Brief Announcement: Optimizing Persistent Transactions

Tingzhe Zhou
Lehigh University
tiz214@lehigh.edu

Pantea Zardoshti
Lehigh University
zardoshti@lehigh.edu

Michael Spear
Lehigh University
spear@lehigh.edu

ABSTRACT

There is a mechanical transformation by which algorithms for software transactional memory can be transformed to work with persistent memory. While correct, this transformation does not take into account differences between the persistent and volatile programming models. We show that fundamental properties of the data regions accessed by a persistent software transaction allow for a variety of optimizations not available in the volatile setting, and these lead to significant performance gains.

KEYWORDS

Transactional Memory, Concurrency, Persistence

ACM Reference Format:
Tingzhe Zhou, Pantea Zardoshti, and Michael Spear. 2019. Brief Announcement: Optimizing Persistent Transactions. In *31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '19), June 22–24, 2019, Phoenix, AZ, USA.* , 2 pages. https://doi.org/10.1145/3323165.3323176

1 INTRODUCTION

Transactional Memory (TM) [8] allows programmers to identify lexically-scoped *transactions* that must appear to execute in an atomic, isolated, and consistent manner, and then a run-time system monitors the behavior of concurrent transactions as they run. When transactions have non-conflicting memory accesses, they are allowed to complete. When transactions conflict, some subset of conflicting transactions are aborted, undone, and retried, to ensure that the behavior of the program is equivalent to one in which transactions ran sequentially.

Persistent memory (PM), such as 3D-Xpoint [10], fundamentally reshapes the memory and storage hierarchies by providing a single memory that is dense, byte-addressable, fast, and able to retain its contents without consuming energy. With PM, data does not persist until it exits the CPU cache and crosses some persistence threshold. A programmer can use special instructions (e.g., `clwb`) to force a cache line to persist, but cannot persist multiple lines of data without additional instrumentation. The overlap between TM and PM instrumentation is substantial, leading to several concurrent persistent TM (PTM) libraries [2, 3, 11].

There are two main approaches to PTM. With *undo*, a persistent transaction (PTx) writing to location W_i must first write the current value at W_i to its undo log. Then it must persist that write (via `clwb` and a memory fence). Finally, it can update W_i with a new value. In this manner, if the system crashes before the PTx completes, all of its W_i can be restored to their state prior to the PTx's execution. When the PTx completes, it simply discards its undo log. In contrast, with *redo*, a PTx writing to W_i places the new value in a private redo log. Subsequent reads of W_i must check the log to ensure processor consistency. When the PTx is ready to complete, it must persist the log (via `clwb` instructions and a memory fence), then replay its writes from the log. If the system crashes before the replay is complete, then the replay must be re-done after the system restarts. Note that even when a PTM does not support concurrent transactions, it must use either undo or redo.

SPAA '19, June 22–24, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6184-2/19/06…$15.00
https://doi.org/10.1145/3323165.3323176

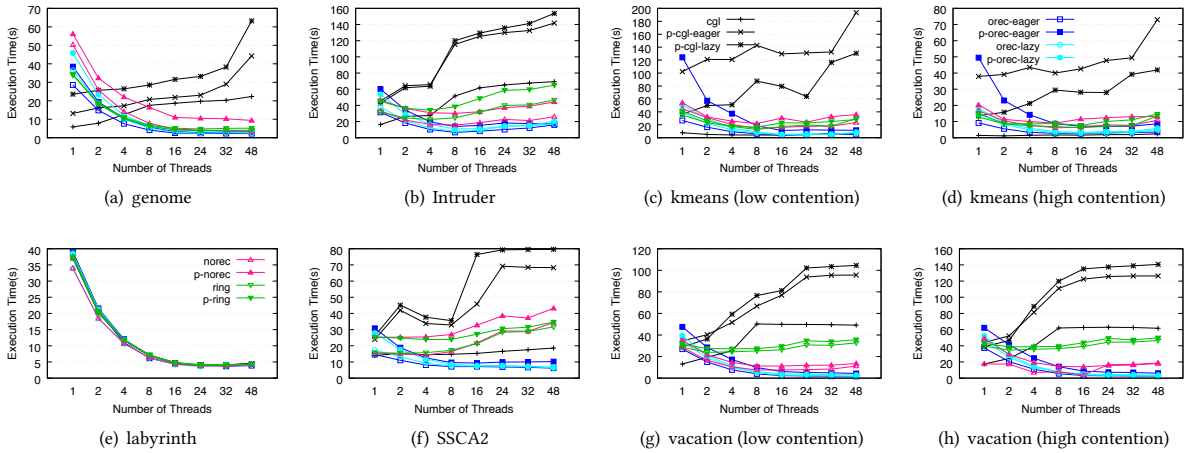
2 NONVOLATILE CONSIDERATIONS

Yoo et al. showed that TM implementations must incur significant overhead to contend with the idiosyncrasies of modern programs [7]. *Privatization* refers to an idiom in which a datum transitions from a state where it is accessed via transactions to a state where it is accessed without transactional synchronization (e.g., because it becomes logically private to a single thread). *Granular lost updates* occur when a TM algorithm performs its undo or redo logging at too coarse of a granularity. Consider a program in which B_1 and B_2 are adjacent bytes. If B_1 is private to thread T_1, but B_2 is accessible to all threads' transactions, then a transaction cannot perform undo or redo logging at cache-line or word granularity, lest its logging reads and writes of B_1 race with T_1. TM also must support *irrevocability* [4], where a transaction wishing to perform I/O or use code that cannot be rolled back first serializes all transactions, then runs in isolation *without per-access instrumentation.*

We observe a fundamental difference between PM and TM. In TM, the instrumentation requirements of a location are a dynamic property of how that location is *used.* In PM, the instrumentation requirements of a location derive from the physical characteristics of the underlying device. Languages are likely to require that all accesses to PM are performed from transactions, and hence every access to the PM will be instrumented. Thus the above concerns do not apply to PTM transactions, except in the unlikely case that they also access shared volatile memory. Thus in the common case, the following optimizations become possible:

(1) Undo and redo logging can occur at a coarse granularity (e.g., half cache line) without risking granular lost updates.
(2) Privatization-related overheads at the end of transactions will not be necessary.
(3) Irrevocability-related overheads at the beginning of transactions can be eliminated.

Note that (1) will reduce a constant overhead that occurs on every access in redo-based systems, (2) will reduce an overhead that is linear in the number of threads, and (3) will eliminate a memory fence and branch.



(a) genome (b) Intruder (c) kmeans (low contention) (d) kmeans (high contention)

(e) labyrinth (f) SSCA2 (g) vacation (low contention) (h) vacation (high contention)

Figure 1: PTM execution on STAMP benchmarks, using the default parameters

Threads	1	2	4	8	16	24	32	48
cgl_eager	1.18	1.03	1.05	1.05	1.02	1.03	1.18	1.10
cgl_lazy	1.19	1.17	1.12	1.13	1.10	1.79	1.27	1.11
orec_eager	1.00	1.06	1.15	1.45	2.90	4.22	4.77	4.82
orec_lazy	1.03	1.10	1.25	1.70	3.24	5.79	6.47	6.84
norec	0.98	1.06	1.18	1.40	1.70	2.07	2.12	2.21
ring	1.16	1.23	1.28	1.35	1.52	1.73	1.96	1.85

Table 1: Microbenchmark speedup for optimized PTM.

3 EVALUATION

We measure the impact of these changes on a Dell PowerEdge R640 with two 2.1GHz Intel Xeon Platinum 8160 processors and 192GB of RAM. Each processor has 24 cores / 48 threads, runs Red Hat Linux server 7.4, and LLVM/Clang 6.0 with O3 optimization. Experiments are the average of five trials; to avoid NUMA effects, we limited execution to a single CPU socket. Note that on this system, the RAM is not persistent, but `clwb` incurs accurate latencies.

We compare variants of four TM algorithms. In CGL, every transaction is protected by the same coarse-grained lock. In Orec, locations hash to entries in a table of 1M locks [6]. NOrec detects conflicts using values instead of locks [1]. Ring uses bit vectors to express the read and write sets of transactions [5]. Our default version of each algorithm is privatization-safe. "Eager" indicates undo, and "lazy" indicates redo. NOrec and Ring are always lazy. All experiments are run on the STAMP benchmark suite [9].

Figure 1 presents results for our unoptimized PTM algorithms. Our first finding is that latencies due to persistence are much more significant than latencies due to concurrency: persistent, concurrent TM outperforms persistent lock-based code as early as 2 threads, whereas non-persistent, concurrent TM cannot outperform non-persistent lock-based code until 4-8 threads. From this, we conclude that TM has the potential to be much more valuable in the persistent setting than in the non-persistent setting. The second finding is that redo substantially outperforms undo, except when there is a high incidence of read-only and read-mostly transactions (genome).

Table 1 presents preliminary speedup results for a data structure microbenchmark (RBTree, 16-bit keys, 80% lookup), with each PTM optimized according to Section 2. We observe two trends. The first is that coarsening the granularity of logging has a more significant impact on lazy algorithms. The second is that avoiding quiescence (only applicable to Orec PTM) is a powerful optimization, which appears to favor Orec-Lazy in all cases. In future work, we plan to complete our experimentation with STAMP to determine if the same result holds true across more varied workloads, and to explore additional optimizations to PTM.

ACKNOWLEDGEMENT

This work was supported by the Intel and NSF joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA) under Grant CCF-1723624.

REFERENCES

[1] Luke Dalessandro, Michael Spear, and Michael L. Scott. 2010. NOrec: Streamlining STM by Abolishing Ownership Records. In *Proceedings of the 15th PPoPP*. Bangalore, India.
[2] Joel Coburn et al. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth ASPLOS*. New York, NY, USA.
[3] Mengxing Liu et al. 2017. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *Proceedings of the 22nd ASPLOS*. Xi'an, China.
[4] Michael Spear et al. 2008. Implementing and Exploiting Inevitability in Software Transactional Memory. In *Proceedings of the 37th ICPP*. Portland, OR.
[5] Michael Spear et al. 2008. RingSTM: Scalable Transactions with a Single Atomic Instruction. In *Proceedings of the 20th SPAA*. Munich, Germany.
[6] Pascal Felber et al. 2008. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In *Proceedings of the 13th PPoPP*. Salt Lake City, UT.
[7] Richard Yoo et al. 2008. Kicking the Tires of Software Transactional Memory: Why the Going Gets Tough. In *Proceedings of the 20th SPAA*. Munich, Germany.
[8] Maurice P. Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th ISCA*. San Diego, CA.
[9] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford Transactional Applications for Multi-processing. In *Proceedings of IISWC*. Seattle, WA.
[10] Newsroom, Intel. 2018. Intel and Micron produce breakthrough memory technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.
[11] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight persistent memory. In *ACM SIGARCH Computer Architecture News*.