Architectural Support for Atomic Durability in Non-Volatile Memory

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

I. Introduction

Byte-addressable non-volatile memory (NVM) 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. 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.

Write-ahead logging (WAL) [1] using an undo log is a commonly used mechanism to provide atomic durability [2], [3], [4]. This mechanism operates 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 the state can be recovered to either of the versions. An undo based WAL writes undo log entries for all data updates, and enforces the ordering constraint that all log entries become durable before any data update (log $\rightarrow$ 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* [5], [6].

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. 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 the fine grained log $\rightarrow$ data ordering constraint in the memory controller (out of the critical path).

We propose ATOM [7]: 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* instructions).

Undo Logging with NVM. In systems with NVM, the boundary between volatile caches and non-volatile memory is hardware controlled. Therefore, it is necessary to persist undo log entries before modifying data structures in-place to enforce log $\rightarrow$ data ordering. And doing so results in an overlap between volatile execution and persistence operations (Figure 1a). Persistence operations, which loosely translate into stores, are buffered in the store queues (SQ) employed by modern processors and are therefore typically out of the critical path. However, because of the high latency of writes to NVM these SQs eventually fill up and stall the processor pipeline.

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 1b, 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.

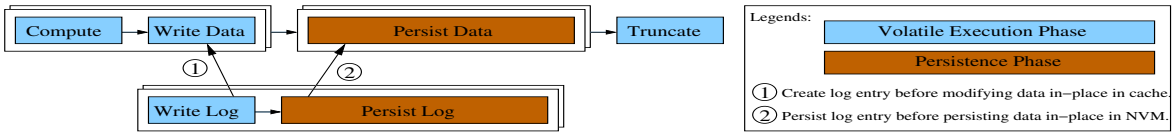
II. ATOM Design

We first 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.

A. Baseline Design

ATOM is implemented as a distributed log manager across L1 caches and memory controllers – with the former

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



(b) 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.

responsible for creating log entries and the latter responsible for enforcing log → data ordering constraint.

Creating a log entry. We use a store operation, belonging to an atomic update, to trigger the creation of log entries. 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.

Enforcing log → data ordering. After completing the log write to NVM, the log manager updates the value of the cache line in-place in the cache and retires the store from SQ. This ensures that an in-place data write cannot become durable before the corresponding log entry, thus satisfying *Invariant 2*.

Logging cost. 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.

B. Posted Log Optimization

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 and sends an acknowledgement to the cache controller before completing the log write to NVM. By doing so, we move the performance overhead of durably writing log entry to NVM, out of the critical path. 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.

C. 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 a scenario where the cache controller receives a write request for a cache line which misses in the cache. 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. In this case, the memory controller itself can write the old value of the cache line to the log area in NVM and respond to the cache controller with an indication that the cache line has already been logged. We call this optimization, which reduces wasteful data movement, as *source log optimization*.

III. Evaluation

We evaluated the performance benefits of ATOM using GEM5 simulator for a 32-core machine with a multi-banked last level cache and multiple memory controllers [7].

We compare ATOM with both the posted log (§II-B) and the source log (§II-C) optimizations to the baseline design (§II-A). For micro-benchmarks, ATOM provided 27% higher throughput on average over the baseline by reducing the SQ full cycles by 21%. Moreover, this throughput improvement of ATOM over the baseline is within 11% of that of an ideal design, which does not perform logging and flushing of log entries and therefore does not guarantee atomic durability. For TPC-C ATOM improves the throughput by 60% over the baseline design.

References

[1] 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.*, 1992.

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

[3] 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*.

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

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

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

[7] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "ATOM: Atomic Durability in Non-volatile Memory through Hardware Logging," in *HPCA*, 2017, http://homepages.inf.ed.ac.uk/s1372211/pub/hpca17.pdf.