Efficient Durability for Lock-based Code

Swapnil Haria, Pratyush Mahapatra, Mark D. Hill, Michael M. Swift
University of Wisconsin-Madison
{swapnilh,pratyush,markhill,swift}@cs.wisc.edu

1. Introduction

Hardware and software support for Persistent Memory (PM) typically assumes a transactional memory (TM) programming model. However, TM has not been widely adopted, and most parallel code uses locks. We expect this trend to continue due to the high overheads of TM and the difficulty in transforming existing lock-based code to use transactions [4].

Given the enduring popularity of lock-based code as well as the rapid emergence of PM, we seek to facilitate their use together. By storing key data structures in PM, applications can recover after a crash, reusing work done in the prior execution and make forward progress. To ensure consistent and useable state after crashes (e.g., no dangling pointers), updates to persistent data are made atomic via logging and have ordering constraints. Atlas [1], the software runtime we leverage, uses code interposition through an LLVM-based compiler pass to insert these logging and ordering constraints (Table 1).

Unfortunately, on existing x86-64 hardware, ordering is only supported through individual cacheline flushes (e.g., synchronous `clflush` and asynchronous `clwb`) and fences (e.g., `sfence`). Most flushes (log entries for PM stores, lock acquires/releases) must be synchronous to satisfy ordering constraints. For example, the undo log entry for a store must be persisted before the in-place update. Thus, cache flushes result in high durability overheads (up to 25X).

Unlike transactions, overlapping lock acquisitions require logging of synchronization history (SH) to identify happens-before relationships between critical sections (CS) executing on different threads [1]. The global SH has to be updated on every lock acquire/release operation. High contention for this shared data can significantly degrade performance.

In this work, we make three contributions. First, we show that the Hands-Off Persistence System (HOPS), originally designed for persistent transactions [3], also benefits durable lock-based code. HOPS minimizes synchronous flushing by decoupling ordering from durability. Second, we show that HOPS can be extended to HOPS+ with hardware dependency tracking to replace expensive software tracking of SH. Third, we developed and will release a realistic PM full-system simulator based on gem5. Our evaluation shows that HOPS+ improves the performance of durable lock-based code by 108% compared to existing x86-64 hardware and is 18% better than unmodified HOPS.

2. Background

Persistent lock-based code. While transactions generally require perfect isolation until commit, locks allow more flexible use such as hand-over-hand locking and releasing isolation early on contended structures. As a result, durable lock-based code may need to undo completed critical sections after a crash to restore consistency. As an example, Figure 1 shows CS2 executing on thread 2 after CS1-A completes on thread 1, which results in {*FLAG*=1, *DATA*=1}. If a crash occurs before CS1 completes, though, the effects of CS1 must be reverted, which includes CS1-A. This results in inconsistent persistent state {*FLAG*=1, *DATA*=0}. As a result, ordering information about dependent critical sections on other threads must be persisted so that recovery code can also undo completed CS2.

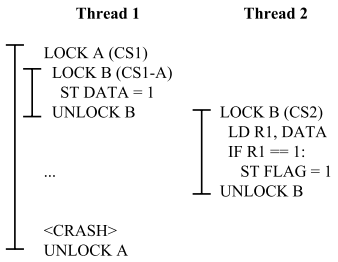

Figure 1: Issues with lock-based durable code. FLAG, DATA are stored in PM and initialized to 0.

Our approach leverages Atlas's durability semantics for lock-based code [1]. Atlas assumes the invariant that data structures are consistent outside of a CS and conversely may be inconsistent only within CS. Thus, synchronization operations, e.g., lock acquires/releases, are reliable indicators of consistent program points. These operations along with PM stores are logged to allow recovery to a consistent state.

HOPS design. HOPS provides ISA primitives for software to express durability and ordering constraints separately [3]. This follows from observations from WHISPER benchmarks that ordering events are 5-50X more common than durability events.

PM write-ordering is enforced by per-thread Persist Buffers (PBs, also in [2]) in hardware. The Ordering FENCE (`ofence`) primitive ensures a thread's stores following an `ofence` become durable only after all stores preceding the `ofence` become durable. These stores are buffered locally in the volatile PBs, and are made durable asynchronously. Each PM store is associated with a timestamp, and PBs ensure that stores are written back in timestamp order. Thus `ofence` is implemented simply by incrementing a per-thread timestamp. The Durability FENCE (`dfence`) triggers a flush of a thread's PB and waits for all updates to become durable.

HOPS also enforces ordering of durable updates across

Original	ATLAS, x86-64	ATLAS, HOPS	ATLAS, HOPS+
Lock L	Lock L; Log-Acq(L); CLFLUSH Log; Log-SH(L); CLFLUSH Log; Update SH(L);	Lock L; Log-Acq(L); OFENCE Log-SH(L); OFENCE; Update SH(L);	Lock L; Log-Acq(L); OFENCE;
Unlock L	SFENCE; Log-Rel(L); CLFLUSH Log; Unlock L;	OFENCE; Log-Rel(L); DFENCE; Unlock L;	OFENCE; Log-Rel(L); DFENCE; Unlock L;
ST A, val	Undo-Log(A); CLFLUSH Log; ST A, val; CLWB A;	Undo-Log(A); OFENCE; ST A, val;	Undo-Log(A); OFENCE; ST A, val;

Table 1: Software transformations for durability

threads. If thread 2 updates data logically after an update from thread 1, thread 2's update cannot be made durable before thread 1's update. HOPS modifies the coherence protocol to pass timestamps along with coherence permissions, so that thread 2's PB can record a cross-thread dependency on the preceding PM write by thread 1.

3. HOPS for lock-based code

We first demonstrate the value of original HOPS for lock-based code and then discuss extensions for higher performance.

HOPS primitives. First, we replace all synchronous flushing in Atlas with unmodified OFENCEs and one DFENCE per critical section. For instance, log updates are ordered before data updates using OFENCEs. Minimizing synchronous flushing greatly reduces the time spent in critical sections, thereby reducing lock contention and improves concurrency. Table 1 shows the code changes to Atlas for utilizing HOPS primitives.

Synchronization History. Second, we eliminate the need for maintaining synchronization history in software by using hardware dependency tracking in HOPS. One approach, which we evaluate here as HOPS+, is to run outermost critical sections in a serializable order using closed nesting, i.e., delaying the global visibility of updates from nested critical sections until the outermost critical sections complete. In HOPS+, closed nesting is achieved by having private caches retain exclusive permissions for modified cachelines until the outermost CS releases its lock.

Furthermore, updates from CSes on different threads are made durable in commit order, which ensures correctness in the presence of crashes. For this, HOPS+ uses the original HOPS mechanism of ordering all writebacks from PBs to PM and disallowing all cache writebacks to PM as these could have been reordered. However, the current approach can result in deadlocks for rare programs that rely on open nesting. For such programs, SH must be recorded by software. Besides HOPS+, HOPS enables other promising directions for optimizing durable lock-based code. Ongoing research involves support for open nesting and using hardware logging of SH on rare occasions when certain isolation guarantees are violated.

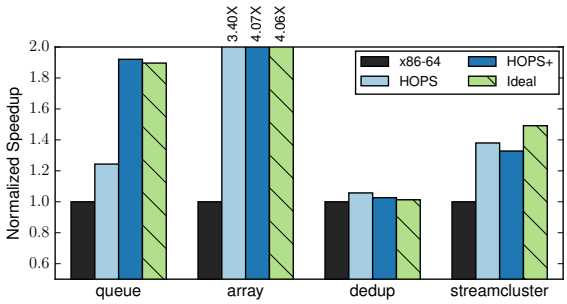

Figure 2: Performance of HOPS and HOPS+ compared to x86-64 or ideal and unsafe implementation.

4. Evaluation

Methodology. We extended gem5 to create *gem5-pm*, which realistically simulates PM. It adds an additional NUMA node with only memory (no CPU) to represent PM, which allows PM read/write latency and bandwidth to be separately configured. We will release gem5-pm publicly before the workshop. Using numactl, we constrain all volatile memory allocations to be satisfied from non-PM nodes. PM allocations are made from files on a tmpfs volume backed by memory on the PM node. Using gem5's support for full-system software stacks, our extensions allow researchers to realistically evaluate a variety of PM configurations.

We evaluate four separate implementations. First, we evaluate current x86-64 PM support using `clwb` and `clflush` instructions for flushing cachelines. We also evaluate HOPS and new HOPS+, which tracks SH in hardware. Finally, we evaluate an ideal implementation that elides cache flushes and thus cannot recover from crashes. In each, PM write latency is set as 10xDRAM latency and the PM controller has persistent write queues for tolerating longer latencies.

We use two microbenchmarks having multiple threads operating on a shared queue or array stored in PM. We also run two PARSEC benchmarks modified to store essential state in PM (i.e., the hashmap in dedup).

Performance. HOPS performs much better than x86-64--76% on average--and within 16.5% of ideal, unsafe performance. HOPS+ further outperforms x86-64 by 108% and HOPS by 18% and is within 1.5% of ideal. Variation between benchmarks occur because (a) greater write intensity increases differences between HOPS/HOPS+ versus x86 and (b) greater lock contention increases differences between HOPS+ and HOPS. Thus, in our view, hardware designs should consider both HOPS and HOPS+ as alternatives to x86-64 primitives.

References

[1] D. R. Chakrabarti, H-J. Boehm, and K. Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. OOPSLA '14.

[2] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch. Delegated persist ordering. MICRO-49, 2016.

[3] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton. An analysis of persistent memory use with whisper. ASPLOS '17, 2017.

[4] W. Ruan, T. Vyas, Y. Liu, and M. Spear. Transactionalizing legacy code: An experience report using gcc and memcached. ASPLOS '14.