2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)

Optimizing Persistent Memory Transactions

1st Pantea Zardoshti*
Lehigh University
zardoshti@lehigh.edu

2nd Tingzhe Zhou*
Facebook
tzzhou@fb.com

3rd Yujie Liu
Google
yujie.liu@gmail.com

3rd Michael Spear
Lehigh University
spear@lehigh.edu

***Abstract*—Byte-addressable, non-volatile, random access memory (NVM) has the potential to dramatically accelerate the performance of storage-intensive workloads. For applications with irregular data access patterns, and applications that rely on ad-hoc data structures, the most promising model for interacting with NVM is a transactional model. However, the specifics of the model matter significantly.**

We introduce two models for programming persistent transactions. We show how to build concurrent persistent transactional memory from traditional software transactional memories. We then introduce general and model-specific optimizations that can substantially improve the performance of persistent transactions. Our evaluation shows a substantial improvement in the both the latency and scalability of persistent transactions.

***Index Terms*—Non-volatile Memory, Transactional Memory, Concurrency, Persistence, Synchronization, Performance**

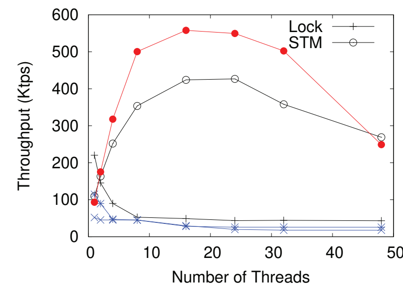
I. Introduction

Non-volatile byte-addressable memories present an exciting new opportunity for creators of high-performance systems. With non-volatile main memory (NVM), a program can avoid sources of latency associated with writing to traditional storage medium, and instead achieve persistence through memory writes to an NVM whose latency is within a constant factor of the speed of RAM.

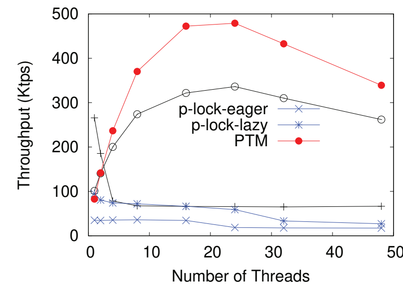
Transforming a program to use NVM can be non-trivial. Consider an application that persists program data via the file system interface. If the program crashes *between* file writes, or fails in a way that corrupts RAM, the integrity of the persisted data is not compromised. Similarly, if a fault occurs during a file write, the operating system or hardware (e.g., RAID) is responsible for ensuring write integrity. In contrast, if program memory is also the storage medium, then it is the program's responsibility to ensure the integrity of the data in the face of program crashes at arbitrary points in the program's execution.

Three programming models have emerged to address this challenge [1]. The simplest is to implement a file system on the NVM. No changes to the program are needed, but many of the performance and programmability benefits of NVM are lost. The second is ad-hoc techniques, through which the programmer uses custom assembly instructions to flush data from caches to the NVM, and fences to ensure ordering between these flushes and other accesses to program data. The third model exposes a transactional interface to programmers. With persistent transactional memory (PTM), programmers mark the regions of code that access NVM, and a run-time system tracks accesses to NVM within those regions.

* The first two authors contributed equally to this work.



(a) TPCC-HashTable



(b) TPCC-B+Tree

Fig. 1: TPCC benchmark performance. When the program data is in DRAM, synchronization is achieved using a single coarse lock or general-purpose STM. When the program data is in NVM, synchronization is achieved using a coarse lock + undo (eager), a coarse lock + redo (lazy), or PTM.

The run-time system ensures the atomicity of transactions by performing the necessary flushing and fencing, along with roll-forward or roll-back logging.

In many ways, PTM resembles software transactional memory (STM), an approach to creating high-performance concurrent programs [2], [3]. STM simplifies the creation of scalable programs by raising the level of abstraction for programmers: instead of thinking about explicit fine-grained locks, programmers mark regions of code that require atomicity, and then a run-time system tracks the memory accesses of those regions to maximize the number of transactions that can complete simultaneously without causing data races.

It is possible for special-purpose STM to achieve high scalability and modest run-time latency [4]. However, general-purpose STM must support challenging programming idioms that lead to run-time overheads and scalability bottlenecks [5].

978-1-7281-3613-4/19/$31.00 ©2019 IEEE
DOI 10.1109/PACT.2019.00025

IEEE computer society

Consider Figure 1, which shows the scalability of the TPCC "new order" benchmark when the underlying data store is represented as a hash table or a B+ tree. Every access to shared memory occurs within a language-level transaction. When we use a global mutex ("lock") to implement transactions, single-thread throughput is 2× to 3× that of a general-purpose STM. General-purpose STM requires 4–8 threads to match the throughput of the lock-based code, and at its peak it has only 2× the throughput.

The overheads that are most harmful to general-purpose STM arise because data is not statically or spatially partitioned according to whether it is accessed transactionally or not: The same datum can transition between transactional and non-transactional modes of access during execution, and adjacent bytes may be accessed via different modes. In contrast, it is reasonable to assume that data is statically partitioned at a coarse granularity for PTM: any variable stored in a page of NVM memory will always be accessed using a persistent transaction. As a result, there are optimizations available to PTM that are not available to general-purpose STM.

In this paper we propose and evaluate PTM optimizations, and contrast the added costs of PTM (cache flushes and fences) against the savings these optimizations provide relative to general-purpose STM. Figure 1 illustrates the cost of persistence. The p-lock-eager curve extends lock-based critical sections with undo log-based persistence, and p-lock-lazy uses redo logs. Undo logging has simpler instrumentation than redo logging, but more fences, leading to sequential slowdowns of 10× and 2×, respectively, versus non-persistent critical sections. The implementations perform the same number of flushes. The figure also shows the impact of PTM-specific optimizations: Our most optimized PTM achieves 90% of the performance of p-lock-lazy, scales to 5× its single-thread throughput, and outperforms the peak performance of a general-purpose STM. Note that while our PTM optimizations could be applied to certain STM workloads, the differences between STM and PTM programming models mean they can always be applied to PTM.

In this paper, we study the relationship between PTM and STM. We introduce a PTM transformation for lock-based, single version STM, characterize fundamental overheads associated with different programming models for PTM, and present optimizations for PTM within these programming models. In particular:

- We argue that PTM cannot use STM techniques to ensure progress, and we present a new progress mechanism.
- We demonstrate the importance of the persistence model on the performance of PTM algorithms.
- We introduce run-time optimizations for PTM, which raise performance by as much as 60%.

The remainder of this paper is organized as follows. In Section II, we introduce two system models for persistent transactions. Both take into account modern hardware trends, but one is more restrictive, constraining transactions to exclusively access NVM or DRAM, but not both. Then, in Section III, we present a general transformation for turning lock-based, single-version STM algorithms into PTM algorithms. We also present baseline performance numbers for PTM versus STM. Section IV presents a set of optimizations, some of which are only applicable to the more restrictive model, others of which apply to both models. We also measure the impact of each optimization, in isolation. Section V measures the impact of combining optimizations. Finally, Section VI summarizes our conclusions and discusses future work.

II. Programming Models for Persistence

The fundamental challenge for PTM is to ensure that program data is in a recoverable state at all times. That is, if the system should encounter a failure, then after the failure is addressed and the system restarted, the program's data should be valid. A transactional model ensures this property by executing atomic transactions that appear to happen all at once or not at all. However, the implementation of persistent transactions depends on hardware characteristics, the recovery model, and how a workload's transactions interact with the NVM and DRAM.

A. Hardware Persistence Domains

Marathe et al. [6] describe three hardware persistence domains. The simplest (persistence domain 0, or PDOM-0) only contains the NVM DIMM modules themselves. PDOM-1 adds the memory controller. PDOM-2 adds the entire CPU state, including caches and registers. As the persistence domain expands, it becomes easier to ensure a recoverable state. For example, if a power failure occurs in a PDOM-2 system, then when the machine is powered back, it can resume immediately, with no loss of state. In PDOM-1, memory buffers are flushed to DIMMs on power failure. As a result, programmers must ensure that data reaches the buffers in a correct order, through the use of `clwb` instructions that cause a cache line to write back, and `sfence` instructions to order the `clwb` with respect to subsequent stores. Finally, in PDOM-0, only the DIMMs are persistent, leading to additional instructions (e.g., `pcommit`) that run *after* all `clwbs`, to move data from the memory controller to the DIMMs.

Current and upcoming Intel systems provide PDOM-1. In PDOM-1, a failure that occurs in the middle of a transaction requires care to recover correctly: When the system recovers, the program counters at the time of the failure are unknown, so persistent transactions must either (a) use undo logs to record all overwritten values, so that they can roll back a transaction that is interrupted, or (b) use redo logs to record all to-be-updated values, so that they can roll forward a transaction after it is guaranteed to complete. The contents of either log must be stored in persistent memory, and updates require specific ordering with respect to accesses to program data.

B. Cost of Recovery

Typically, a persistent region is achieved by mapping a named, contiguous range of physical addresses from NVM into a program's virtual address space via `mmap` [7]. When a program restarts and reloads the region, its virtual-to-physical

benchmark	TPCC-B+Tree	TATP	TATP (1Kops/tx)
overhead	5.15%	2.67%	5.1%

TABLE I: Overhead of self-referential pointers

mappings may change. To minimize the time needed to recover a data structure, a program may use position-independent pointers (PIPs). These can either consist of two machine words (to represent a file ID and offset) [8] or a single machine word that represents an offset relative to the location of the pointer (e.g., for a pointer at `0xAA00` to refer to a word at `0xAAF0`, it would store the value `0xF0`). With PIPs, the pointers in a file are valid as soon as the file is mapped into the virtual address space. Otherwise, the pointers are invalid until they are adjusted by application-specific recovery code that traverses the entire persistent region.

Table I shows the increase in latency that PIPs introduce in a non-persistent program. The experiment was conducted by using our transactional instrumentation (discussed in Section III) to dynamically treat each pointer in the benchmark as a self-referential pointer. Considering these costs, we focus on non-position-independent pointers in this paper.

C. Accesses to Volatile Memory

A persistent memory region R_P is mapped into the virtual address space as a contiguous range, via `mmap`, and deallocated all at once via `munmap`. After being mapped, a persistent allocator manages R_P by creating and reclaiming contiguous ranges of memory within R_P. The persistent allocator cannot return reclaimed sub-ranges of R_P to the operating system. In contrast, allocators for traditional (volatile) memory can return individual pages of virtual memory to the operating system when they are no longer allocated.

In most TM systems, it must be assumed that while one thread T_1 is transactionally accessing some location L, another thread T_2 could commit a conflicting transaction that renders L unreachable, after which T_2 frees L. For general-purpose STM, L is in DRAM and could be returned to the operating system. Thus until T_1's transaction aborts, it could segfault if it accessed L. This is one manifestation of the "privatization" problem in STM: L has become logically private to T_2, but uses of L by T_2 can race with speculative accesses by T_1's doomed transaction [9]. The most common solution is to require transactions to block during their commit operation, until every concurrent transaction reaches a safe point. The blocking operation is commonly known as "quiescence" [10], and impedes scalability.

Yoo et al. observed that quiescence is necessary in the general case, but can be avoided on a workload-by-workload basis [5]. Zhou et al. later showed that quiescence overheads can be disabled at even finer granularity [11]. In the case of PTM, a stronger outcome is possible: if a PTM transaction only accesses NVM, it does not require quiescence [12].

D. Access Granularity

When transactions are used for concurrency, there is no need to instrument *every* access to DRAM; only accesses that could be concurrent with a transactional access to the same location need to be instrumented. As a result, general-purpose STM must assume a worst case, where on a single cache line, one byte may be private to a thread, while an adjacent byte is shared among many threads and accessed via transactions [5].

In contrast, persistence is not a dynamic property. Our focus on PDOM-1 means that *every* store to the NVM must be instrumented, so that `clwb` and `sfence` instructions can be performed correctly. Thus it is natural to require that every store be part of a transaction. We can also require that every load from a persistent region is part of a transaction (micro-transactions make the overhead of such a design minimal [4]). When the allocator uses padding and alignment to keep its metadata (e.g., boundary tags) on separate cache lines from program data, PTM algorithms for PDOM-1 can track memory at arbitrarily coarse granularities, saving overhead relative to the fine-grained tracking needed for general-purpose STM.

E. Multiple Persistent Regions

Applications should be able to work with multiple persistent regions at the same time. However, past work has established that some constraints may be enforced, such as forbidding pointers from NVM-backed regions to DRAM, or between NVM regions [7]. For the purposes of this paper, the distinction is not significant: as long as every attempt to `mmap` a named persistent region is done in a manner that persistently tracks (a) the name of the region (e.g., file name), (b) the virtual address assigned to the first byte of the mapped region, and (c) the size of the mapped region, then management of cross-region pointers can be handled by the code that runs upon recovery after a failure.

F. Models Considered in this Paper

From the above, we focus on two programming models in this paper. In both models, the underlying hardware is assumed to provide PDOM-1, and the programmer is expected to provide recovery code, so that persistent regions do not require position independent pointers. Note that during recovery, it will be necessary to both (a) apply a redo/undo log to clean up from incomplete transactions, and (b) remap pointers within the persistent region. Upon this base, the general persistence model (GP) assumes that any single transaction may access both NVM and DRAM, and that programs may access memory (reads and writes of DRAM, reads of NVM) from outside of transactions. The ideal persistence model (IP) assumes that a transaction may only access one type of memory (NVM or DRAM), and that every access to NVM is performed from within a transaction.

III. Transforming STM into PTM

We now present a strategy for transforming general-purpose STM algorithms into PTM algorithms. We focus on a set of lock-based, single-version STM algorithms [13], [14], [15], [16], [17], [18], [19], that are compatible with the C++ TM Technical Specification [20]. These algorithms involve

five functions that interact with program addresses and STM metadata:

- `Begin`: Start a transaction by snapshotting the thread's architectural state and possibly reading/updating global metadata.
- `Write(a, v)`: Speculatively write value `v` to address `a`. `Write` may save the old value at address `a` to an "undo" log and directly modify the value at `a`, or it might store `v` in a private buffer, to "redo" at commit time. It may also cause a transaction to validate (i.e., make sure no concurrent transaction made changes to a location the current transaction has already accessed).
- `Read(a)`: Attempt to read the value at `a`. Like `Write`, `Read` may cause a validation. It may also need to check if `a` is in the redo log managed by `Write`.
- `Commit`: Finalize the transaction's writes only if the reads all remain valid.
- `Abort`: Roll back any writes, clear all thread-local metadata, and restore the checkpoint from `Begin` to retry the transaction.

An instrumenting compiler [21], [22], [23], [24] inserts calls to `Begin` and `Commit` at the boundaries of transactions. Within the body of a transaction, every load and store is replaced with a call to `Read` or `Write`. A variety of optimizations have been proposed over the years to reduce the latency of this instrumentation.

To detect conflicts, STM algorithms map program addresses to some form of concurrency-control metadata. The metadata may be explicit readers/writer locks [19], or ownership records (orecs) [13] that superimpose a lock bit on a version number, so that optimistic readers can avoid acquiring read locks, instead validating the consistency of reads by tracking changes in the versions of the orecs protecting locations they read. In some cases, program values [17], [18] or bit vectors [16] are used instead of orecs.

The general read strategy for STM is similar regardless of the metadata: a transaction checks global metadata, reads a location, and possibly checks the metadata again. If the metadata is unchanged and compatible with previous reads, the new value can be returned (and the read set updated to include the new address). Otherwise, the transaction aborts. To write, a transaction either places an address/value pair into a write set (lazy), or locks the location, logs the old value in an undo log, and updates the value directly (eager). With lazy writes, it is necessary for reads to check the log, or else they may fail to see values previously written by the same thread in the same transaction. To commit, a lazy transaction acquires locks for all its writes, validates its reads, replays its redo log to update program memory, and releases locks. An eager transaction merely validates and releases locks. Conversely, to abort, a lazy transaction only needs to reset its local lists, whereas an eager transaction must use its undo logs to restore the values of locations it wrote, then release locks.

By definition, STM algorithms prevent deadlock. However, some algorithms are prone to livelock and starvation. A "distressed" transaction is one that repeatedly fails to commit, due to conflicts with other transactions. There are a number of sources of distress [25]. A thread may try to use an out-of-band "contention manager" to resolve conflicts [26]. In the worst case, a thread may resort to irrevocability [10], [27], a mechanism in which a transaction runs in isolation, without any instrumentation, in order to guarantee that it can complete. Since an irrevocable transaction can never abort, irrevocability is also used to allow transactions to turn off speculation, e.g., in order to perform I/O.

A. Ensuring Recoverability for Incomplete Transactions

Listing 1 presents the generic behavior of lazy and eager STM algorithms, and extends them to make them correct when operating on persistent regions. The comment `algorithm specific` indicates that the next lines of code would vary depending on the STM algorithm, but are immaterial to the persistent transformation. These algorithms treat all memory as persistent, issuing `clwb` and `sfence` instructions even when interacting with DRAM. To do so is inefficient, but correct, and simplifies the discussion in the remainder of this section.

In the GP model, the entirety of the effort in making a lazy transaction recoverable occurs in the `Commit` function. Prior to line 6, the state of memory is as if the transaction never happened. At line 6, the transaction has acquired all of its locks and ensured the validity of its reads. Additionally its redo log is stored in persistent memory. In traditional STM, the transaction would write back its redo log (line 10) and then clean up. In PTM, the transaction must first ensure that its entire redo log has reached a persistent level of the memory hierarchy. Line 6 performs up to W `clwb` instructions, where W is the number of entries in the redo log, to flush the entries to the persistent storage. It then sets the transaction's state to `active` (line 8). Prior to line 8, if the system crashed, then on recovery, the redo log would be discarded, and it would be as if the transaction never ran. After line 8, if the program crashed, the recovery procedure would see that s was `active` for this thread, and hence its redo log would need write-back.

On line 10, the redo log is replayed to memory. Note that this is an idempotent operation. If it were interrupted by a crash, then on recovery, it could be re-done (though potentially with re-mapped addresses, depending on the new base virtual address of the persistent region). Since write-back is idempotent, it does not matter if recovery leads to it executing more than once, but every write-back must reach persistent memory (via up to W `clwb` instructions on line 10). Once line 12 is reached, it is known that the write-back was successful, and need not be done again. After that, the thread can release its locks and clean up (line 14).

The eager algorithm is more complex. The main issue is that an undo (also idempotent) will be triggered by any system failure between the first write by a transaction and the point where it is known to have succeeded. We approximate this space by marking the transaction `active` on line 23, prior to its first read or write. As with the lazy algorithm, there are no changes to the read code. However, before writing, a transaction must log the old value to the undo log, and

Listing 1: Transforming STM to PTM

```
Thread-local Variables (Located in NVM):
    rl   : redo log for lazy algorithms, initially empty
    ul   : undo log for eager algorithms, initially empty
    s    : status of current transaction: {active, inactive}

   function Begin.Lazy()
       // algorithm specific:
1      StartTransaction ()

   function Commit.Lazy()
2      if rl.empty then
           // algorithm specific:
3          ResetMetadata ()
4          return
       // algorithm specific:
5      AcquireLocksAndValidate (rl)
       // changes for NVM:
6      clwb(rl)
7      sfence
8      clwb(s ← active)
9      sfence
10     clwb(rl.writeBack())
11     sfence
12     clwb(s ← inactive)
13     sfence
       // algorithm specific:
14     ResetMetadata ()

   function Read.Lazy(addr)
       // Check redo log:
15     if addr ∈ rl then
16         return rl.get(addr)
       // algorithm-specific:
17     val ←ConsistentRead (addr)
       // abort on error, else return val:
18     if err then Abort.Lazy ()
19     return val

   function Write.Lazy(addr, val)
       // Save addr/val to redo log:
20     rl.insert(addr, val)

   function Abort.Lazy()
       // algorithm specific:
21     ResetMetadata ()

   function Begin.Eager()
       // algorithm specific:
22     StartTransaction ()
       // changes for NVM:
23     clwb(s ← active)
24     sfence

   function Commit.Eager()
25     if ul.empty then
           // algorithm specific:
26         ResetMetadata ()
27         return
       // changes for NVM
28     sfence
29     clwb(s ← inactive)
30     sfence
       // algorithm specific:
31     ResetMetadata ()

   function Read.Eager(addr)
       // Fast path if owned
32     if ThisTxOwns (addr) then
33         return *addr
       // algorithm specific:
34     val ←ConsistentRead (addr)
       // abort on error, else return val:
35     if err then Abort.Lazy ()
36     return val

   function Write.Eager(addr, val)
       // Get permission to update addr
37     GetOwnershipOf (addr)
       // changes for NVM
38     clwb(ul.insert(addr, *addr))
39     sfence
       // update memory
40     clwb(*addr ← val)

   function Abort.Eager()
       // changes for NVM
41     clwb(ul.writeBack())
42     sfence
43     clwb(s ← inactive)
44     sfence
       // algorithm specific:
45     ResetMetadata ()
```

persist the change (lines 38–39). In addition, aborting is more complex, since it must restore memory, and that restoration must reach the NVM before the transaction marks itself as `inactive`.

For a successful transaction, both eager and lazy will incur $2W + 2$ `clwb` operations, to ensure that the redo or undo log is persisted, that all writes to program memory are persisted, and to persist two toggles of the transaction's state. The key difference is in fences: the lazy algorithm has 4, whereas the eager algorithm has $W + 3$ fences.

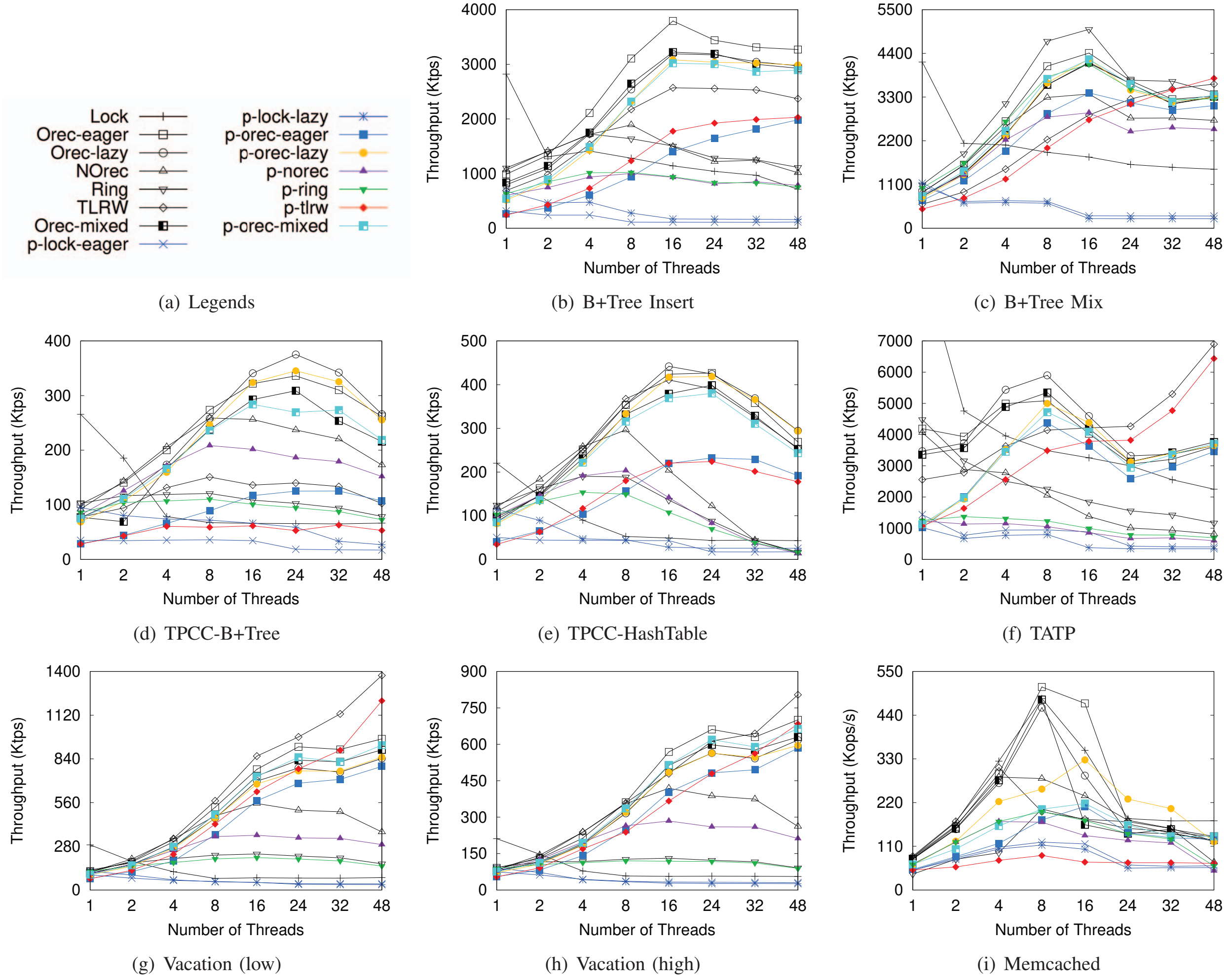
B. Ensuring Progress and Instrumentation

Unlike STM, PTM cannot use irrevocability. There are two problems. The first is that irrevocability does not use instrumentation, and thus there is no mechanism by which an irrevocable transaction can perform the `clwb` instructions needed to ensure that updates reach the NVM. The second is that in PDOM-1, the program counter is not persistent. If an irrevocable transaction is in the midst of performing I/O when there is a system failure, there may not be a mechanism for determining, at recovery-time, if the I/O has taken place. For the former problem, this means PTM cannot use irrevocability for ensuring progress. For the latter, programs that perform I/O from STM transactions will need to be rewritten in order to use PTM.

Without irrevocability, we require a new mechanism to prevent starvation and livelock. We propose an "hourglass" scheduler. We say that a transaction T_D is distressed if it has aborted k consecutive times. After k aborts, irrevocability would require that T_D set some flag to prevent new transaction attempts from starting, and then wait for all active transactions to commit or abort. Then T_D would run in isolation, after which it would clear the flag. Note that T_D must wait after setting the flag, because it will not use instrumentation, and thus will not be able to detect conflicts with concurrent transactions.

The key idea behind the hourglass is to reduce concurrency slowly, without making T_D wait, in the hopes that T_D can complete earlier than if it waited until it could run in isolation. Briefly, after T_D sets the flag, it immediately begins its (instrumented) transaction. Concurrent transactions continue to execute, and may cause T_D to abort. However, new transaction attempts are not allowed, to include attempts by transactions that aborted. Thus once T_D has set the flag, it is guaranteed that every concurrent transaction will either (a) commit, and then be forbidden from starting a new transaction, or (b) abort, and then be forbidden from starting a new transaction. While these transactions are running, T_D tries to complete. If it continues to fail, it is guaranteed to eventually run in isolation, and thus it can no longer starve.[1]

[1]Starvation is possible if some thread T can never acquire the flag. Substituting the flag with an adaptation of the wait-free enqueue of the MCS lock [28] would ensure progress even in the worst case.



(a) Legends (b) B+Tree Insert (c) B+Tree Mix

(d) TPCC-B+Tree (e) TPCC-HashTable (f) TATP

(g) Vacation (low) (h) Vacation (high) (i) Memcached

Fig. 2: Performance comparison of general-purpose STM to naive PTM (general model).

An added benefit of the hourglass, relative to irrevocability, is reduced latency in `Begin`. While the details of implementing irrevocability are outside of the scope of this paper, the coordination between regular and irrevocable transactions in `Begin` resembles Dekker locks [29], requiring every transaction to announce its intent to start, then fence, then check for irrevocable transactions, and possibly clear its intention and loop. In contrast, hourglass does not require transactions to announce that they have started, saving a memory fence. In the worst case, this could allow each thread to "sneak" one additional transaction attempt after T_D sets the flag, but as long as distressed transactions are rare, we expect the latency savings to outweigh this risk.

C. Performance of Naive PTM and STM

Figure 2 presents the performance of several general-purpose STM algorithms and their PTM equivalents across a set of common persistence benchmarks. We compare seven STM algorithms. "Lock" refers to a lightweight, non-concurrent STM where all transactions are protected by a single global lock. "Orec-eager" uses ownership records and undo logging, similar to GCC TM [14]. "Orec-lazy" is identical to "orec-eager", except it acquires locks at commit time and uses redo logging [13], [15]. "Orec-mixed" uses redo logging, but still acquires locks early, like orec-eager [30]. Orec-mixed has less overhead than orec-lazy on lines 15–16, because it can use knowledge of the locations it has locked to reduce the incidence of lookups in the redo log. However, for workloads with high contention, it is likely to scale worse than orec-lazy. "NOrec" [18] is a lazy algorithm that does not use orecs, instead relying on a single sequence lock to order transaction commits, and storing the values it reads so that it can validate address/value pairs instead of orec version numbers. "TLRW" [19] is an eager algorithm with carefully-crafted readers/writer locks. "Ring" [16] is a lazy algorithm that uses a log of 1024-entry bit vectors to capture the history of committed writer transactions. The persistent versions of the above algorithms are indicated by the "p" prefix. They were

created via the transformation in Listing 1. The exception is "Lock". We created two versions of "Lock", one eager, one lazy. These implementations bridge the gap between STM and past work on persistent critical sections [31]. Excluding "lock" algorithms, all STM and PTM algorithms ensure progress with the hourglass scheduler.

We instrumented code using an open-source LLVM extension for STM [24], which has been shown to have low instrumentation latency for general-purpose STM. We integrated the 7 STM and 8 PTM algorithms into it, which allowed us to isolate differences among STM algorithms, e.g., by using the same redo and undo log implementations. For the "p-lock-eager" PTM, we created a custom version of the LLVM extension that did not instrument reads. We also employed Link Time Optimization (LTO), which inlined most of the function call overhead related to instrumentation.

All experiments were conducted 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 uses LLVM/Clang 6.0 with O3 optimizations. 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 consider every open-source multi-threaded PTM benchmark we could find, which includes (i) one real world application, Memcached [32], [33], [1]; (ii) write-only benchmarks from DudeTM [34]: the TPCC transaction processing benchmark, TATP telecom application benchmark, and a B+ tree data structure microbenchmark; (iii) the "vacation" travel reservation benchmark [35], [1]. We tested the B+ tree for an insert-only workload, as well as a workload with an even mix of lookup, insert, range query, and remove operations. We ran two TPCC benchmarks, one using a B+ tree as the index, the other using a hashtable; both were the New Order workload. We tested Update Location transactions for TATP, using a hashtable for the index. We also looked at the recommended "high" and "low" contention settings for vacation. We evaluated Memcached by assigning 8 threads in one NUMA zone to serve as clients, and then varying from 1 to 48 worker threads in a second NUMA zone. For the Memcached experiments, we used a get/set ratio of 90/10.

The most striking finding of these experiments is that supporting persistence seems to tip the balance in favor of lazy strategies. We shall see in subsequent sections that this observation is mitigated, to a degree, by algorithm-specific optimizations for eager PTM. While the performance of orec-lazy and orec-eager are competitive with each other across all benchmarks, the linear number of `sfence` instructions hurts the performance of p-orec-eager. The (eager) TLRW algorithm is consistently among the best in Figure 2(c)(f)(g)(h), as is the persistent version. The success of persistent TLRW is its unique, scalable approach to privatization safety: it does not require quiescence, which introduces costs that grow with the number of threads.

Another surprise was the poor performance of NOrec. Support for persistence can increase the time that transactions spend holding locks. NOrec is more sensitive to this overhead than the other STM algorithms we consider. NOrec is lauded for its ability to provide a simple, scalable fallback when hardware TM cannot succeed [36], [37], but at the current time, the `clwb` instruction is incompatible with hardware TM. Without hardware acceleration, p-norec does not appear viable.

For orec-mixed, which matches the PTM algorithm in Mnemosyne [30], we see that the optimization for reducing lookups in the redo log has little benefit: it has a scalability cost, due to early locking, and does not save much read lookup latency. Consequently, orec-lazy performs better overall. Note that the scalability trends from the original Mnemosyne paper match with the behaviors we observed.

The last algorithm we considered was RingSTM. Like NOrec, RingSTM is a scalable lazy STM. RingSTM is less precise in its conflict detection than any of the other algorithms we consider, potentially leading to more aborts. However, it provides a feature that NOrec lacks: like the orec-based algorithms and TLRW, it can overlap the write-back of multiple software transactions. Unfortunately, naively transforming RingSTM to support persistence does not result in good performance.

IV. PTM Optimizations

A. Captured Memory

Most STM systems avoid instrumentation for accesses to memory on stack frames whose lifetime was limited by the scope of the transaction. In addition, Riegel et al. [22] and Dragojevic et al. [38] developed techniques to avoid instrumentation of "captured memory", locations that could be statically shown to be accessible only to the thread running the transaction. In some cases, captured memory would still require lightweight undo logging, e.g., for accesses to portions of the stack that were not transaction-local. While effective, captured memory optimizations are not part of modern STM implementations, due to the pointer analysis needed before any significant gains are achieved.

For NVM transactions, an important subset of captured memory is the memory allocated to a transaction during its execution. In our workloads, a transaction that allocates memory (e.g., calls `malloc`) is guaranteed to *write* to that memory. Thus it needs some amount of instrumentation (at least a `clwb` of each cache line written). A lightweight, dynamic optimization for these allocations can have a significant impact on latency. We call this optimization "last allocation tracking."

A typical STM will log the result of every `malloc` called within a transaction, so that it can `free` those pointers if the transaction aborts. To support last allocation tracking, we instead store a tuple, consisting of the returned value and also the size of the allocated region. We then make the following two modifications to the PTM implementation. First, on any `Read` or `Write`, we check if the address being accessed is within the range of the most recent allocation. If so, we do not perform any further instrumentation, instead performing the read or write directly to memory. This results in an

	TPCC-HashTable	TPCC-B+Tree	B+Tree (Insert)	Vacation (low)	Vacation (high)	Memcached
p-lock-eager	1.674	1.543	1.235	1.190	1.139	1.01
p-lock-lazy	1.055	1.045	1.041	1.026	1.021	1.01
p-orec-eager	1.674	1.443	1.126	1.179	1.177	1.272
p-orec-lazy	1.115	1.101	1.048	1.067	1.069	1.12
p-norec	1.107	1.081	1.061	1.052	1.024	0.999
p-ring	1.068	1.093	1.011	1.003	1.092	1.031
p-tlrw	1.626	1.358	1.127	1.173	1.162	1.264
p-orec-mixed	1.118	1.125	1.088	1.099	1.058	1.026

TABLE II: Speedup from the last allocation tracking optimization (single thread)

	TPCC-HashTable	TPCC-B+Tree	TATP	B+Tree (Insert)	B+Tree (Mix)	Vacation (low)	Vacation (high)	Memcached
p-lock-eager	1.229	1.519	1.049	1.366	1.167	1.231	1.197	1.11
p-lock-lazy	1.086	1.227	1.296	1.226	1.229	1.218	1.205	1.03
p-orec-eager	1.185	1.464	1.224	1.406	1.109	1.185	1.196	1.16
p-orec-lazy	1.041	1.124	1.084	1.113	1.096	1.104	1.120	1.114
p-norec	0.423	0.315	0.905	0.750	1.042	0.651	0.567	1.02
p-ring	1.022	1.121	1.107	1.158	1.075	1.076	1.058	1.022
p-tlrw	1.251	1.347	1.116	1.357	1.183	1.155	1.177	1.139
p-orec-mixed	1.088	1.113	1.052	1.063	1.086	1.144	1.097	1.055

TABLE III: Single-thread speedup of aligned memory and coarse-grained logging

additional branch before lines 15 and 20 for the lazy algorithm in Listing 1, and before lines 32 and 37 of the eager algorithm. Second, at commit time, prior to line 7 of the lazy algorithm or line 29 of the eager algorithm, we loop through the list of allocations. For each, we iterate through its range, and `clwb` once per cache line. In this manner, we ensure that all writes to the new memory region have crossed the persistence domain before marking the transaction as complete. For completeness, note that these steps must also be performed in the read-only fast path of the commit operations, in case a transaction's only writes are to a region it allocated.

Last allocation tracking affects latency, but not scalability. To evaluate its effectiveness, Table II presents its impact on single-threaded execution of our benchmarks. In the "lock-eager" algorithm, where reads are not instrumented, the impact should be least; however, it is 13% or higher for all but Memcached. This is due to the reduction in `sfence` instructions that the technique achieves for eager algorithms. Indeed, p-orec-eager and p-tlrw also show substantial improvement. The benefits for lazy algorithms are more limited (3% to 12%), and more in line with the gains to be expected from captured memory instrumentation in STM. We conclude that last allocation tracking is a generally effective strategy, and particularly effective for eager PTM.

B. Memory Alignment and Logging Granularity

The IP model assumes addresses in NVM will only be accessed transactionally. Since NVM is given to the program at the granularity of pages, the IP model permits a coarser granularity of management than in general-purpose STM.

In STM, when a transaction accesses the byte at address A, it cannot eagerly read adjacent bytes, even if those addresses are protected by the same metadata (e.g., the same orec), because adjacent addresses may be accessed by a concurrent, non-transactional thread. Thus with undo logs, entries in the log must have variable granularity, and with redo logs, a system must either (a) log at the granularity of individual bytes, or (b) accompany each coarse log entry with a bitmap indicating which bytes of the entry are valid. These choices also affect how the redo log is checked during reads (lines 15–16): In a general-purpose STM implementation that supports C++ casting and mixed-granularity access, the lookup in Listing 1 may need to use the bitmap to compose bytes from the redo log with bytes that would be read on line 17.

Composing logging granularity with memory alignment creates a new opportunity to improve PTM performance. We dynamically replace each `malloc` of NVM with a call to `aligned_malloc`, and we align on a boundary that is determined by the underlying STM (e.g., to match orec granularity). We then log at that same granularity. For undo logging, this means we can log at a fixed granularity (we chose half a cache line, 32 bytes); the log then holds $\langle address, value\rangle$ tuples, instead of $\langle address, value, length\rangle$. For redo logging, the redo log no longer needs a bitmap, and redo log entries always are populated with a full 32 bytes of program data read from NVM. As discussed above, `Read` is also simplified, leading to fewer instructions and fewer branches on each read.

Our decision to use 32-byte granularity was based on balancing improvements in performance (especially for TPCC and Vacation) against the increased potential for conflicts due to false sharing and the potential for unnecessary logging due to poor spatial locality (especially in the B+ Tree and TATP). In separate experiments, we found that 16-byte granularity improved performance for the B+ Tree, and 64-byte granularity was best for TPCC. We opted to show a single consistent granularity, and we encourage developers to think carefully about granularity, so that it can be a tunable parameter in future systems.

Table III presents the impact of this optimization for single-threaded code. Note that while the optimization has the potential to harm scalability, if threads concurrently access the same cache line, such problems do not manifest in our

	TPCC-HashTable	TPCC-B+Tree	TATP	B+Tree (Insert)	B+Tree (Mix)	Vacation (low)	Vacation (high)	Memcached
Speedup	13.3%	14.2%	0.67%	10.91%	2.5%	2.85%	4.05%	9.07%

TABLE IV: Single-thread speedup of fence pipelining for TLRW

benchmarks, which exhibit good spatial locality and are free from false sharing.

The impact of the optimization varies by workload and PTM algorithm. While it is generally effective, it performs poorly for NOrec. NOrec differs from the other algorithms in this study, in that it does not use metadata to detect conflicts among threads. Instead, it logs the locations that were read, and the values observed at those locations. Coarsening the redo log granularity leads to a coarsening of the read log, which means that any read must log 32 bytes. This increased write pressure during reads translates to worse performance for NOrec, while the other PTM algorithms enjoy speedups of 2% to 46%.

Note that in the GP model, exploiting this optimization would require the transaction to maintain two redo logs: one for NVM addresses, with coarse granularity, and one for DRAM addresses, with STM granularity.

C. Fence Pipelining

Eager PTM algorithms incur a penalty due to the need to flush undo log entries before writing new values to the NVM. With W writes, the addition of W `sfence`s has a deleterious effect on single-thread latency, even in p-lock-eager.

Among eager STM algorithms, TLRW is unique in that every memory access, whether read or write, must acquire a lock. These acquisition operations cause the same type of ordering as is needed for undo logging. That is, in TLRW, line 34 has a memory fence, as does line 37. However, the fence on line 37 must precede the `clwb` on line 38, as it is necessary before dereferencing `addr`.

While we cannot combine two fences within the same `Write` call, we can coalesce the `sfence` on line 39 with a subsequent fence in the *next* call to `Read` or `Write`. Our TLRW "pipeline" optimization defers the write and `clwb` on line 40, by storing the address and value to a thread-local variable. It also omits the fence on line 39. Then, on the next `Read` or `Write`, after line 34 or line 37, we execute the deferred store and `clwb`. We also execute the deferred store immediately before line 33, and immediately before line 29. In this manner, the most recent write to NVM delays until the transaction performs its next operation that requires a memory fence, allowing the fences to be combined. As a result, persistent TLRW is able to reduce its fencing overhead to the same as the original TLRW algorithm.

Table IV shows the impact on single-thread latency for TLRW when using this optimization. Across our benchmarks, the optimization reaches 14% speedup in the best case, and never reduces performance.

D. Deferred Flushing

In lazy PTM algorithms, the `Commit` operation is responsible for writing values to main memory on line 10. These values must be flushed to the NVM via `clwb` instructions. Unfortunately, `clwb` has high latency, and line 10 executes while holding locks. We propose shrinking the critical sections by performing the `clwb` instructions after locks are released.

As long as a thread has not yet marked itself $inactive$, it would seem that it could tolerate a crash between line 10 and some later point when it has released locks but not yet performed its `clwb`s: during recovery, it could replay its redo log again, and perform the `clwb`s then. However, an incorrect ordering could arise. In Listing 1, if two transactions both write to X, and there is a system failure during one of the transactions' execution of line 10, then the other thread is either (a) not yet to line 5, or (b) past line 14. Thus during recovery, only one thread will have X in its $active$ redo log. In contrast, if the `clwb`s happen after locks are released, but before becoming $inactive$, then two threads can have X in their redo logs, and the recovery algorithm will not know which to write back first.

Conveniently, in all but the TLRW algorithm, some global counter, or single global lock, is used to order all writing transactions. For algorithms that use a global counter, we can use the value of this counter in place of $active$, and 0 in place of $inactive$, to convey the commit order to the recovery algorithm, so that writeback can be done in the proper order. For TLRW and single-lock algorithms, we use the CPU's high-resolution timestamp counter (`rdtscp`), which is coherent across cores on the x86.

In more detail, we replace the $active$ status word with a timestamp representing the transaction's commit order. Then we split line 10, such that the write-back occurs *without* `clwb`s. After write-back completes, we release locks (part of line 14), then we issue the `clwb`s, then clear the status, and then run the rest of line 14. In this manner, flushing new values to the persistence domain is done without holding locks. Note that if a thread delays before issuing its `clwb` of location L, then some other thread may lock L, update it, and flush its update. In this case, coherence ensures that the delayed `clwb` will flush the new value.

Table V depicts the performance improvement from this optimization. The effect is most pronounced for TATP, which is dominated by small transactions. In TATP, at 24 threads performance is more than $2.7\times$ the unoptimized 24-thread throughput. For some workloads, we observe a small slow-down (up to 3%), due to the shorter critical sections leading to transactions committing in different orders. However, the overall impact is positive.

V. Combining Optimizations

We conclude our evaluation by measuring the impact of optimizations, in combination, for each benchmark. We are

	TPCC-HashTable			TATP			B+Tree (Insert)			Vacation (low)			Memcached		
Threads	4	8	24	4	8	24	4	8	24	4	8	24	4	8	24
p-lock-lazy	1.199	1.003	1.258	1.317	1.392	2.791	1.069	1.643	1.056	1.035	1.032	1.085	1.005	1.33	1.354
p-orec-lazy	1.026	1.052	1.007	1.018	1.056	1.050	1.010	0.998	1.008	1.012	1.017	0.998	1.057	1.368	1.485
p-norec	1.108	1.167	1.147	1.301	1.245	1.229	1.086	1.141	1.169	1.061	1.087	1.100	0.992	1.256	1.272
p-ring	1.041	1.028	1.130	1.152	1.211	1.457	1.085	1.145	1.152	1.027	1.014	1.017	1.111	1.184	1.631
p-orec-mixed	0.998	1.003	1.012	0.979	1.002	1.034	1.002	1.017	1.047	0.971	0.980	0.991	1.201	1.271	1.321

TABLE V: Speedup of Deferred Flushing (single thread)

particularly focused on understanding the implications of the programming model on performance.

Recall that in the general (GP) model, a single transaction might access both NVM and DRAM. In such a scenario, the cost of determining the nature of an address may be expensive: if N persistent heaps are mapped into the program's address space, then determining if an address A is in NVM could require N base/bound checks. To avoid the worst case, our GP implementations of PTM omit optimizations that are inappropriate for DRAM transactions. Since every GP transaction might access DRAM, we also keep privatization overheads (e.g., quiescence) in place. In contrast, PTM algorithms in the IP model follow prior work [30], [12] in only requiring quiescence when unmapping a persistent region, so that individual transactions do not need to quiesce.

This leads to the following configurations. For the GP model, we use the hourglass scheduler, last allocation tracking, fence pipelining (in TLRW), and deferred flushing. For the IP model, we add 32-byte logging granularity for redo and undo logs, and we remove quiescence. Note that TLRW does not require quiescence.

A. Performance in the General Persistence Model

In Figure 3, algorithms optimized for the GP model are prefixed with `gp`. After optimization, the performance for each algorithm improves, often by a substantial margin. The peak speedup of the best choice, when compared to the naive PTM transformation (from Figure 2), is from 1.1× (b) to 1.3× (i), with a geometric mean of 1.22×. If it were possible to pick the best PTM algorithm at run time, based on advance knowledge of the workload, thread count, and other program characteristics, we might expect this much improvement. Note that the decision may not be difficult, since either gp-orec-lazy or gp-tlrw is near the top in every workload. If only one algorithm could be used for all programs, gp-orec-lazy appears to provide the best overall performance.

Fence pipelining had a significant effect on eager TLRW, helping it to perform 1.7× better than other PTM algorithms on Vacation, compared to 1.15× without the optimization. However, eager TLRW has unsatisfactory performance in benchmarks with high write frequencies or large read sets, due to the latency of acquiring locks on every read.

The most disappointing results were for RingSTM and NOrec. While these algorithms provide privatization safety without quiescence, neither matched gp-orec-lazy at high thread counts. In separate experiments, we made RingSTM somewhat more competitive at low thread counts by using its "relaxed commit order" optimization [16]. However, this optimization sacrifices privatization safety, and is offset by a need for quiescence. In the case of NOrec, note that whenever a writing transaction commits, all other transactions block. Adding `clwb` instructions and fences to the commit sequence for PTM increases latency at this most critical point.

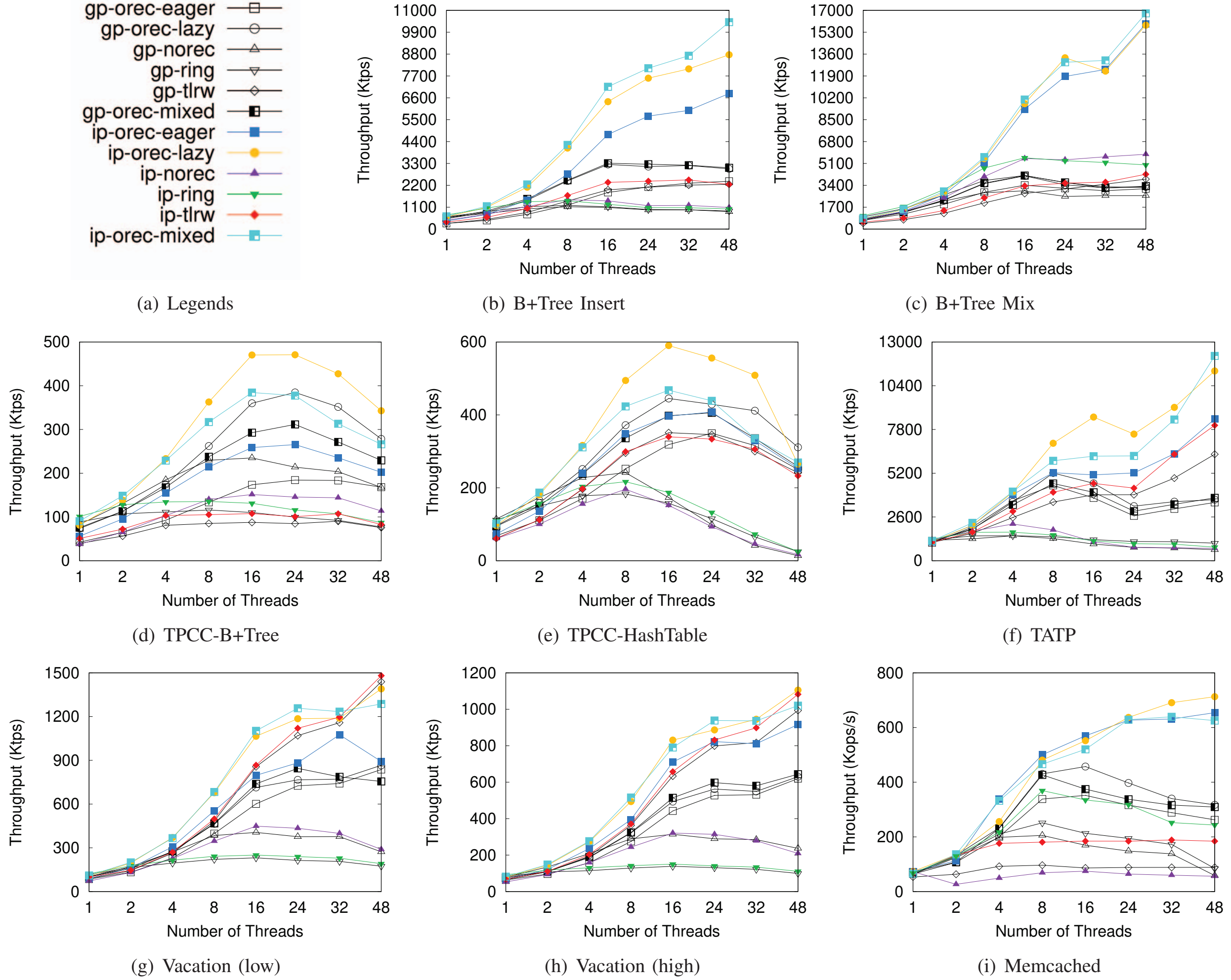
B. Performance in the Ideal Persistence Model

We now turn our attention to the performance of PTM algorithms after applying the additional optimizations of the IP model. Here, we find that improvements in single-thread latency, arising from the use of coarse granularity logging and last access tracking, are stable: the boost to algorithms at one thread is borne out at higher thread counts. Furthermore, when it can be assumed that transactions *only* access NVM, and thus do not require quiescence, the scalability is greatly improved.

As a result, the three orec-based algorithms rise above the rest, with only one instance (Vacation, low contention, 48 threads) where TLRW outperforms. Furthermore, while optimizations are effective in reducing the overhead of orec-eager, the lazy algorithms perform better, and in general, increasing laziness (via commit-time locking) has a beneficial impact on scalability. The mixed mode (encounter-time locking with write-back), which was proposed for Mnemosyne [30], occasionally outperforms orec-lazy, but when orec-lazy performs better, it is by a larger margin, suggesting that orec-lazy is the best PTM algorithm for the IP model. The peak performance speedup for orec-lazy, when comparing with the GP model, is from 1.23× (d) to 4.06× (c), with a geometric mean of 1.92× across all benchmarks.

C. Implications for STM

As we have discussed throughout this paper, the programming models for STM and PTM are governed by different requirements. General-purpose STM must assume a worst case, as described by Yoo et al. [5], that does not occur for PTM. Still, the optimizations proposed in this paper can be applicable to STM, so long as the underlying workload and system exhibit the right characteristics. For example, the workloads in our experiments are compatible with the IP model, and thus any STM produced from an IP algorithm by removing `clwb` and `sfence` instructions should be both correct and faster than that corresponding IP algorithm. As future work, we believe it will be valuable to develop static analyses that can discover when PTM optimizations can be applied to an STM workload.



(a) Legends (b) B+Tree Insert (c) B+Tree Mix

(d) TPCC-B+Tree (e) TPCC-HashTable (f) TATP

(g) Vacation (low) (h) Vacation (high) (i) Memcached

Fig. 3: Optimized performance of PTM algorithms. The GP prefix indicates that the PTM applied optimizations for the General Persistence model. IP indicates that additional optimizations for the Ideal Persistence model were also applied.

VI. Conclusions and Future Work

In this paper, we studied the performance of PTM algorithms. We considered two programming models, one in which a single transaction could interact with traditional DRAM and also NVM, and another in which transactions only accessed NVM. We also presented optimizations for PTM, significantly improved PTM latency and throughput.

Our study is the most comprehensive to date, considering a diverse set of STM algorithms and every publicly-available PTM benchmark. It shows that the choice of PTM algorithm will depend critically on the programming model: under our general persistence model, a variety of PTM algorithms performed comparably well, especially at low thread counts, whereas in the ideal model, a single algorithm was best. An important question is whether the ideal model is realistic: at the time of this writing, there are no commercially-available NVM-only systems, nor are there any production-worthy applications that use STM for transactions over DRAM. Furthermore, at the present time hardware TM (HTM) is not compatible with NVM.

In the future, we plan to use HTM to prefetch or precompute results for persistent transactions, even if those results must be flushed using a software protocol. We also plan to look at special-purpose STM algorithms, to see if there are opportunities to optimize them for PTM. Additionally, while the p-orec-lazy algorithm has proven to be the most successful, its latency for performing lookups in its redo log is not trivial. We plan to develop hardware extensions, such as content-addressable memory, to reduce this overhead in the common case. We also plan to explore new STM and PTM algorithms that are able to offer stable performance and good scaling on NUMA systems. Lastly, we plan to explore static analysis that can reduce instrumentation, e.g., by decomposing the PTM interface and coalescing undo or redo operations, similar to past work on STM [39].

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers, as well as our colleagues in the SG5 group, for their feedback and advice. This work was supported by the Intel and NSF joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA) under grant CCF-1723624, as well as NSF grant CAREER-1253362. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Intel or the National Science Foundation.

REFERENCES

[1] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, “An analysis of persistent memory use with whisper,” in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS’17, Xi’an, China, April 2017.

[2] N. Shavit and D. Touitou, “Software Transactional Memory,” in *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, Ottawa, ON, Canada, Aug. 1995.

[3] M. P. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III, “Software Transactional Memory for Dynamic-sized Data Structures,” in *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*, Boston, MA, Jul. 2003.

[4] A. Dragojevic and T. Harris, “STM in the Small: Trading Generality for Performance in Software Transactional Memory,” in *Proceedings of the EuroSys2012 Conference*, Bern, Switzerland, Apr. 2012.

[5] R. Y. et al., “Kicking the Tires of Software Transactional Memory: Why the Going Gets Tough,” in *Proceedings of the 20th SPAA*, Munich, Germany, Jun. 2008.

[6] V. Marathe, A. Mishra, A. Trivedi, Y. Huang, F. Zaghloul, S. Kashyap, M. Seltzer, T. Harris, S. Byan, B. Bridge, and D. Dice, “Persistent memory transactions,” in *arXiv preprint arXiv:1804.00701*, 2018.

[7] J. C. et al., “NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories,” in *Proceedings of the Sixteenth ASPLOS*, New York, NY, USA, Mar. 2011.

[8] Intel Corporation, “Nvml: Implementing persistent memory applications,” https://www.snia.org/sites/default/files/.

[9] M. Spear, V. Marathe, L. Dalessandro, and M. Scott, “Privatization Techniques for Software Transactional Memory (POSTER),” in *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing*, Portland, OR, Aug. 2007.

[10] A. Welc, B. Saha, and A.-R. Adl-Tabatabai, “Irrevocable Transactions and their Applications,” in *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, Jun. 2008.

[11] T. Zhou, P. Zardoshti, and M. Spear, “Practical Experience with Transactional Lock Elision,” in *Proceedings of the 46th International Conference on Parallel Processing*, Bristol, UK, Aug. 2017.

[12] A. Correia, P. Felber, and P. Ramalhete, “Romulus: Efficient algorithms for persistent transactional memory,” in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*. ACM, 2018, pp. 271–282.

[13] D. Dice, O. Shalev, and N. Shavit, “Transactional Locking II,” in *Proceedings of the 20th International Symposium on Distributed Computing*, Stockholm, Sweden, Sep. 2006.

[14] P. F. et al., “Dynamic Performance Tuning of Word-Based Software Transactional Memory,” in *Proceedings of the 13th PPoPP*, Salt Lake City, UT, Feb. 2008.

[15] M. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott, “A Comprehensive Strategy for Contention Management in Software Transactional Memory,” in *Proceedings of the 14th ACM Symposium on Principles and Practice of Parallel Programming*, Raleigh, NC, Feb. 2009.

[16] M. S. et al., “RingSTM: Scalable Transactions with a Single Atomic Instruction,” in *Proceedings of the 20th SPAA*, Munich, Germany, Jun. 2008.

[17] M. Olszewski, J. Cutler, and J. G. Steffan, “JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory,” in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, Brasov, Romania, Sep. 2007.

[18] L. Dalessandro, M. Spear, and M. L. Scott, “NOrec: Streamlining STM by Abolishing Ownership Records,” in *Proceedings of the 15th PPoPP*, Bangalore, India, Jan. 2010.

[19] D. Dice and N. Shavit, “TLRW: Return of the Read-Write Lock,” in *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, Jun. 2010.

[20] ISO/IEC JTC 1/SC 22/WG 21, “Technical Specification for C++ Extensions for Transactional Memory,” May 2015. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4514.pdf

[21] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian, “Design and Implementation of Transactional Constructs for C/C++,” in *Proceedings of the 23rd ACM Conference on Object Oriented Programming, Systems, Languages, and Applications*, Nashville, TN, USA, Oct. 2008.

[22] T. Riegel, C. Fetzer, and P. Felber, “Automatic Data Partitioning in Software Transactional Memories,” in *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, Jun. 2008.

[23] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Riviere, “Evaluation of AMD’s Advanced Synchronization Facility within a Complete Transactional Memory Stack,” in *Proceedings of the EuroSys2010 Conference*, Paris, France, Apr. 2010.

[24] P. Zardoshti, T. Zhou, P. Balaji, M. Scott, and M. Spear, “Simplifying Transactional Memory Support in C++,” pp. 25:1–25:24, Jul. 2019.

[25] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood, “Performance Pathologies in Hardware Transactional Memory,” in *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, CA, Jun. 2007.

[26] W. N. Scherer III and M. L. Scott, “Advanced Contention Management for Dynamic Software Transactional Memory,” in *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, Las Vegas, NV, Jul. 2005.

[27] M. S. et al., “Implementing and Exploiting Inevitability in Software Transactional Memory,” in *Proceedings of the 37th ICPP*, Portland, OR, Sep. 2008.

[28] J. M. Mellor-Crummey and M. L. Scott, “Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors,” *ACM Transactions on Computer Systems*, vol. 9, no. 1, 1991.

[29] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[30] H. Volos, A. J. Tack, and M. M. Swift, “Mnemosyne: Lightweight persistent memory,” in *ACM SIGARCH Computer Architecture News*, March 2011.

[31] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, “Atlas: Leveraging Locks for Non-Volatile Memory Consistency,” in *ACM SIGPLAN Notices*, vol. 49, no. 10. ACM, 2014, pp. 433–452.

[32] memcached.org, “Memcached, a distributed memory object caching system.” 2014, http://memcached.org/.

[33] W. Ruan, T. Vyas, Y. Liu, and M. Spear, “Transactionalizing Legacy Code: An Experience Report Using GCC and Memcached,” in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, Salt Lake City, UT, Mar. 2014.

[34] M. L. et al., “DudeTM: Building Durable Transactions with Decoupling for Persistent Memory,” in *Proceedings of the 22nd ASPLOS*, Xi’an, China, Apr. 2017.

[35] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, “STAMP: Stanford Transactional Applications for Multi-processing,” in *Proceedings of IISWC*, Seattle, WA, Sep. 2008.

[36] A. Matveev and N. Shavit, “Reduced Hardware NORec: A Safe and Scalable Hybrid Transactional Memory,” in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, Istanbul, Turkey, Mar. 2015.

[37] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. Scott, and M. Spear, “Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory,” in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, CA, Mar. 2011.

[38] A. Dragojevic, Y. Ni, and A.-R. Adl-Tabatabai, “Optimizing Transactions for Captured Memory,” in *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures*, Calgary, AB, Canada, Aug. 2009.

[39] T. Harris, M. Plesko, A. Shinar, and D. Tarditi, “Optimizing Memory Transactions,” in *Proceedings of the 27th ACM Conference on Programming Language Design and Implementation*, Ottawa, ON, Canada, Jun. 2006.

Appendix

A. Abstract

This artifact includes two parts to evaluate proposed optimization techniques on different algorithms.

- **Performance comparison of STM to naive PTM** as shown in Figure 2 in the paper.
- **Optimized performance of PTM algorithms** as shown in Figure 3 in the paper.

B. Artifact check-list (meta-information)

- **Algorithm: all algorithms are listed in: llvm-transmem/Makefile.libnames**
- **Program:** B+Tree, TPCC-B+Tree, TPCC-HashTable, TATP, Vacation and Memcached
- **Compilation:** Clang++ 6+, GCC
- **Run-time environment:** Linux
- **Hardware:** Intel Xeon Platinum and its successors with CLWB and SFENCE instructions supported
- **Execution:** python/bash script
- **Metrics:** Throughput (Ktps/Kops/s)
- **Output:** Figures, text files
- **How much disk space required (approximately)?:** 1GB
- **How much time is needed to complete experiments (approximately)?:** 23 hours
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Lehigh University
- **Archived?:** Yes, DOI.10.5281/zenodo.3346054.

C. Description

1) How delivered: All the source codes and inputs are delivered via Github and Zenodo

2) Hardware dependencies: Intel Xeon Platinum and its successors with CLWB and SFENCE instructions supported

3) Software dependencies: Docker CE, LLVM, C++ Compiler, Clang++

4) Data sets: None.

D. Installation

we have prepared a docker file so that users can build their docker image and perform all the experiments inside the docker. Note that the docker can influence the result. In order to avoid the overhead, there is a script to install all the dependencies on ubuntu. https://github.com/pzardoshti/llvm-transmem

E. Experiment workflow

Follow the instructions shown in https://github.com/pzardoshti/llvm-transmem to build and run the docker image or the script. Once inside the container, download the package by the following command.
- git clone https://github.com/pzardoshti/llvm-transmem.git
- cd llvm-transmem

Compile all libraries and benchmarks
- ./compile.sh

Run all benchmarks
- ./run.sh all

generate Figure 2 and 3
- ./generate.sh

F. Evaluation and expected result

The result of the artifact depends on the hardware, the result may be different from the one presented in the paper. The output graphs are located into the result/plots folder and the raw data can be found in results/data folder which includes throughput.

G. Experiment customization

In order to run each benchmark separately use alphabet number related to each benchmarks based on Figures 2. For example to run bplustree (insert), type ”./run.sh b” and to generate its graph, type ”./generate.sh b”. Also scripts related to each benchmarks are located in scripts/*$benchmark_name*.

H. Notes

The platform is extendable by adding a new algorithm and reusable by adding new benchmarks. To add new benchmarks, please follow instruction in README file located in the root path. Furthermore, If you have trouble to build and use the docker container or your machine does not support CLWB and SFENCE instruction, please contact us via the AE committee, and we will provide a test machine.