ATOM: Atomic Durability in Non-volatile Memory through Hardware Logging

Arpit Joshi, Vijay Nagarajan, Stratis Viglas
University of Edinburgh
arpit.joshi@ed.ac.uk, vijay.nagarajan@ed.ac.uk, sviglas@inf.ed.ac.uk

Marcelo Cintra
Intel, Germany
marcelo.cintra@intel.com

***Abstract*—Non-volatile memory (NVM) is emerging as a fast byte-addressable alternative for storing persistent data. Ensuring atomic durability in NVM requires logging. Existing techniques have proposed software logging either by using streaming stores for an undo log; or, by relying on the combination of `clflush` and `mfence` for a redo log. These techniques are suboptimal because they waste precious execution cycles to implement logging, which is fundamentally a data movement operation. We propose ATOM, a hardware log manager based on undo logging that performs the logging operation out of the critical path. We present the design principles behind ATOM and two techniques to optimize its performance. Our results show that ATOM achieves an improvement of 27% to 33% for micro-benchmarks and 60% for TPC-C over a baseline undo log design.**

I. Introduction

Byte-addressable non-volatile memory (NVM), also known as persistent memory, is a new type of memory that aims to bridge the gap between memory and storage and is fast emerging as a new tier in the memory and storage hierarchy. NVM can be realized, for instance, using the recently announced 3D XPoint memory [1] and various other technologies under development like PCM, STT-MRAM and ReRAM. Because of its low power and high storage density properties, NVM is widely expected to replace or complement DRAM in future systems [2], [3]. NVM incorporates the speed and byte-addressability of volatile memory and the persistence and high capacity of storage. With NVM, applications can access vast amounts of persistent data using the fast (load/store) processor interface, without having to pay the costs of packing/unpacking data in/out of storage and executing expensive system calls.

The Problem. An important challenge in designing NVM systems is guaranteeing the consistency of persistent data in the presence of system failures. Data structure consistency is required for the correct recovery of program state after a failure. Consider the example of two cache lines being modified as part of an atomic update to a data structure. If the system crashes after one of the cache lines reaches NVM, then the data structure is left in an inconsistent state because of the partial update to NVM. To avoid this scenario, a mechanism for *atomic durability* needs to be provided.

Recovery mechanisms like write-ahead logging [4] have been employed to provide atomic durability in many NVM proposals [5], [6], [7], [8], [9]. These mechanisms operate on the principle of physical logging: maintaining a persistent copy of the old and new versions at all times during the atomic update so that state can be recovered to either of the versions. Write ahead logging writes undo or redo log entries for all data updates, and enforces the ordering constraint that all log entries become durable before any data update (log → data ordering). In systems with NVM, log implementations rely on instructions like *non-temporal stores* and *cache-line write backs* to durably write log entries to memory. Moreover, ordering constraints to memory have to be explicitly enforced using instructions like *pcommit* and *sfence* [10], [11], [12].

Support for atomic durability using the above method has a fundamental drawback: durably writing log entries to NVM is in the critical path of execution. Since the software has no control over when a cache line is flushed out of the cache, any data update cannot be performed until the corresponding log entry persists in NVM, which can result in significant performance degradation. Our experiments with a set of micro-benchmarks show that durably writing log entries in the critical path degrades throughput by 40% on average and upto 70% (Figure 5: BASE vs NON-ATOMIC).

Our Approach. Our goal is to reduce the overhead of logging by moving it out of the critical path. We observe that logging, fundamentally, is a data movement task associated with stores in the original program. Our insight is to perform logging transparently in hardware by: (i) coupling log writes with data stores; and (ii) co-locating data and their corresponding log entries at the same memory controller. In doing so, we not only minimize wasteful data movement, but also enforce log → data ordering constraint in the memory controller (out of the critical path).

We propose ATOM: a hardware log manager to guarantee atomic durability through transparent and efficient logging. ATOM manages log allocation, ordering and log truncation in hardware. At the same time, ATOM is distributed across memory controllers and handles logging for multiple threads on a multicore processor. Our logging design is in many ways similar to the data movement tasks offloaded to a DMA engine. Offloading logging to a log manager in hardware frees up CPU resources, and relieves the programmer from explicitly implementing the logging logic.

In ATOM, we expose atomic durable regions to hardware via ISA support (Atomic_Begin and Atomic_End instruc-

2378-203X/17 $31.00 © 2017 IEEE
DOI 10.1109/HPCA.2017.50

tions). Stores in this region that require logging (i.e., the first store to a cache line) are detected dynamically and the log write corresponding to the store is performed transparently.

We leverage operating system (OS) support to reserve log space behind each memory controller. ATOM ensures that a log write is sent to the same memory controller as that of the corresponding data. This allows us to efficiently enforce the log → data ordering constraint at the memory controller level, thereby moving the ordering overhead out of the critical path. We also propose an optimization called source logging in which the memory controller eagerly performs logging for read exclusive requests, thereby eliminating wasteful data movement.

Finally, we ensure that the log structure is preserved for recovery by forcing every memory controller to flush critical hardware structures (128 bytes per memory controller) to the NVM. Recovery is then ensured through a routine implemented as a system call that undoes all the updates that were incomplete at the time of the crash.

Contributions. In summary:

- We propose a log organization that allows us to eliminate log persist operations from the critical path of program execution by enforcing log → data ordering at the memory controller (§III-C).
- We propose an optimization to minimize data movement for log entries by dynamically identifying when logging can be done at the source (§III-D).
- We propose an efficient log manager in hardware that manages allocating log space, writing log entries and truncating logs transparently with only 3.125 KB overhead per memory controller (§IV).
- We evaluate ATOM and show that it can improve performance by 27% to 33% for micro-benchmarks and by 60% for large-scale transactional workload (TPC-C) over a baseline undo log design. ATOM also compares favorably with a competing approach which provides support for redo logging (§VI).

II. Background and Motivation

A. Logging Techniques

Storage systems have traditionally employed write-ahead logging (WAL) [4] or shadow paging [13] to provide atomic durability for a group of writes – an *atomic update*. While shadow paging is useful if writes belonging to an atomic update happen at page granularity, WAL works better for atomic updates consisting of scattered writes which happen at a cache line or finer granularity [5], [6], [7], [8], [9], [10]. WAL, as the name suggests, requires that the log entries be made persistent before data values can persist in memory. It can be implemented by using either a redo log or an undo log. When the system crashes in the middle of an atomic update, the atomic update can either be reapplied (for a redo log) or undone (for an undo log). We consider an undo log based WAL implementation as it enables in-place data writes, so the program can read the latest value without any redirection. In a redo log based implementation, data writes happen in the log area and read requests need to be redirected to the redo log for the latest value. Alternatively, if in-place writes are allowed, cache overflows need to be stored in a victim cache [14].

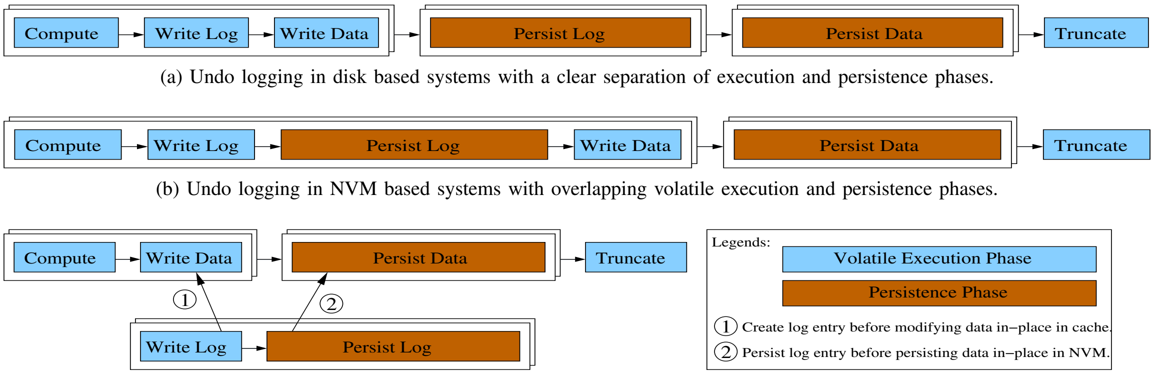
B. Traditional Undo Logging

Traditional systems with volatile main memory and persistent secondary storage typically follow the sequence of actions shown in Figure 1(a) for implementing atomic durability through an undo log. An atomically durable update using WAL is divided into two phases. The first phase is *volatile execution*: for each data item that is part of the atomic update, new values are computed in the compute stage, an undo log entry is written to in the Write Log stage and the data locations are updated in-place in the Write Data stage. The second phase is *persistence*: first the entire log is made durable in the Persist Log stage and then all the data updates are made durable in the Persist Data stage. After updating data, the log is truncated. Note here the clear separation of the volatile and persistence phases, which is justified due to two reasons. First, secondary storage is many orders of magnitude slower than memory and hence making any data durable incurs high latency. Second, secondary storage devices like disks are block based devices, so any update will write an entire page or block to secondary storage. Thus, traditional systems have a separate persistence phase to amortize the cost of performing the atomic update. Moreover, the boundary between volatile memory and persistent storage is software controlled: no data can persist without software's knowledge; this enables the separation.

C. Undo Logging with NVM

In contrast, in systems with non-volatile memory (NVM), the boundary between volatile caches and non-volatile memory is hardware controlled. Cache line replacements can move data from volatile caches to NVM without software's knowledge. Therefore, such systems cannot completely separate volatile execution from persistence. Moreover, NVM has very different properties than secondary storage. It is expected to have close to DRAM speed, while allowing for updates at a much finer (cache line) granularity. Because of the relatively lower cost (low latency and fine granularity) of persistence there is little need for amortization. Therefore it is not necessary to separate volatile execution from persistence. In fact, it is important to begin persisting data as soon as it is modified to avoid being limited by the write bandwidth, which can happen if all data is simultaneously flushed to persist at the end of the update.

Since we cannot and need not decouple volatile execution from persistence, let us examine the challenges (or constraints) for an undo log implementation in systems with



(a) Undo logging in disk based systems with a clear separation of execution and persistence phases.

(b) Undo logging in NVM based systems with overlapping volatile execution and persistence phases.

(c) Undo logging in NVM based systems with decoupling of execution and logging by enforcing two invariants.

Figure 1: Sequence of actions to be performed for undo logging in various scenarios.

NVMs. Undo log WAL implementation requires that the system maintain a persistent copy of the old version of all data items that are part of the atomic update at all times during the update. Hence, the in-place version of data cannot be modified until the undo log entry of the corresponding data has been made durable. Therefore, it is necessary to persist undo log entries before modifying data structures in-place. Figure 1(b) shows the sequence of actions performed for an undo log WAL implementation in NVM. The update process is split into two phases. In the first phase, there is an interaction between volatile execution and persistence operations; the compute stage takes place and an undo log entry is written to; then, the undo log entry is persisted and data is modified in-place. In the second phase the data updates are persisted and finally the log is truncated.

The bottleneck in this approach is that undo log entries have to be made durable in the critical path of execution. Our goal is to decouple log management from volatile execution and move the operation of persisting log entries out of the critical path of execution. As shown in Figure 1(c), writing an undo log entry and persisting log entries can be safely moved out of the critical path only if the following two invariants are satisfied.

Invariant 1. *A store should not complete until an undo log entry is created for the data being modified by the store.*

Invariant 2. *In-place data should not be made durable until the corresponding log entry is made durable.*

Invariant 1 ensures that an undo log entry exists for every data that is being modified as part of an atomically durable update. *Invariant 2* ensures that if the atomically durable update fails, undo log entries for all the data items updated in-place are durable. These log entries can be used to undo the partial changes of the failed update.

III. ATOM Design

In this section we introduce the conceptual design for ATOM, a hardware log manager for undo logging. We begin the section by first introducing the programming model with and without ATOM and then go on to establish a baseline design for an undo log manager in hardware. We then propose two optimizations: (i) to eliminate log persist operations out of the the critical path, and (ii) to minimize data movement.

```
While (!Done) {
    Write Undo Log
    Flush Log
    Write Data
}
Flush Modified Data
```

(a) Traditional Model

```
Atomic_Begin
While (!Done) {
    Write Data
}
Flush Modified Data
Atomic_End
```

(b) Proposed Model

Figure 2: Undo Log Programming Model

A. Programming Model

A typical approach towards atomic durability in software using an undo log is shown in Figure 2(a). An undo log entry is created and flushed before writing to data in-place. After completing the update, the modified data is flushed to NVM to complete the atomic update. In ATOM, we introduce two primitives, `Atomic_Begin` and `Atomic_End` to demarcate the start and end of the code segment performing an atomic update. Using these two primitives, the programmer does not have to create and flush undo log entries, but only write data in-place and flush data on completion of the update (Figure 2(b)). ATOM, the hardware log manager, will create undo log entries and flush them to memory before the in-place data modifications are written to memory.

The `Atomic_Begin` and `Atomic_End` construct only guarantees atomic durability and not isolation in a multi-threaded context. We require software to provide isolation. Specifically, following Chakrabarti et al. [5], we require the durable regions to coincide with outermost critical sections.

B. Baseline Design

The purpose of ATOM is to provide atomic durability for updates in NVM. Recall that to provide atomic durability, an undo log manager has to perform two tasks. First, creating a log entry consisting of: the old value of the data being modified, and its address. Second, ensuring that the log entry persists before the corresponding data is persisted. For the purposes of our discussion, we consider a generic chip multi-processor with private L1 caches, a multi-banked shared L2 cache and multiple memory controllers. ATOM is implemented as a distributed log manager, that is distributed across L1 caches and memory controllers – with the former responsible for creating log entries and the latter responsible for enforcing log → data ordering constraint. Finally, the OS reserves log space behind each memory controller for ATOM to write log entries into.

Creating a log entry. A log entry (old value, address pair) has to be created before modifying any data in an atomic update. Hence we use a store operation, belonging to an atomic update, to trigger the creation of log entries. We propose that the log manager in L1 cache couple the creation of a log entry with the processing of a write request from a store operation. Specifically, when the L1 cache controller receives a write request for a cache line, it first sends a log entry to the memory controller by piggy backing on the cache write-back interface. This ensures that the log entry is created before completing the write request, satisfying *Invariant 1*. The memory controller then writes the log entry into the log area in the NVM.

Note that while a cache line can get modified multiple times during an atomic update, it does not have to be logged every time it gets modified. Since an undo log stores the old value, it is sufficient to log a cache line only once: on the first write. To detect the first write to a cache line, we augment all the cache blocks with an additional *log* bit. The log bit is set when a cache line is written to for the first time during an atomic update. It is cleared when the modified value of the cache line is durably written to memory. This mechanism is similar to the log mechanism employed in LogTM [15]. The critical difference is that the log write in this case is not cached but has to be written to NVM. The log bit is only maintained during the lifetime of a cache line in the cache. As soon as a cache line is replaced, information about whether that cache line is logged or not is lost. Therefore, after being flushed to memory, when a cache line is modified again in the same atomic update the log bit is not set and the log manager logs it again. However, this is not a problem for ensuring correct recovery. During recovery the roll backs are applied in the order of newest first. This ensures that, at the end of recovery, the value of a cache line is restored to the one before the atomic update started.

Enforcing log → data ordering. The next task is to enforce ordering between log writes and in-place data writes. Therefore, upon detecting the necessity to log a cache line, the log manager first durably writes a log entry to the log area in NVM. After completing the write, it updates the value of the cache line in the cache and retires the store from the store queue (SQ). This ensures that an in-place data write cannot become durable before the corresponding log entry, thus satisfying *Invariant 2*. Figure 3(a) shows the sequence of operations. The log manager in the cache controller, upon receiving a write request from the SQ, checks if the log bit is set for the cache line (A) being updated. If the log bit is not set, the log manager creates a log entry ($CL(A)$) and sends it to the memory controller. The memory controller issues a write request for the log entry ($WL(A)$). After durably writing the log entry to NVM ($PL(A)$), the memory controller sends an acknowledgement ($Ack(A)$) back to the log manager in the cache controller. The log manager then completes the write request by modifying data in-place in the cache ($WI(A)$), which allows the store to be retired from the SQ. Under this baseline design, durably writing the undo log entry is in the critical path of completing the corresponding store operation from the SQ.

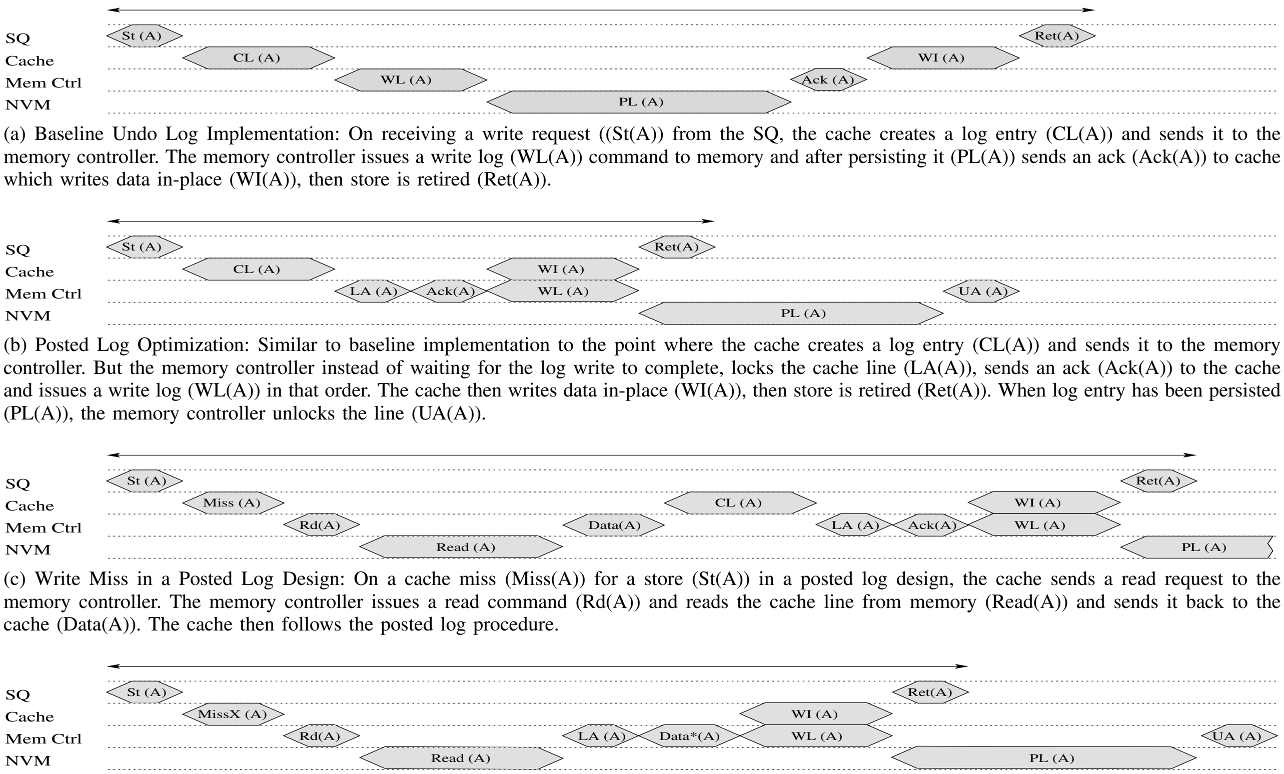
Sources of reordering. The log manager cannot allow the update of data in the cache until it receives an acknowledgement that the log entry has been made durable. This is because the cache line containing the modified data can be replaced at any time from the cache and could possibly overtake the log entry to NVM, which in turn will violate *Invariant 2*. This overtaking can happen because of the possible reordering in either the on-chip network (between the cache and NVM) or at the memory controller. Reordering is possible even if we consider the network and the memory controllers to be strictly ordered. It can arise if the log area and the data (cache line) are mapped to *different* memory controllers.

Logging cost. Store operations are typically not in the critical path in modern processors because they employ a queue to buffer store operations. But durably writing the undo log to memory is in the critical path of store operations. This reduces the rate at which store operations are completed from the SQ, which leads to a back pressure that can fill up the SQ and eventually stall the processor pipeline. Thus, it is important to reduce the critical path of store operations.

C. Posted Log Optimization

Currently, for each store, the critical path includes writing the log durably to memory as shown in Figure 3(a). To minimize the performance overhead of enforcing the log → data ordering constraint, we propose to allow the log manager in the cache controller to perform *posted log writes* to the memory controller, where the log manager enforces log → data ordering at the memory controller level. By doing so, we move the performance overhead of durably writing log entry to NVM, out of the critical path.

Figure 3(b) shows the sequence of operations for logging with a posted write feature. Upon receiving a write request from the SQ, the log manager in the cache controller sends a log entry to the memory controller. The memory controller locks the cache line (LA(A)) for which the log entry is being persisted and then sends an acknowledgement back to the



(a) Baseline Undo Log Implementation: On receiving a write request ((St(A)) from the SQ, the cache creates a log entry (CL(A)) and sends it to the memory controller. The memory controller issues a write log (WL(A)) command to memory and after persisting it (PL(A)) sends an ack (Ack(A)) to cache which writes data in-place (WI(A)), then store is retired (Ret(A)).

(b) Posted Log Optimization: Similar to baseline implementation to the point where the cache creates a log entry (CL(A)) and sends it to the memory controller. But the memory controller instead of waiting for the log write to complete, locks the cache line (LA(A)), sends an ack (Ack(A)) to the cache and issues a write log (WL(A)) in that order. The cache then writes data in-place (WI(A)), then store is retired (Ret(A)). When log entry has been persisted (PL(A)), the memory controller unlocks the line (UA(A)).

(c) Write Miss in a Posted Log Design: On a cache miss (Miss(A)) for a store (St(A)) in a posted log design, the cache sends a read request to the memory controller. The memory controller issues a read command (Rd(A)) and reads the cache line from memory (Read(A)) and sends it back to the cache (Data(A)). The cache then follows the posted log procedure.

(d) Source Logging Optimization: After reading the cache line (Read(A)) on a store miss (MissX (A)), the memory controller locks the cache line (LA(A)). It then sends the cache line back to the cache (Data*(A)) with log bit set, so the cache does not send a log write request. The cache then writes data in-place (WI(A)), then store is retired (Ret(A)). The memory controller, meanwhile issues a write log (WL(A)) request and unlocks the cache line (UA(A)) after persisting the log entry (PL(A)).

Figure 3: Sequence of actions of store queue (SQ), cache, memory controller (Mem Ctrl) and non-volatile memory (NVM) for undo logging in NVM based systems.

cache controller. Upon receiving the acknowledgement, the cache controller completes the write request, allowing the store to retire from the SQ (without having to wait for the log write to persist). When the log write eventually completes, the log manager in the memory controller unlocks the cache line (UA(A)). Whenever a write entry is ready to be scheduled out of the memory controller, the log manager is first consulted; only if the cache line is not locked, the write is allowed to go to NVM. In effect, this is a simple and efficient approach to enforcing the log → data ordering at the memory controller.

The posted log optimization cannot be applied if the log and data are mapped to different memory controllers. It can be challenging to ensure log-data co-location in software because an application program might be modifying data scattered behind multiple memory controllers. But because we perform logging in hardware, we are able to ensure that the log entry is sent to the same memory controller as the corresponding data (§IV-B). Thus, by co-locating log and data behind the same memory controller, we can enable the posted log optimization. With posted log optimization even though a store completes before durably writing the log entry to NVM, log → data ordering is enforced by the memory controller and hence *Invariant 2* is satisfied.

D. Source Log Optimization

Performing a posted write to the memory controller still incurs the cost of writing to and receiving an acknowledgement from the memory controller in the critical path of the store operation. But this can be further optimized in certain scenarios. Consider the scenario shown in Figure 3(c). The cache controller receives a write request for a cache line (A). It misses in the cache (Miss(A)), so the cache controller sends a fetch request to the memory controller. When the memory controller responds with the data (Data(A)), the cache controller checks for the log bit, which in this instance is not going to be set since the cache line has just been read from NVM. So the cache sends a log entry for cache line A to the memory controller. In a posted log design, the memory controller locks the cache line and responds with an acknowledgement, which completes the write request enabling the SQ to retire the store.

In the above example, however, there is unnecessary data movement from the cache controller to the memory controller in performing the log write. If a cache line is not present in the cache, then the in-place data in NVM is actually the old value of the cache line that needs to be written to the undo log. So the data that the cache controller sent back along with the undo log request is actually the same data that it just received from the memory controller because of its fetch request. This data movement from the cache controller to the memory controller can be avoided if the memory controller itself can write the old value of the cache line in the log area. We call this optimization *source log optimization* and is shown in Figure 3(d). The cache controller on detecting a miss on a write request (MissX (A)), sends a fetch exclusive request to the memory controller. The memory controller follows the posted log procedure after reading the cache line from NVM (Read(A)). It first locks the cache line (LA(A)), and then sends a data response to the cache with the log bit set (Data*(A)). On receiving data with the log bit set, the cache controller completes the write to the cache line. The memory controller, after sending the data to the cache, writes the log entry to NVM and eventually unlocks the cache line (UA(A)) on completion of the log write (PL(A)). Thus, this technique completely removes logging out of the critical path for stores that miss in the cache. It also eliminates redundant data movement.

IV. ATOM Architecture

In this section, we present the architectural and implementation details of ATOM.

A. Overview

The primary functions of ATOM are initiating log writes, managing log space (log allocation and clearing) and enforcing the log → data ordering constraint. These functions are implemented across two modules. The *log write initiate* module (LogI) and the *log manage* (LogM) module. The LogI module is embedded in the L1 cache controller as shown in Figure 4(a) and is responsible for initiating log write requests. The LogM module is embedded in the memory controller as shown in Figure 4(a) and is responsible for managing log space and enforcing the log → data ordering constraint. ATOM supports one atomic update per core. But it allows for concurrent execution of atomic updates across different cores by creating multiple (one per core) instances of the tracking structures in the LogM module.

B. Log Write Initiate (LogI) module

As discussed in §III-A we extend the processor-to-memory system interface to include two new commands, `Atomic_Begin` and `Atomic_End`. These commands signify the start and the end of an atomic update respectively. The memory system, upon seeing the `Atomic_Begin` command, will start logging for the cache lines being modified by subsequent stores. It will stop logging upon receiving an `Atomic_End` command. We handle nested atomic sections by flattening them.

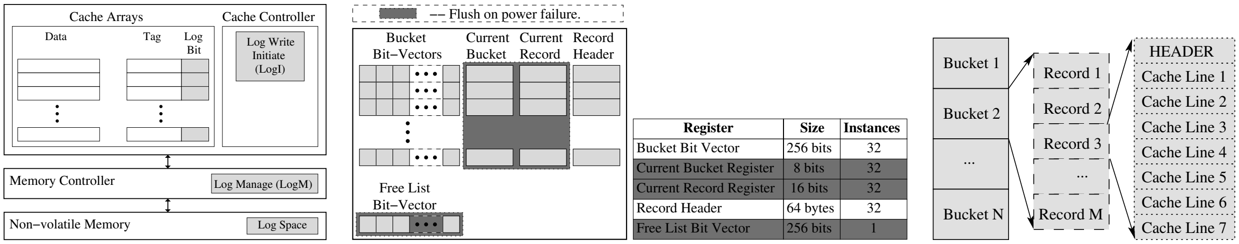
The LogI module looks at the log bit of each cache line before completing a write request. If the bit is set, the write request is immediately serviced. Otherwise the write request is stalled, a miss status handling register (MSHR) is allocated and a log write request is initiated to the memory controller associated with the corresponding cache line. The memory controller associated with the cache line is easily determined from the cache line address. By sending the log request to the same memory controller as the data, we ensure log-data co-location.

C. Log Manage (LogM) module

ATOM's LogM module manages a central log space which is shared across all threads and is statically allocated by the OS. ATOM manages this log space in terms of records and buckets as is described next.

Log Record Organization. We consider a system with 64 byte cache lines and ATOM performs logging at a cache line granularity. Therefore, each log entry consists of a cache line as data and address as meta-data. The simplest way to organize logging is to allow all threads to create individual log entries in the central log space. Writing a log entry to NVM in this way would require 2 write requests to memory since the size of a log entry is greater than a cache line. To minimize the overhead of multiple write requests we propose log entry collation (LEC), in which multiple log entries are collated into a single log record. The size of each log record is 512 bytes (or 8 cache lines). A log record can contain up to 7 log entries, and is divided into data (7 cache lines) and header (1 cache line) as shown in Figure 4(c). The header contains the addresses of all the 7 cache lines, the number of cache lines logged in the current record, and some reserved bits. On receiving a log write request, only the data field is written to memory at first. The meta-data for the log entry (consisting of its address) is added to the *record header*. A log entry is not considered durable until its corresponding record header persists. After logging 7 cache-lines, the header is written to memory, thus persisting the entire log record. When all 7 log entries in a log record are occupied, LEC reduces the overhead of writing a log entry: from 2 write requests for 1 log entry to 8 write requests for 7 log entries, which is a 57% reduction in the number of write requests to memory for logging.

LEC can lead to a violation of Invariant 2 if the cache line containing an in-place update is replaced from the cache and is made durable before the log header corresponding to its log entry. To avoid this, and before writing any data cache line to NVM, the memory address is compared to the addresses in the record header. The data cache line is written to memory only if there is no match in the header. If there is a match, then the header is first made durable to



Register	Size	Instances
Bucket Bit Vector	256 bits	32
Current Bucket Register	8 bits	32
Current Record Register	16 bits	32
Record Header	64 bytes	32
Free List Bit Vector	256 bits	1

(a) ATOM Hardware Extensions (b) Log Manage Module Components and Size (c) Log Organization

Figure 4: ATOM Components

complete the log write and then the data write is allowed to persist in NVM. Adding the address of a cache line in the record header corresponds to the concept of locking the cache line described in §III-C. The record header is cleared after persisting it in NVM, which corresponds to unlocking the cache lines.

The centralized shared log space can potentially be managed at a log record granularity. The log manager can maintain a log record head pointer and keep adding new log records to the central log space based on the requests received from different atomic updates. There are two ways to clear such a log. In the first approach, on completion of an atomic update, the log manager can read the log space starting from the beginning of log and clear all records belonging to the corresponding atomic update individually until the corresponding commit record is encountered. Unfortunately this will generate additional memory read requests to read log record headers sequentially and additional memory write requests to clear records corresponding to the completed update. Moreover, this will leave the log space fragmented. In the second approach the log manager – instead of clearing log records on completion of atomic updates – can wait for the completion of all concurrent atomic updates and then clear the log space. This method will avoid fragmentation, but will stall the processing of new atomic updates during the wait.

Log Bucket Organization. To overcome these limitations we propose dividing the shared log space into buckets of log records and managing log space allocation and deallocation at a bucket granularity, resulting in the organization shown in Figure 4(c). An atomic update has an associated bit-vector, known as *bucket bit vector*, indicating the buckets allocated to that update. Using a bit-vector alleviates the first problem of requiring additional memory read and write requests to allocate or clear the log as it can be used to identify free buckets and to clear allocated buckets. Along with the bit vector, there is a *current bucket* register that identifies the bucket to which log records are being added currently; a *current record* register that indicates the record in the current bucket being written to; and, finally, a *record header* register that stores the meta-data for the log record currently being updated as shown in Figure 4(b). A new bucket is allocated from the *free list bit vector*, which is generated by NORing all the bucket bit vectors.

The bucket bit vector and current bucket, current record and record header registers – together track a single atomic update and are collectively known as an *atomic update structure* (AUS). So to support concurrent atomic updates, these need to be replicated as shown in Figure 4(b). We support up to 32 concurrent updates in our system (1 per core). The sizes of all the registers is shown in Figure 4(b). The space overhead of LogM module amounts to 3.125 KB.

The bucket organization, by allocating log buckets from a central pool, allows for dynamic sharing of log space by concurrent atomic updates. It also simplifies log clearing on completion of an atomic update. LogM does not have to read the log space, but only has to clear the bit vector corresponding to the atomic update and update the free list bit vector. This is a single cycle operation and will be completed even if a power failure occurs at the moment of clearing the log.

D. Recovery

After a power failure, the incomplete atomic updates need to be undone to restore the system to a consistent state. The enforcement of *Invariant 2* guarantees that at any point of time during execution, if a log entry has not persisted then the corresponding data would not have persisted either. Hence in the event of a power failure, all the pending log writes in the memory controller store buffers can be safely discarded. Only those log entries that have already persisted need to be considered during recovery. However, on a power failure the information about valid log buckets in the memory controller will be lost. To correctly access the log space we need to be able to identify which buckets are valid (contain valid log records). This can be identified by taking a complement of the *free list bit vector*. Also, some of the valid buckets might be partially filled because log entries were being added to them when the power failure occurred. These partially filled buckets can be identified from the *current bucket register*. And finally the number of valid log records in those partially filled buckets can be identified from the *current record register*. The total size of the above 3 critical structures is 128 bytes. To ensure

Cores	32 OoO cores @ 2GHz
ROB Size	192 Entry
Store Queue	32 Entry
L1 I/D Cache	32KB 64B lines, 4−way
L1 Access Latency	3 cycles
L2 Cache	1MB×32 tiles, 64B lines, 16−way
L2 Access Latency	30 cycles
MSHRs	32
Memory Controllers	4
NVM Access Latency	360 (240) cycles write (read)
On-chip network	2D Mesh, 4 rows, 16B flits

Table I: System Parameters

that these critical structures are preserved, we utilize a feature similar to Asynchronous DRAM Refresh (ADR)[16] supported by Intel. ADR ensures that on a power failure, all the memory controller buffers (24 or more cache lines) are flushed to memory. In our implementation, only the critical structures (amounting to only 2 cache lines) need to be written to NVM on detecting a power failure.

Recovery after a power failure is accomplished in software through a generic recovery routine provided as a system call which relieves the programmer from having to implement custom recovery schemes. The recovery routine will read the bucket bit vectors and current bucket and record information from NVM and reconstruct the state of the log space at the time of the crash. It will then perform undo operations in the reverse order starting from the last log record to the first one for each incomplete atomic update. The recovery routine performs undo operations for all the cache lines recorded in the log even though some of the cache lines may not have been updated in memory at the time of crash. This might impose a performance overhead during recovery but does not affect the correctness.

E. Log Allocation and Overflow

In our design, a central log space is allocated by the operating system (OS) which is shared between concurrent atomic updates. The OS is aware of the number of physical pages associated with each memory controller. It reserves a proportional number of these pages as the log area. The OS then ensures that no virtual page is mapped to any of these reserved log pages. Recall that the LogI module ensures that each log entry is correctly directed to the memory controller where the corresponding data page resides.

There can be two kinds of overflows in the system. The first type of overflow, known as structural overflow, occurs when the number of concurrent update requests are higher than the number of updates supported by the hardware. An `Atomic_Begin` instruction checks for the availability of an AUS. If an AUS is not available it will stall. Eventually as other atomic updates complete (execute `Atomic_End` instruction), an AUS will free up and will be allocated to the stalled update. The waiting update does not have any resources reserved and hence cannot block any other update. Thus, a structural overflow cannot result in a deadlock.

Hash	Insert/delete entries in a hash table
Queue	Insert/delete entries in a queue
RBTree	Insert/delete nodes in a red-black tree
BTree	Insert/delete nodes in a b-tree
SDG	Insert/delete edges in a scalable graph
SPS	Random swaps between entries in an array

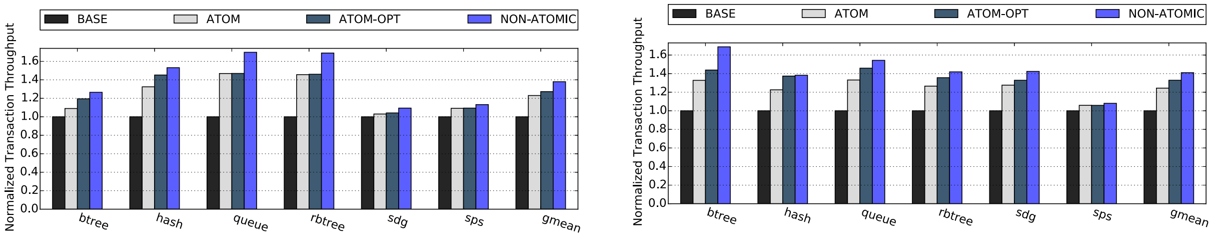
Table II: Micro-benchmarks used in our experiments

The second type of overflow, known as log overflow, occurs when a new bucket needs to be allocated behind a memory controller, but no more buckets are available in the corresponding free list bit vector. In other words, all of the reserved log pages in the memory controller have been exhausted. In this scenario, the OS is interrupted to allocate additional log pages for that memory controller, which will be used to store subsequent log records. Because this additional resource (log space) is allocated to the requesting update, it will make forward progress and not block any other update. Hence, a log overflow will also not result in a deadlock. Moreover, dynamically sharing the log space between atomic updates reduces the probability of log overflow as opposed to a design where the log space is statically partitioned.

V. Experimental Setup

We now describe our simulation infrastructure, system configuration, benchmarks and designs that we evaluate. We implemented ATOM on gem5 [17] with Ruby in full system simulation mode. The on-chip interconnect is modelled using Garnet [18]. We extend the Ruby memory model to implement the proposed log manager. We evaluate ATOM on a 32-core multicore (1 thread per core) with multi-banked LLC and 4 memory controllers placed on the corners of the die. We consider a MESI based coherence protocol for our evaluation. Table I shows the main parameters of the system. The memory write latency that we consider is 10× that of typical DRAM latency. We assume a single memory channel per memory controller unless otherwise stated. The peak memory bandwidth in our setup is 5.3 GB/s per memory channel. We model an address match latency of 1 cycle in the memory controller to check if the data write request has a corresponding log entry pending in the record header.

Workloads. We use the micro-benchmarks listed in Table II to evaluate ATOM and the proposed optimizations. These micro-benchmarks implement data structures that are similar to those in the benchmark suite used by NVHeaps [7], except for the queue micro-benchmark, which is similar to the copy-while-locked queue of [19]. We evaluate these workloads with two data set sizes (table entries, tree nodes, queue entries etc.): small (512 bytes) and large (4 KB). Each benchmark performs search and atomic insert and delete operations on the corresponding data structure.



(a) Small

(b) Large

Figure 5: Transaction throughput normalized to BASE for micro-benchmarks.

We also evaluate ATOM using the TPC-C benchmark where the TPC-C schema is implemented using B$^+$-Trees [6]. We use a scaling factor of 1 and use 32 threads to simulate the 32 terminals issuing new order transactions. Our goal is to measure the overhead in write-intensive operations. Therefore, the new order transaction is the best choice as it is the most write-intensive TPC-C transaction. We slightly modified the benchmark and removed the wait times (implemented using *sleep* system call) to allow us to execute the benchmark in a reasonable amount of time.

Designs: We compare the following designs:

- **BASE**: The baseline hardware undo log which performs logging transparently in hardware (without additional instructions for logging), but the log write happens in the critical path of a store operation. (§III-B).
- **ATOM**: Proposed design with posted log optimization (§III-C).
- **ATOM-OPT**: The above with source log optimization as well (§III-D).
- **NON-ATOMIC**: No logging operations are performed, and hence this design represents upper bound on performance for a logging implementation. On completion of each atomic update, all the data modified within the atomic update is still written back to NVM.
- **REDO**: The redo log design of Doshi et al. [14] with a couple of modifications (that actually benefit their design). First, although their implementation requires additional log write instructions in software, we do this in hardware by allowing the cache to issue log writes on receiving a store, for the sake of fair comparison. Second, we consider an infinite size victim cache. Similarly to their design, we implement write combining for log writes.

VI. Results

We first present the speed-up due to ATOM and analyze the impact of both posted logging and source logging optimizations. We then show how ATOM reduces the critical path of logging operations by looking at the occupancy of the store queue and also analyze the reasons behind the magnitude of performance improvement due to source logging. We also compare ATOM with a REDO log based design [14] and perform a sensitivity study by varying memory latency. Finally we present the performance of ATOM for the TPC-C benchmark.

A. Transaction Throughput

Figure 5(a) shows transaction throughput for the ATOM, ATOM-OPT and NON-ATOMIC designs, normalized to BASE for small dataset sizes. On average, ATOM improves transaction throughput by 23%. Recall that the posted log optimization reduces the critical path of store operations by enforcing log → data ordering at the memory controller. ATOM-OPT improves the throughput by 27% on average over BASE which is a 4% improvement over ATOM. Recall that source logging optimization further reduces the critical path of stores that miss in the cache by eliminating the log write request from cache to memory controller. The improvement because of this optimization will depend on the percentage of log writes that are source logged. We analyze this further in §VI-C.

The NON-ATOMIC design has a 38% higher throughput than BASE. The ATOM-OPT design, by improving the throughput by 27%, is able to close about 71% of the performance gap between BASE and the optimal (NON-ATOMIC) design.

Figure 5(b) shows the normalized transaction throughput for large dataset sizes. On average, ATOM improves the transaction throughput by 24%, while ATOM-OPT improves it by 33% over BASE which is a 9% improvement over ATOM. For large dataset sizes NON-ATOMIC design improves the throughput over BASE by 41% and ATOM-OPT design is able to close 83% of this performance gap between NON-ATOMIC and BASE designs.

B. Impact on Critical Path

Store operations are not typically in the critical path of program execution because most processors employ store queues (SQ) to complete stores out of the critical path. But with logging, writing to NVM is in the critical path of completing store operations from the SQ. This creates a back pressure, which eventually fills up the SQ and stalls the processor pipeline. Figure 6 shows the number of SQ-full events for ATOM-OPT and NON-ATOMIC designs, normalized to BASE for benchmarks with small dataset

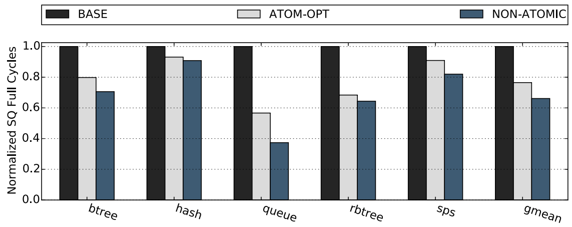


Figure 6: SQ full cycles normalized to BASE for microbenchmarks with small dataset size.

	btree	hash	queue	rbtree	sdg	sps
small	0.12	0.12	0.07	0.01	0.04	0.01
large	0.4	0.4	0.7	0.4	0.07	0.01

Table III: % of source logged cache lines for ATOM-OPT

sizes. ATOM-OPT reduces the SQ-full cycles by 21% on average which correlates with the increase in throughout. Benchmarks with high reduction, like *queue* (43%) and *rbtree* (35%) also show high improvement in throughput: 47% and 46% respectively. Similarly, *sps* which has the minimum reduction (1%) shows the minimum improvement in throughput (4%). On average ATOM-OPT has only 10% more SQ-full cycles than NON-ATOMIC.

Benchmarks with large dataset sizes show a similar trend but we do not show them because of space constraints. In these benchmarks the average reduction in the number of SQ-full cycles drops to 11% from the high 21% seen for benchmarks with small dataset sizes. With increasing dataset sizes, the number of cache lines to be written back at the end of an atomic update increases. This places additional pressure on the SQ occupancy and hence the scope for reducing SQ-full cycles decreases.

C. *Source Logging*

The source logging optimization removes log writes from the critical path for store operations that miss in the cache. ATOM-OPT logs the cache lines for which a fetch exclusive request is received by the memory controller during an atomic update. Table III shows the percentage of source logging for benchmarks with small and large datasets. We see that even with as little as 0.12% of log writes being source logged, ATOM-OPT provides a transaction throughput improvement of about 10% and 13% over ATOM for *btree* and *hash* benchmarks respectively for small datasets.

As the dataset size grows, the percentage of store operations missing in the cache increases. We see that *queue*, which has the highest percentage (0.7%) of source logging, provides the highest throughput improvement (16%) for ATOM-OPT over ATOM. Moreover, *sps*, which has the lowest percentage of source logged cache lines for both small and large datasets does not show any improvement compared to ATOM.

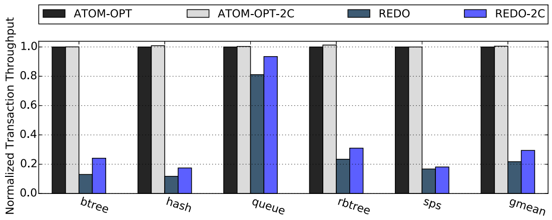


Figure 7: Transaction throughput for REDO and ATOM-OPT designs normalized to ATOM-OPT for benchmarks with small dataset size.

D. *Comparison with Redo Log*

We compare ATOM-OPT with the recently proposed REDO log design [14]. In addition to the setup of §V, we also evaluate these designs in a configuration with two memory channels at each memory controller (*-2C), where one channel is used for data while the other channel is used for logging, in order to mimic the configuration used by the authors in [14]. Figure 7 shows the transaction throughput normalized to ATOM-OPT. In the single channel configuration REDO is only able to achieve 22% of the transaction throughput of ATOM-OPT while in the two channel configuration it is able to achieve 30%. We identified that the disparity in performance between ATOM-OPT and REDO is because of the difference in their memory bandwidth requirements.

REDO generates 19× more log entries than ATOM-OPT. This is because in REDO, every store operation in an atomic section generates a log entry. Whereas in ATOM, a log entry is generated only on the first write to a cache line. These log entries increase the pressure on the memory write bandwidth. Moreover, in REDO the log entries have to be read from memory to perform in-place data updates. These log read requests interfere with the critical data read requests from the cores, thus slowing down execution. In the two channel configuration, the log and data reads go to separate channels and hence the log read requests do not interfere with the critical data read requests. Therefore, the throughput of REDO-2C increases by 9% over REDO.

E. *Sensitivity to Memory Latency*

Figure 8 shows the transaction throughput for the *rbtree* benchmark with small dataset size for varying memory latencies (as a ratio of DRAM latency). At NVM latencies similar to DRAM, REDO provides higher transaction throughput than ATOM-OPT because of two reasons. First, the low latency memory is quickly able to absorb the large number of log writes generated by REDO. So this is no more a bottleneck for REDO. Second, REDO performs in-place updates of data in the background, whereas ATOM-OPT has to persist all the in-place modifications to NVM at the end of each atomic update. But on increasing the latency, the performance of REDO degrades super-linearly because

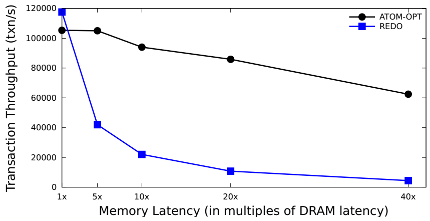


Figure 8: Transaction throughput variation (ATOM-OPT vs REDO) with varying memory latency.

	BASE	ATOM	ATOM-OPT	REDO
Throughput	1	1.58	1.6	1.47

Table IV: TPC-C throughput normalized to BASE.

of the relatively high memory bandwidth requirement. The throughput of ATOM-OPT degrades almost linearly because its memory bandwidth requirement is lower than REDO.

F. TPC-C

As a case study we evaluate ATOM using TPC-C, annotating all the critical sections as atomic regions. Table IV shows the throughput for TPC-C normalized to BASE. ATOM provides a throughput improvement of 58% over BASE whereas ATOM-OPT provides an improvement of 60% over BASE. ATOM-OPT provides negligible improvement over ATOM because only 0.02% of log operations were source logged. ATOM-OPT reduces the SQ-full cycles by 42%.

REDO on the other hand provides a throughput improvement of 47% over BASE (13% lesser improvement than ATOM-OPT). It is worth noting that both ATOM and REDO provide higher gains for TPC-C as opposed to micro-benchmarks. This is because TPC-C has relatively lower frequency of updates in comparison to micro-benchmarks, and hence memory bandwidth is less of a problem.

VII. Related Work

Non-volatile memory (NVM) technologies have been studied for various application scenarios, e.g., program checkpointing [20], [21], [22], databases [23], [24], [25], [26], in-memory persistent data structures [5], [6], [7], [8], [9], [21], [27] and file systems [10], [28]. All these scenarios require support for atomic durability, which can be implemented using either WAL or shadow paging.

Systems like Mnemosyne [8], REWIND [6] and Atlas [5] support atomic durability through write ahead logging implemented in software. Hence they rely on the `pcommit` instruction which enforces the log $\rightarrow$ data ordering in the critical path of execution. In [29], the authors propose a software approach to reducing ordering overhead for providing atomic durability, by reducing the number of copy operations and by persisting data in bulk. In [23], the authors propose a group commit mechanism to amortize the cost of persist ordering constraints within a transaction. All these techniques have to persist the log in the critical path.

Many hardware techniques have been proposed to avoid persisting the log in the critical path of execution. NVHeaps [7] relies on *epoch barriers* [19], [28] to persist the log out of the critical path. Implementing *epoch barriers*, however, requires significant changes to the cache hierarchy. Besides, their efficacy is limited (and hence performance sub-optimal) for smaller epoch sizes [21]. In a concurrent proposal [30], the authors propose delegated persist ordering that, similar to our posted log optimization, enforces ordering constraints at the memory controller. However, they only provide ordering but not atomic durability. In LOC [31], the authors provide hardware support for atomic durability through redo logging. Their proposal again requires extensive changes to the cache hierarchy along with support for multi-versioned caches.

Kiln [32] provides atomic durability in presence of a non-volatile cache (NVC). Having an NVC eliminates the requirement of logging by allowing NVC and NVM to hold two versions of a cache line where one of the versions can conceptually be considered as a log. Memory controller optimizations [33], [34], [35] have been proposed to improve the performance by differentiating between log writes and data writes. These proposals are broadly aimed at reducing latency of persist operations and are complementary to our proposal of removing log writes from the critical path.

Pelley et al. propose the concept of memory persistency models in terms of persist ordering constraints [19]. In [36] the authors analyze dependencies that need to be satisfied to implement transactions under various persistency models and propose optimizations to improve performance. These proposals broadly deal with reducing dependencies across transactions and are complimentary to our approach of reducing dependencies within a transaction.

Recently, redo logging for atomic durability was proposed in [14]. After completing an atomic update, the backend controller reads the log entries from the log area in memory and updates data in-place. Reading log entries after each update places additional pressure on the memory read bandwidth and can significantly delay the critical read requests coming from the processor. Another drawback is that it can lead to multiple log entries for the same data if the data gets modified multiple times during an atomic update (§VI-D). They also need a victim cache to avoid spilling dirty cache lines into memory.

NVM cannot be used as a drop-in replacement for disks without modifying the surrounding software stack [23], [25], [37]. In systems with NVM, the synchronization overheads of a centralized log are high and hence there have been proposals for using per-thread distributed logs [24], [25]. In ATOM, however, the log space is centralized and shared across all threads to reduce fragmentation and improve utilization. We overcome the synchronization overhead by partitioning the log space into buckets and managing log space at bucket granularity in hardware.

ATOM provides atomic durability and relies on software locks to provide isolation [5]. But it can be adapted to leverage other ways to provide isolation such as hardware transactional memory (including but not limited to Intel's Transactional Synchronization Extentions [12] and [15]).

VIII. Conclusion

We have presented ATOM: a hardware log manager that provides atomic durability in NVM via undo logging. We have described the salient principles behind ATOM's design and have shown how these principles can be implemented in hardware so that logging operations can be moved out of the critical path. We have evaluated ATOM on a variety of workloads, ranging from standard micro-benchmarks to large-scale database management scenarios. Our results show that ATOM delivers on its promise of high-performance logging at minimal overhead across the board.

Acknowledgment

We would like to thank the anonymous reviewers for their helpful comments. This work is supported by the Intel University Research Office and by EPSRC grants EP/M001202/1 and EP/M027317/1 to the University of Edinburgh.

References

[1] Intel Corporation and Micron. Intel and Micron Produce Breakthrough Memory Technology. http://newsroom.intel.com/community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-technology.

[2] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting Phase Change Memory As a Scalable Dram Alternative," in *ISCA 2009*.

[3] E. Kultursay, M. T. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating STT-RAM as an energy-efficient main memory alternative," in *ISPASS 2013*.

[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, no. 1, 1992.

[5] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," in *OOPSLA 2014*.

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

[7] 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 *ASPLOS 2011*.

[8] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *ASPLOS 2011*.

[9] X. Wu and A. L. N. Reddy, "Scmfs: A file system for storage class memory," in *SC 2011*.

[10] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *EuroSys 2014*.

[11] Intel Corporation. Persistent Memory Programming. http://pmem.io/.

[12] ——, *Intel® Architecture Instruction Set Extensions Programming Reference*.

[13] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger, "The Recovery Manager of the System R Database Manager," *ACM Comput. Surv.*, vol. 13, no. 2, 1981.

[14] K. Doshi, E. Giles, and P. Varman, "Atomic Persistence for SCM with a Non-intrusive Backend Controller," in *HPCA 2016*.

[15] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "LogTM: log-based transactional memory," in *HPCA 2006*.

[16] Intel Corporation. Platform brief Intel Xeon Processor C5500/C3500 Series and Intel 3420 Chipset. http://download.intel.com/design/intarch/prodbref/323306.pdf.

[17] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, 2011.

[18] N. Agarwal, T. Krishna, L.-S. Peh, and N. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *ISPASS 2009*.

[19] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *ISCA 2014*.

[20] X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, and Y. Xie, "Leveraging 3D PCRAM Technologies to Reduce Checkpoint Overhead for Future Exascale Systems," in *SC 2009*.

[21] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Efficient Persist Barriers for Multicores," in *MICRO 2015*.

[22] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "ThyNVM: Enabling Software-transparent Crash Consistency in Persistent Memory Systems," in *MICRO 2015*.

[23] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge, "Storage Management in the NVRAM Era," *Proceedings of VLDB Endowment*, vol. 7, no. 2, 2013.

[24] T. Wang and R. Johnson, "Scalable Logging Through Emerging Non-volatile Memory," *Proc. VLDB Endow.*, vol. 7, no. 10, 2014.

[25] J. Huang, K. Schwan, and M. K. Qureshi, "NVRAM-aware Logging in Transaction Systems," *Proc. VLDB Endow.*, vol. 8, no. 4, 2014.

[26] J. DeBrabant, J. Arulraj, A. Pavlo, M. Stonebraker, S. Zdonik, and S. Dulloor, "A prolegomenon on oltp database systems for non-volatile memory," in *ADMS@VLDB*, 2014.

[27] E. Giles, K. Doshi, and P. Varman, "Bridging the programming gap between persistent and volatile memory using WrAP," in *CF 2013*.

[28] 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 *SOSP 2009*.

[29] Y. Lu, J. Shu, and L. Sun, "Blurred Persistence: Efficient Transactions in Persistent Memory," *Trans. Storage*, vol. 12, no. 1, 2016.

[30] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated Persist Ordering," in *MICRO 2016*.

[31] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-ordering consistency for persistent memory," in *ICCD 2014*.

[32] 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 *MICRO 2013*.

[33] J. Zhao, O. Mutlu, and Y. Xie, "FIRM: Fair and High-Performance Memory Control for Persistent Memory Systems," in *MICRO 2014*.

[34] R.-S. Liu, D.-Y. Shen, C.-L. Yang, S.-C. Yu, and C.-Y. M. Wang, "NVM Duet: Unified Working Memory and Persistent Store Architecture," in *ASPLOS 2014*.

[35] L. Sun, Y. Lu, and J. Shu, "DP2: Reducing Transaction Overhead with Differential and Dual Persistency in Persistent Memory," in *CF 2015*.

[36] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, "High-Performance Transactions for Persistent Memories," in *ASPLOS 2016*.

[37] J. Meza, Y. Luo, S. Khan, J. Zhao, Y. Xie, and O. Mutlu, "A Case for Efficient Hardware-Software Cooperative Management of Storage and Memory," in *WEED 2013*.