Non-intrusive Persistence with a Backend NVM Controller

Libei Pu[†], Kshitij Doshi^{*}, Ellis Giles[†], Peter Varman[†]
[†] Rice University ^{*} Intel Corp.

Abstract—By providing instruction-grained access to vast amounts of persistent data with ordinary loads and stores, byte-addressable storage class memory (SCM) has the potential to revolutionize system architecture. We describe a non-intrusive SCM controller for achieving light-weight failure atomicity through back-end operations. 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 memory controller with a firmware managed volatile space, are described.

Index Terms—Non-volatile memory, persistent memory, atomicity, durability, consistency

1 INTRODUCTION

The high capacity of emerging storage class memory (SCM) devices allows applications to access vast amounts of persistent data directly using ordinary load/store instructions, without having to pack and unpack it for durable storage on disk. SCM allows applications to use in-memory computing freely, since its fine-grained accessibility facilitates popular applications involving graph traversals, pointer chasing, searching, indexing, and many other data structure intensive algorithms. Since SCM is nonvolatile it raises 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 SCM. First, the complex intervention techniques that databases and operating systems 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, SCM is part of the memory subsystem and interacts directly with the processor cache hierarchy, leading to new problems caused by uncontrolled cache evictions. 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.

2 OVERVIEW

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). We describe how to make such an update sequence atomic across a potential machine restart, while allowing their values to be freely communicated via the CPU cache hierarchy for performance. To achieve this failure atomicity, two complementary problems need to be solved as described next.

The first problem is ensuring that writes have actually been made durable on the non-volatile medium. Processors make no guarantees about when exactly new values of x , y , and z will flow out of their cache locations or be durably written into SCM. 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. Even cache flushing instructions

and non-temporal stores complete asynchronously from volatile store buffers, and barrier instructions (such as SFENCE) only control the order in which stores are made visible to other threads. Neither can guarantee durability at a fine instruction-grained level. Consequently, manufacturers are providing a persistence commit instruction (e.g., x86 PCOMMIT) with which software can achieve confirmation that previously flushed cache lines have reached a power-safe domain. We assume the existence of an instruction like PCOMMIT. However, while this can provide the necessary durability for a single update, it is not sufficient by itself to provide atomicity. A cache line flush after every store to persistent memory followed by a fenced PCOMMIT can be used to ensure the *ordering* of writes into persistent memory. However, a failure within the atomic section will still leave the memory in an inconsistent state, and additional complex hardware mechanisms [1] to precisely restart the system are needed for recovery.

The second problem is background evictions by the cache controller: modified cache lines are written back autonomously and there are no guarantees when such evictions occur. An arbitrary set of stores from an atomic region can be evicted to the SCM unknown to software, while others are held back in caches or volatile buffers along the way. Thus, programmers need to monitor the intended order of updates to SCM in separate metadata structures (logs, journals, etc.) for replay following a machine restart to remove the effects of incomplete or out of sequence updates in SCM.

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);
```

Although similar issues arise in relaying buffered updates from volatile memory to disks, the durable data in such systems are accessed indirectly through the services of a file system or a database storage manager, which effectively coordinates the sequencing of updates and manages recovery from torn updates. 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 persistent memory. Instead, the home locations for the variables (in SCM) 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 volatile 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 (in the comments section on the right) shows a pair of library calls **OpenWrap** and **CloseWrap**, 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 a *wrap controller*.

When a wrap is *opened*, **OpenWrap** allocates a *log bucket* in a non-volatile memory area known to the wrap 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 $z = x$ at line 5 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 the specified 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 the 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 PCOMMIT, SFENCE). 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.

Write back stores of single variables are optimized by fusing the three library operations – **OpenWrap**, **WrapStore**, and **CloseWrap** –

into an anonymous wrap that writes a singleton log record. These transactions are completed without a synchronous fencing of the log record, unless the programmer has manually fenced the writes to force an ordering. Further, any streaming (non-temporal) writes to SCM are considered as controller bypassing writes.

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 the 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.

Fig. 1. Wrap Controller: Intercepts cache misses and evictions and log records. Log records must be in NVM while victim cache can be volatile.

3.1 Wrap Controller Algorithm

The wrap 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 operation of the wrap controller abstractly at a high-level in Algorithm 2. We follow this with two implementations (controller hardware, or controller firmware based) in Section 4.

The wrap 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 controller routine **RetireWrap** 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 tracks all wraps between their opening and retirement 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, and retired asynchronously with backend writes into persistent memory home locations (see **RetireWrap**).

On a cache eviction, an evicted block B is stored in the victim cache and is tagged with its *dependence set* DS_B : DS_B indicates the wraps that were open when B was evicted but 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 as follows.

Suppose block 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 $w \notin DS_B$, B has been written to SCM and can be safely deleted from the victim cache. If $w \in DS_B$, then B may be part of a currently open wrap, and not yet updated 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. At that time B and all other blocks that may have been updated by wrap w have been written to persistent memory.

Algorithm 2: Controller Algorithm

```

HandleOpenWrap (wrapId  $w$ )
    openWraps = openWraps  $\cup \{w\}$ ;

HandleCloseWrap (wrapId  $w$ )
    Add bucket  $w$  to end of closedBuckets list;

CacheEvict (block address  $B$ , data  $d$ )
     $DS_B = \text{openWraps}$ ;
    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  $a \in \text{bucket}(w)$ 
        Store  $\text{val}(a)$  in SCM address  $a$ ;
    for each block  $B \in \text{VictimCache}$ 
         $DS_B = DS_B - \{w\}$ ;
    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 $t = 1, 3$ 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.

In the above example, the last update to A must have been by one of wraps 1 or 3, and since both these log buckets have been retired by $t = 10$, the latest value of A is reflected in its home location, after $t = 10$. Block B may also have been last updated by either of these wraps (and hence safe to delete), or the last wrap to update it would already have had its log retired (due to FIFO retirement order) by this time, which means B's latest value is safely reflected in persistent memory. This behavior 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.

4 IMPLEMENTATION

Algorithm 2 may be implemented as a hardware structure or as firmware extensions to a memory controller. We describe both alternatives in this section.

TABLE 1
Example Trace
 t : Time, op : Operation, ow : openWraps
 E_B : Evict(B) M_B : Miss(B) O_w : HandleOpenWrap(w), C_w : HandleCloseWrap(w), R_w : RetireWrap(w)

t	1	2	3	4	5	6	7	8	9	10
op	O_1	E_A	O_3	E_B	M_A	E_A	C_3	C_1	R_3	R_1
ow	$\{1\}$	$\{1\}$	$\{1, 3\}$	$\{1, 3\}$	$\{1, 3\}$	$\{1, 3\}$	$\{1, 3\}$	$\{1, 3\}$	$\{1\}$	ϕ

Hardware Implementation: In the hardware implementation *openWraps* is maintained in a bit vector. Its size is a limit number for the number of concurrent wraps open at a 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. A new wrap can then take the id of one of the cleared bits. 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 the same as 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. At that point, if the bit vector representing the dependence set of a victim cache block becomes 0, the block can be deleted and reassigned.

Ordinarily, log retirements going on in the background prevent the victim cache from becoming saturated. On rare occasions, the number of active victim cache blocks (*i.e.*, with non-empty dependence sets) may completely fill up a set and thus hinder the ability to handle further processor cache evictions. One solution is to use a more space-efficient structure like a cuckoo hash table at the expense of more complex hardware. Alternatively, we propose storing the entries overflowing the victim cache in a DRAM area managed by the controller firmware. If an entry cannot be placed in the victim cache, the controller adds it to a DRAM overflow area, which is searched when a processor cache miss is not satisfied in the victim cache. By appropriate sizing of the victim cache the probability of such an overflow can be reduced.

The above implementation of the wrap 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 for the victim cache as mentioned above.

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 *CacheMiss* the controller looks up the KVS to retrieve the data. 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 as follows.

When a wrap retires, it is added to a set R and also removed from the dependence set of the head entry of DFIFO (if it is not in the dependence set, it is ignored). When the head entry has a null dependence set, its pointer into the KVS is followed and the block is deleted from the KVS. DFIFO is then walked towards the tail. At each entry, all wrap in R are removed from its dependence set, and if now null, the corresponding KVS entry is deleted. The walk continues till an entry with non-null dependency is found. This becomes the new head of the queue. If w is not in any dependence set at the time of its retirement it need not be placed in R . Due to the inclusiveness property, this only requires checking that w is not in the dependence set of the tail entry of DFIFO.

A wrap id w is recycled (and deleted from R) when no entry in DFIFO has w in its dependence set. This can be checked during the walk of DFIFO. If w is in the dependence set of the current but not the following entry in DFIFO (or this is the last entry) it can be recycled. Finally, to avoid unnecessary stalls due to the lazy release of wrap ids, it is sufficient to alternate between two versions for each wrap id, to achieve the effect of an unbounded number of wrap ids.

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.

We note that the write traffic to the wrap controller (outside of evictions) occurs as compact write-combined log records, which are then written by the controller to the backend SCM as scattered data writes. There is one write per log record though multiple writes to the same location within a log bucket can be combined with no effort.

5 RELATED WORK

Memory controller designs for persistent memory have been proposed in [2]–[5]. Adding a small DRAM buffer in front of SCM to improve latency and to coalesce writes was proposed in [2]. The use of a volatile victim cache to prevent uncontrolled cache evictions from reaching SCM was described in [4]. 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 [5] 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 [3]. Except for [4], none of these works deal with the issues of atomicity or durability of write sequences. Analysis of consistency models for persistent memory was considered in [6].

Changes to the front-end cache for ordering cache evictions were proposed in [7]–[9]. BPFS [7] proposed *epoch barriers* to control eviction order by tagging cache blocks with epoch numbers. The *flush* software primitive proposed in [8] 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 [1]. 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.

Software approaches to achieve atomicity in persistent memory have been proposed [7], [8], [10]–[14]. Some solutions [11], [12] combine concurrency control with persistence in an integrated framework. BPFS [7] uses epoch barriers and shadow paging to provide atomicity of page-based tree-structured file systems using copy-on-write mechanisms. CDDS [8] provides atomicity using a persistent multi-version B-Tree to maintain time-ordered versions of the database, and [10] studies persistence issues in non-volatile memory heaps. A custom atomic doubly linked list structure is used in [14] to minimize flushing overheads for its write-ahead log records. Finally a proposal for a software-only wrap library was presented in [13]. In contrast, this paper proposes a hardware controller design to non-intrusively provide support for atomicity at the back-end.

6 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 or store instructions, avoids synchronous cacheline writebacks 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 each ordinary store into the cache hierarchy.

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

REFERENCES

- [1] D. Narayanan and O. Hodson, “Whole-System persistence,” in *ASPLOS*, 2012.
- [2] M. K. Qureshi, V. Srinivasa, and J. A. Rivers, “Scalable high performance main memory system using phase-change memory technology,” in *ISCA*, 2009.
- [3] P. Zhou, Y. Du, Y. Zhang, and J. Yang, “Fine-grained QoS scheduling for PCM-based main memory systems,” in *IPDPS*, 2010.
- [4] E. Giles, K. Doshi, and P. Varman, “Bridging the programming gap between persistent and volatile memory using WrAP,” in *CF*, 2013.
- [5] J. Zhao, O. Mutlu, and Y. Xie, “FIRM: Fair and high-performance memory control for persistent memory systems,” in *MICRO-47*, 2014.
- [6] S. Peley, P. M. Chen, and T. F. Wenisch, “Memory persistency,” in *ISCA*, 2014.
- [7] J. Condit *et al.*, “Better I/O through byte-addressable, persistent memory,” in *SOSP*, 2009.
- [8] S. Venkatraman *et al.*, “Consistent and durable data structures for non-volatile byte addressable memory,” in *FAST*, 2011.
- [9] J. Zhao *et al.*, “Kiln: Closing the performance gap between systems with and without persistence support,” in *MICRO-46*, 2013.
- [10] J. Coburn *et al.*, “NV-Heaps: Making persistent objects fast and safe with next generation, non-volatile memories,” in *ASPLOS*, 2011.
- [11] H. Volos, A. J. Tack, and M. Swift, “Mnemosyne: Lightweight persistent memory,” in *ASPLOS*, 2011.
- [12] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, “ATLAS: Leveraging locks for non-volatile memory consistency,” in *OOPSLA*, 2014.
- [13] E. Giles, K. Doshi, and P. Varman, “Wrapping operations for atomicity and durability,” in *HPEC*, 2014.
- [14] 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, 2015.