```

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.

MEAO'13, October, 2013, Montreal, Canada.
Copyright 2013 ACM X-XXXX-XX-X/XX/XX ...$15.00.
The group of operations contained between the Trans Begin and Trans Commit is called a transaction. A transaction must satisfy ACID properties: Atomicity (a transaction executes completely or not at all), Consistency (the state of the system is preserved), Isolation (a transaction is unaffected by other concurrent transactions), and Durability (the updates made by a committed transaction are permanent) [5]. To provide durability, transactional updates are recorded on non-volatile storage media before the transaction is committed, so that the updates can be recovered in case of power failure or a system crash. In traditional database systems updates are made via calls to the transaction manager that logs the changes to (typically) hard disks; a number of elegant techniques have been developed to hide the latency of disk-based logging [15]. When SCM is used as the durable medium, traditional arbitrated access has an overwhelming overhead and fails to exploit the potential advantages of direct memory stores and processor caching.

Guaranteeing transactional execution while exploiting the cache hierarchy is not straightforward. In the case of a balance transfer, if the system were to fail after the store operation updating toAccount but before the update of fromAccount, then the atomicity property would be violated. The problem is compounded because of the uncertainty in store completions in a high-performance memory system. Just because program control reaches commit without failure is not in itself a guarantee of successful atomic execution, since some or all of the updates might still be stuck in the cache hierarchy and not yet written to SCM. If the program were to flush all the updates from the cache to SCM after completing the transaction, a system failure could still happen during this flush violating atomicity. A similar problem exists in the interest calculation example. Should the loop be interrupted by a machine restart, it is impossible to know which of the accounts values in SCM represent updated values, even if program-visible state (like program counter and the loop index variable) had been saved at the time of interruption.

The method to make groups of write operations to persistent memory atomic is the main focus of this paper. This method is introduced as write-aside-persistence (or a WrAP), and is described in the following sections.

2.1 Achieving Atomic Persistence

In our model persistent objects are stored in non-volatile memory (SCM) and managed by an object-management layer that handles the naming and space allocation requirements. Applications use familiar mmap primitives to map the object into their virtual address space. Subsequently, memory load and store instructions are used to access the object for reading and writing the objects; these accesses are intercepted by the underlying caching mechanism and moved to and from the cache hierarchy as accesses to regular DRAM. Figure 1 shows two objects represented by the blue and brown rectangles in persistent memory, that are mmaped to regions of a processes address space. Reads and writes to these virtual addresses are simply handled as normal loads and stores to the physical addresses corresponding to the persistent memory devices. The SCM controller services reads and writes directed to persistent physical memory analogous to a traditional DRAM memory controller.

As noted previously, transactional applications require that a sequence of store operations to persistent memory be performed atomically even if interrupted by a machine restart. That is, following the restart the state of persistent memory should either reflect none of the changes of the atomic write sequence or all of these locations should reflect the updated values. For now we consider only a single thread and return later to the issue of concurrent access by multiple threads. A solution must exploit memory reuse by utilizing the processor cache hierarchy to transmit values both within a transaction as well as between transactions.

Figure 2 shows an additional issue that arises in implementing atomicity. The example atomic region includes writes to four persistent variables A, B, C and D. Suppose that the system crashes immediately after the store to C. At that point the updates to A, B and C have been recorded somewhere within the cache hierarchy, but may or may not have reached persistent memory. The figure shows a possible execution sequence where the cache line corresponding to B has been evicted to persistent memory due to normal cache management operations, while A and C are still present only in the cache. This is the complementary problem to that caused by some persistent updates not having been evicted to persistent memory at the time of the crash. Hardware-based cache control mechanisms [7, 16] to control the order of cache evictions have been proposed to mitigate such problems, but require significant changes to the cache structure. The implications of such changes on the wider cache operations have not been addressed. Our proposed approach does not require any modifications of the cache subsystem operations.

3. SOFTWARE WRAP APPROACH

In this section we describe a software framework called soft-Wrap to provide atomicity of an arbitrary sequence of writes to persistent memory that may be interrupted by a machine restart caused by hardware or software errors.
Algorithm 2: Programmer annotated atomic region

```
// x, p and array allocated by p_malloc are persistent.
wrap_open : :
begin
    x = 1; // Update persistent memory location
    .........
    p = p_malloc(100); // Allocate persistent memory array
    .........
    for (i = 0; i < 25; i++) ;
    begin
        [ p[i] = i;
        .........
    : wrap_close
```

The `wrap_open` library function returns a `wrap token`, an integer identifying the atomic region. Each open atomic region (also referred to as a `wrap`) has a unique identifier. Stores to persistent memory within a `wrap` are redirected to the library via the `wrap Store` call. The call takes three arguments: the `wrap` identifier, the address of the persistent memory location being written, and the value being stored. In the simplest version every store to a persistent location within a `wrap` are redirected through the library. There is a rich space of compiler optimizations that can be used to reduce this overhead using memorization or static inlining of code. We do not discuss these optimizations in this paper. In Algorithm 3 there are three such persistent store statements identified by `wrap Store` calls. These are used for `x`, the pointer to the unnamed array allocated by `p_malloc` and the accesses to the array elements themselves within the `for` loop. At the end of the atomic region the library function `wrap close` is called to complete the wrap operations and ensure that critical store values have been safely committed to persistent memory.

Algorithm 3: Wrapping of persistent stores in atomic region

```
wrapToken wid;
wid = wrap_open();
begin
    wrapStore(wid, &x, 1);
    .........
    temp = p_malloc(100);
    wrapStore(wid, &p, temp);
    .........
    for(i = 0; i < 25; i++) ;
    begin
        [ wrapStore(wid, p+i, i);
        .........
    wrap_close(wid);
```

3.1 Persistent atomicity using an undo log

A classic method of providing ACID guarantees in transaction management systems is the use of an undo log. In this approach, some variant of a copy-on-write mechanism is used to create a copy of an object in an undo log, before updates are applied to it. In case the transaction aborts or there is a system failure before all the updated values are committed to durable storage, then the system is rolled back to the instant before the start of the transaction using the original values in the undo log.

Algorithm 4 describes an implementation of a `wrap` using an undo log. A persistent store within a `wrap` invokes `wrapStore` as described previously. The routine first reads the current value of the variable and records its value along with its persistent memory address in an undo log. The persistent memory address corresponding to the virtual address `addr` of the `mmapped` object is denoted by `ϕ(addr)`. The log record needs to be committed to persistent memory before the store of the new value can be allowed to proceed. This is shown by the `P_MSYNC` call that is used to force all pending stores (and loads) to complete before execution continues. While similar in spirit to the common fence instruction `MFENCE` provided for memory synchronization in modern processors, `P_MSYNC` also provides a persistence guarantee; that is, all pending stores will have been committed to memory and not merely made visible to other processors using the coherence mechanism. Note there is no benefit to writing the log record to the cache since it is simply written once. Instead, we use streaming non-temporal store instructions for this purpose. These instructions bypass the cache and also employ write combining, which can be used to optimize the sequential write usage of the undo log. The `P_MSYNC` instruction is necessary to force the log contents in the write buffer to persistent memory. Finally the new value is written to the memory address in the cache in `write through` mode to allow the update to asynchronously trickle to persistent memory. When the `wrap` is closed, the program must ensure that all the updated values have reached persistent memory, which is accomplished using a `P_MSYNC`. In the absence of a `write through` mode, the updated values must be explicitly flushed and written back from the cache as discussed below.

As noted above, the virtual addresses generated by the program must be mapped to their physical persistent memory addresses and recorded in the undo log; otherwise their id can be lost along with the page tables in a system crash. Knowing the base address of an objects map and the accessing memory through an offset allows for a simple implementation of the mapping `ϕ`, without involving operating system intervention.

The second point to be noted is the potential performance impact due to many asynchronous memory operations. Each updated variable needs to be read and a log record with the old value must be synchronously written to persistent memory, before it can be updated. Also the new value needs to be written to persistent memory before the transaction is committed. If the updates are cached in `write-through` mode the updated values can be committed to persistent memory while retaining their cache accessibility. In a write back cache the updates need to be explicitly flushed from cache (using the CLFLUSH instruction for instance) and then persisted to memory. Note that CLFLUSH actually invalidates the corresponding cache line as well, which is undesirable since the updated values may still be needed within this transaction or by later transactions. Deferring the flushes to the end creates a write storm of updated values being written to persistent memory. Hence while CLFLUSH will work correctly performance may be an issue.

3.2 Persistent atomicity using softWrap

We now describe our solution `softWrap` that provides a more
The basic idea is to simultaneously propagate transactional updates along two paths: a foreground path through the cache hierarchy that is used for communication within and across transactions, and a slower asynchronous path to persistent memory. The latter path is used to create a redo log that records the new values for all updated memory locations. However, the writes to the redo log can be done asynchronously with respect to the rest of the transaction; the only requirement is that they be made persistent before the transaction ends. In contrast, each record of the undo log had to be made persistent before the corresponding memory location was updated.

Implementing the foreground path correctly without hardwared support can be tricky. The problem is that spurious cache evictions (described in Section 2.1) must be prevented from updating the locations in persistent memory. In [8, 11] we presented a hardware solution to this problem based on the idea of a Victim Persistent Cache that fielded persistent memory locations evicted from the last-level cache. In our pure software approach of this paper, we instead employ aliasing to redirect these updates to a different location where they can do no harm.

Algorithm 5 presents a basic implementation of the aliasing approach in SoftWRAP. When a persistent location is updated for the first time it is entered into a key-value store (implemented as a simple hash map table) that maps the virtual address to a different address which is backed up by a physical DRAM location. All wrapped accesses to a are redirected to address by looking up the table; reads and writes are done from location which will be cached. If evicted, the evicted value updates the shadow DRAM location rather than the persistent home location. Thus transactional communication takes place via the cache hierarchy: the aliased location (primed variables), while the record of updates is streamed to persistent memory asynchronously and concurrently in the form of redo log records.

Several optimizations and issues will be discussed in detail in the complete paper. We mention them briefly here. First, as an alternative to creating aliases in DRAM, one could instead simply alias them to their copy in the redo log record. This saves memory space by avoiding the extra DRAM copy, but requires the redo log records to go through the cache hierarchy. Evictions and cache misses would then need to access slower SCM rather than DRAM, which could potentially cause performance issues when the cache pressure is high. In the latter approach, the aliased location will change as different transactions access the variable and alias it to their private log locations. Frequent updates will cause increased coherency traffic as hash table entries are repeatedly invalidated, in contrast to the DRAM-based solution where the alias addresses do not change till the corresponding entry is deleted. To keep the size of the alias memory bounded, the backing space needs to be deallocated periodically. This can be safely done once the latest value of a variable has been copied from the corresponding redo log to its home location. Care is needed to avoid inconsistencies in alias address when multiple concurrent threads access the same variable; the details depend on the semantics of the isolation modes that are supported. For instance, under strict serializability one can show that there will be no races in accessing the hash map entries. Relaxed isolation modes will provide non-deterministic but consistent semantics. A final consideration concerns the mapping of a shared persistent object in the address space of multiple threads. As is commonly done for shared libraries, in this implementation we assume a fixed mapping based on common agreement, in preference to more costly dynamic alias conflict handling mechanisms.
per cache line, sequential update of the entire array, and random access. The next benchmark utilizes an implementation of an expandable WrapArray template class which can be initialized and compiled to support additions to the array of any type of variable sized data structure. The access to the array operations such as get, set, resize, and the operator overload of the array reference all utilize the WrAP semantics to make sure that an array update is atomic. If several operations to the WrapArray need to be atomic, they can be grouped in a higher level Wrap Open / Close pair which will override lower level Wrap Open and Closes within the data structure. The BenchArray program utilizes the WrapArray template to create an array and perform a variable number of Wraps with a configurable number of Wrap Store operations per Wrap. The final benchmark utilizes an implementation of a WrapMap template class much like the WrapArray template implementation described previously except for a Hash Map implementation. The WrapMap maps keys to values and provides a number of atomic operations to persistent memory, such as Put, Get, and Remove, which utilize the Wrap API. The BenchMap program utilizes the WrapMap template to create a map of integers and perform sets of configurable numbers of atomic random writes to the map.

The analysis procedure is shown in Figure 4. Each benchmark program is instrumented at runtime using Pin [14]. For the evaluation we need to distinguish accesses to persistent memory from accesses to volatile DRAM. We created a Pin tool, WrapPin, that instruments all memory access and instruction references along with injecting run-time code to overload functions such as P_Malloc, P_MSync, wtstore, and clflush. Our system is modeled after a 3 GHz processor. Every instruction incurs a one cycle cost. On each instruction and memory reference, WrapPin passes the request to the cache hierarchy. P_Malloc is overloaded in order for WrapPin to note the virtual address ranges in the user program space which the user benchmark program deems to be persistent. We model our cache hierarchy after an Intel 5660 Processor with split L1 data and instruction caches of 32 KB each, a L2 unified cache of 2 MB, and an L3 cache of 16 MB with 64 byte cache lines. L1, L2, and L3 access incur 1, 10, and 40 cycles respectively. On a cache miss, eviction, flush, or write-through store (wtstore), the memory request is checked against a fast data structure to check if the virtual address is contained within the virtual address ranges obtained from overloading the P_Malloc routine. If the address is in the persistent range it is sent to a memory model that models PCM, otherwise it is sent to a DRAM model. Both our memory models utilize DRamSim2 [20] as a cycle accurate memory simulator. The DRAM is simulated as a Micron DDR3 666 MHz DRAM memory with burst length of 8. For PCM, we utilize the parameters used in [12] for the experimental setup of PCM. These parameters are modeled after Micron’s DDR2-800 SDRAM and further modeled by [13] which operates at 40 MHz and has a burst length of 4. The P_MSync overloaded function waits and cycles the memory subsystems until all writes are out of the PCM write buffer. The wtstore places writes both in the cache and directly in the write buffer, and the clflush invalidates and writes back to memory the cache entry for the specified address is dirty.
3.5 Experimental Results

For the first part of our evaluation, we examined the speedup of the SoftWrap implementation over a traditional UndoLog for the various data structures and options. First, we executed the ArrayUpdate program for a single Wrap of a varying number of random array writes for both SoftWrap and UndoLog and recorded the number of total system cycles. The results are shown in Figure 5. The number of cycles required by the UndoLog approach grows much faster for the number of contained Wrap Stores than the SoftWrap implementation. The speedup for all three access patterns is shown in Figure 6. In all cases, the speedup of SoftWrap increases with the number of Stores per transaction as the operations are done more in cache and don’t have to incur synchronous operations of the Undo Log approach.

Figure 5: Cycles for Wrapped Random ArrayWrites

Figure 6: Array Write Speedup of SoftWrap Over an UndoLog

The additional benchmarks for the WrapArray Template and WrapMap Template were also analyzed. The speedup of using SoftWrap is shown in Figure [?]. The ArrayTemplate performs at almost a 2x speedup over an UndoLog approach and Map Template at about 1.5 speedup.

Next, we examined increasing the number of Wraps in a given benchmark while performing a varying number of Wrap Stores in each Wrap. We utilized the ArrayUpdate program and varied the number of consecutive Wraps from 1 to 10 while testing the number of stores contained within each Wrap. The results are shown in Figure 8. In all cases, the speedup of the SoftWrap implementation increases with the number of consecutiveWraps. With many updates per Wrap (around 600), and multiple consecutive Wraps, a speedup of over 3 can be obtained.

We have done a number of other experiments which have been omitted for reasons of space. They will be presented in the full paper. We evaluated the number of P_MSync, PCM Store and PCM Load operations performed by both an UndoLog and for SoftWrap for a single Wrap Open/Close with the varying number of enclosed Wrap Writes. For SoftWrap there is just a single persistent memory sync per wrap, while for an UndoLog the persistent memory syncs grow one-for-one linearly with the number of enclosed Wrap writes. These PCM Store operations required by the UndoLog grow much more rapidly than for a SoftWrap which is only linear to the number of enclosed stores. This is due to the UndoLog requiring multiple synchronous writes to the undo-log, roughly 3 per each wrap-write. These numbers are identical to any varying of number of wraps times number of operations per wrap since they each involve the product of the two as the number of writes. Therefore, the number of reads and cache performance give rise to the increase in SoftWrap for consecutive Wrap Open/Close operations. For PCM Loads, the number of PCM reads increases for a single wrap that has increasing numbers of enclosed wrap writes. This is due to creating the log structure and loading any required variables into the cache. However, if the number of operations within a wrap is held constant at 600 and the number of wraps increased, which gives rise to a comparable number of wrap store operations, the number of loads only increases slightly which reduces the overall cycle time and increases performance over the increase in the number of reads and writes required by the UndoLog.

4. REFERENCES


