2019

Tailoring Transactional Memory to Real-World Applications

Tingzhe Zhou
Lehigh University

Tailoring Transactional Memory to Real-World Applications

by

Tingzhe Zhou

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Doctor of Philosophy

in

Computer Science

Lehigh University

May, 2019

Approved and recommended for acceptance as a dissertation in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Date

Dissertation Advisor

Committee Members:

Prof. Michael Spear, Committee Chair

Prof. Hank Korth

Prof. Roberto Palmieri

Dr. Victor Luchangco

Acknowledgements

The life for a Ph.D. student is the combination of happiness, arduousness, and adventurousness. I have been told thousands of times about how hard to get a Ph.D. in computer science. However, I was super lucky to have so many friendly, wise, and patient people helped me and made my Ph.D. life much more relaxed and unforgettable than it supposed to be.

I want to express my heartfelt gratitude to my advisor, professor Michael Spear, for his invaluable guidance, and continuous support, caring, patience, and encouragement. Prof. Spear has always been enthusiastic and helpful for answering all kinds of questions (from raising new ideas to writing and debugging the code). I could not keep making progress on my research projects without his selfless guidance and endless patience. Apart from research, he also comforted me when I was struggling with difficulties, taught me how to arrange things efficiently, and encouraged me when I was about to give up. Furthermore, Prof. Spear was super generous about taking vacation time, going for internships, and traveling for conferences. With his generous support, I went through a diversified and colorful Ph.D. life. I could not express how lucky am I worked under his guidance.

I am grateful to the advice from all of my committee members. Specifically, I genuinely thank Prof. Roberto Palmieri for his numerous practical suggestions and encouragement towards my research and my life, and Prof. Hank Korth for his tips on academic writing, presentation, and open-mind for broad research areas. I want to express my gratitude to Dr. Victor Luchangco for helping me polishing papers and being extremely patient for answering questions. I sincerely appreciate their precious time and insightful suggestions regarding my Ph.D. proposal, general exam, and dissertation. I would also like to thank

Dave Dice and Tim Harris for their advice and guidance during the conduct of some of my research.

Special thanks go to my internship managers: Amit Sharma (Microsoft Dynamic CRM), Guoqing Xu (Google Cloud AI), Nicholas Murphy (Google Node Storage), Maged M. Michael and Dave Watson (Facebook System Software). I could not have so many enlightened, productive and joyful industry experiences without their continuous encouragement, endless patient towards my questions, and delicately arranged events.

Infinite thanks to my Lehigh lab-mates and my friends for their continuing support. I appreciate the precious lab-survival-manual from Yujie Liu and Wenjia Ruan. I want to express my thank to Pantea Zardoshti, for whom hosting so many inspiring and cheering conversations during the past three years. She also made our lab lovely and pleasant for work. Special thanks to my friend Hongmin Wang for being so supportive when I was thinking to pursue a Ph.D. in the U.S. Sincere thank you all, my dear friends, Guotai Xiong, Mengyan Li, Zixun Yang, Wenbo Li, Haoxiang Wu, Qinghan Xue, Yujie Ji for the broad spectrum of help and care in my personal life.

Last but not least, I want to thank my parents, Lan Yi and Faxiang Zhou, for always having my back and loving me unconditionally.

Most of the work presented in this dissertation was supported in part by the National Science Foundation under Grant CAREER-1253362, and through financial support from Oracle. Any opinions, findings, and conclusions or recommendations expressed in the dissertation are my own and those of my coauthors and do not necessarily reflect the views of the National Science Foundation.

Contents

List of Tables

List of Figures

List of Algorithms

Abstract

Transactional Memory (TM) promises to provide a scalable mechanism for synchronization in concurrent programs, and to offer ease-of-use benefits to programmers. Since multiprocessor architectures have dominated CPU design, exploiting parallelism in programs is essential to achieve better performance and get further speedup as hardware upgrades increase the number of cores, instead of CPU frequency. However, lock-based synchronization is tricky to use, especially when applications have irregular or hard-to-predict memory access patterns. TM is considered as an alternative synchronization method to locks.

Emerging Non-Volatile Memory (NVM) or Persistent Memory (PM) technologies fundamentally reshape 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. However, the CPU cache and registers are expected to remain volatile. Programming directly with PM is difficult and error-prone due to the additional instrumentation required for failure atomicity. The overlap between TM and PM instrumentation is substantial, making TM an appealing programming model for PM.

Evolving computer architecture brings various challenges to programmers. Transactional memory seems to be a silver bullet to solve them all. It is surprising that there have been few examples of TM being used in "real" software, especially considering that hardware TM is beginning to see widespread availability.

In this dissertation, we conduct comprehensive experiments to identify technical challenges that prevent programmers from using TM. We demonstrate that the existing TM platform and common parallel programming models are not compatible in several scenarios, such as optimistic concurrency control with fast and immediate memory reclamation, com-

plex synchronization patterns, irrevocable or long-running operations, and data persistence. These features are requirements of real-world applications. We explain how to tailor TM in a practical and efficient way to support these features. Our target is to increase our fundamental understanding of how the concurrent threads of real-world programs interact, and to extend transactional programming models and systems, so that they are able to support these requirements efficiently and without significantly increasing programmer effort.

Chapter 1

Introduction

From small digital devices such as smartphones, tablets, and smartwatches, to personal computers and large servers, multiprocessors are replacing single processor systems in all areas of computing. The trend for multi-core processors is not surprising because manufacturers have realized it is impossible to increase clock frequencies in a single chip without overheating. Multiprocessors have became the promising architecture to leverage the increasing density of transistors enabled by Moore's law. This change also brings challenges to programmers and system developers. In old days, the speedup of systems and software depended heavily on the increasing frequency of the CPU. But now, in order to get further improvement, programmers need to exploit parallelism in programs. This requires that programmers rewrite their code into concurrent tasks, and coordinate these tasks with dedicated and correct synchronization methods.

The most well-known technology to synchronize multiple threads is the mutual exclusion lock [31]. A lock guarantees only one thread can hold it at any time. Programmers associate locks with data, and then construct code regions ("critical sections") that only access that data while holding the lock. However, lock-based programming is notoriously difficult and error prone: First, programmers have to choose the granularity of locks: implementations that use a small number of coarse grained locks are usually straightforward but limit concurrency; fine grained locking scales better but introduces latency, and could possibly result in complicated locking protocols (e.g., using fine grained locks to implement a concurrent balanced binary trees). Second, lock-based codes are not composable. Small critical sections protected by locks can not be reused to create big critical sections without knowing the implementation details. Third, locking introduces problems such as deadlock (e.g., two threads holding different locks and waiting for each other to release the lock), livelock (threads are changing status without making forward progress), priority inversion (lower priority threads holding the lock and blocking the higher priority threads), and convoying [46] (e.g., when a thread using hand-over-hand locking to traverse a linked list, it blocks all other threads from bypassing it). All in all, it is difficult to write lock-based programs that provide good performance while remaining easy to maintain and extend.

Transactional Memory (TM) [56] was first proposed more than two decades ago, as a hardware mechanism for simplifying the creation of concurrent data structures. Subsequent research has considered expanding the role of TM to a full-fledged programming model, in which programmers use coarse grained transactions as the primary means of synchronizing threads [14]. As a language extension proposed in response to difficulties faced when synchronizing programs with locks, it is easiest to understand the benefit TM offers to a programmer via comparison: Lock-based programming requires the programmer to reason about what code regions cannot run concurrently without risking a data race, and then to craft a fine grained locking methodology that allows as much concurrency as possible while also preventing incorrect concurrent executions. Transactional programming avoids the second step: regions that cannot always run concurrently without affecting correctness are marked as lexically scoped transactions, and then a run-time system, possibly accelerated by specialized hardware, runs transactions as sandboxed speculations. If a set of speculations conflict, the run-time system rolls at least one back, and typically it also takes some action to ensure that at least one will eventually complete. If concurrent speculations do not conflict, then the run-time system allows them to complete. By monitoring the behavior of speculations at a fine granularity (e.g., addresses of individual memory accesses), only true conflicts result in serialization, and thus more concurrency is possible than with locks.

The potential of TM is not restricted to parallel programming. As CPU architectures evolve through time, another essential hardware component is also improving: In the computer memory hierarchy, storage has been considered either faster but volatile (e.g., CPU registers, DRAM) or slower but durable (e.g., flash, disk). Emerging Non-volatile Memory (NVM) or Persistent Memory (PM) technologies such as 3D-Xpoint [89], PCM [67] and STT-RAM [114] fundamentally reshape 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.

Although PM can provide similar access latency, larger volume, and lower price-per-byte than DRAM, the non-volatile feature of PM brings challenges for traditional system and software designs. Since CPU caches and registers are expected to remain volatile, and store operations can be reordered and delayed as they leave the cache and reach main memory (unless directly using write-through instructions to bypass the cache), programmers need to explicitly put fence and flush instructions into their programs to control the time when updates happen in the PM. However, they cannot persist data across cache lines without additional instrumentation. The overlap between TM and PM instrumentation is substantial, which makes TM an appealing programming model for PM. This motivates investigations of transactional programming models to access PM [15, 45, 66], and lead to several concurrent persistent TM (PTM) libraries [18, 59, 69, 116].

1.1 Transactional Memory

Transactional Memory (TM) [56] was originally proposed as a hardware extension to facilitate the creation of scalable nonblocking data structures. The appeal of TM is its simplicity: a programmer need only wrap an operation inside of a language-level "transaction", and then a run-time system executes the transaction, making use of custom hardware and/or compiler-generated software instrumentation. The run-time system monitors the low-level memory accesses of transactions, and allows concurrent transactions to execute simultaneously as long as their memory accesses do not conflict (i.e., if concurrent transactions access datum D, then they may not all commit unless none write to D; otherwise, at least one must roll back and try again).

Figure 1.1 depicts a program executing under two different programming models. `CS` represents a critical section. On the left side, the lock is used to protect those critical

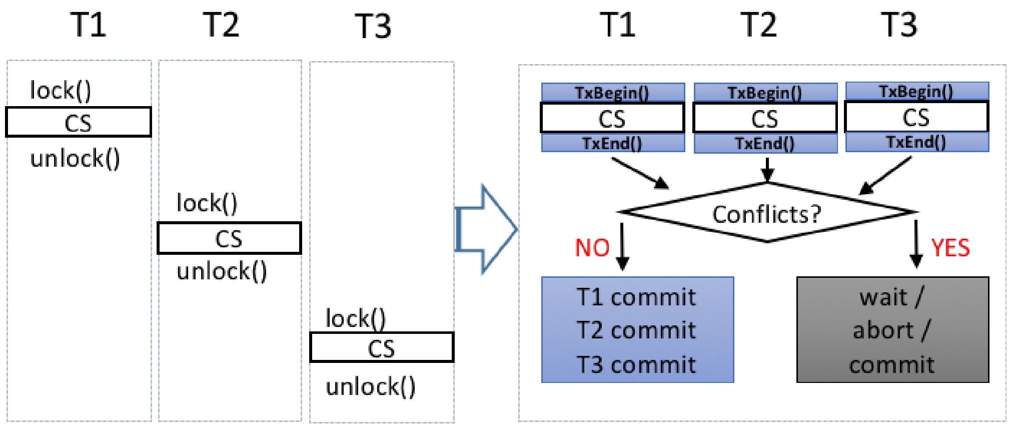


Figure 1.1: Runtime for Locks (left) and Transactional Memory (right)

sections. The right side represents the transactional execution. The lock serializes the execution of different critical sections, no matter whether they conflict or not. For transactional execution, three threads launch the speculative execution concurrently. If no memory conflict happens during the execution, three transactions can commit. Otherwise, a contention manager [48] will decide whether to delay the execution of conflicting transactions to let others commit, or explicitly abort at least one transaction so others can make progress.

1.1.1 The C++ TM Technical Specification (TMTS) Overview

In C++ [61], TM is presented as a block construct: a lexically-scoped block of code is designated as a transaction, and any early termination or exception thrown from the block will cause the TM system to commit or abort the transaction before returning control flow to an outer scope. When multiple threads attempt transactions simultaneously, their behavior should be indistinguishable from a situation in which transactions run one at a time. In practice, the run-time system will typically execute transactions speculatively, using hardware transactional memory (HTM) support or compiler instrumentation to run transactions concurrently, track their memory accesses, detect memory conflicts, and abort/retry transactions as necessary to ensure correct behavior. The implementation of TM is not, however, part of the C++ TMTS.

The C++ TMTS [61] introduces an API containing a handful of keywords to support transactional execution. We limit our discussion to three: `synchronized` and `atomic` indicate the beginning of a lexically-scoped transaction, and `transaction_safe` indicates that a function can be called from within a lexically-scoped transaction.

The primary feature of the TMTS is the notion of an `atomic` block. A program that uses `atomic` blocks will produce output that is equivalent to an execution in which the `atomic` blocks executed in some serial order, without overlap. However, under the hood, TM is expected to be used to execute those blocks concurrently. The `atomic` block is allowed to "cancel" itself. To support cancellation, the compiler must prove that an `atomic` transaction does not perform any operations that are externally visible before the transaction commits (e.g., I/O operations). Thus `atomic` transactions can only contain functions with `transaction_safe` annotations, which indicate that the functions' effects can be undone.

A `synchronized` transaction is free to perform irrevocable operations [118] (such as I/O and system calls), which cannot be undone. `Synchronized` blocks cannot cancel. When a `synhronized` transaction requires irrevocability, no concurrent transactions are allowed until the irrevocable `synchronized` block completes.

1.1.2 Implementations

TM can be implemented in hardware (HTM), software (STM), or a combination of the two (Hybrid TM).

When TM is implemented in hardware, two hardware instructions are used to indicate the boundaries of a transaction [121]. The `TxBegin` instruction creates a register checkpoint and informs the cache controller of the need to monitor memory accesses for conflicts with other threads. The `TxEnd` instruction discards the checkpoint and informs the cache controller that tracking is no longer needed. During the transaction's execution, a set of sufficient conditions on the behavior of the cache dictate whether the transaction remains valid: if the transaction reads a location, then the corresponding cache line must remain in the cache until commit; if the transaction writes a location, then the corresponding cache line must not be evicted or shared with other processors before commit. If a condition is violated, the transaction aborts, and the checkpoint is restored.

When TM is implemented in software, the compiler inserts function calls at the points where a transaction begins and ends. It also inserts a function call in place of each memory access within the transaction [91]. These function calls connect to a library that performs checkpointing, tracks the memory accesses of each transaction, and manages shared metadata for detecting transaction conflicts. For simplicity, one may think of the shared metadata as a table of readers/writer locks, which are acquired as a transaction progresses, and released at the commit point of the transaction. STM algorithms vary with regard to shared metadata implementation, protocols for detecting conflicts, and mechanisms for buffering writes.

HTM has lower latency than STM: there are no per-access function calls or explicit management of metadata (e.g., redo and undo logs). However, HTM is less flexible: there is no opportunity to introduce any nuance in how conflicts are resolved (e.g., the contention management for HTM can be summarized as `requester-wins`, which means a later transaction will always abort previous transactions if conflicts exist between them). In contrast, STM can defer conflict resolution until at least one transaction is guaranteed to succeed. Furthermore, HTM is tightly coupled with the cache controller, and thus hardware transactions cannot access more data than fits in the cache, or survive interrupts and system calls. Thus HTM does not guarantee forward progress and needs to have a fall-back strategy. For transactions that deterministically fail to complete in hardware, a common approach is to temporarily serialize all transactions, execute the failing transaction in isolation and without hardware protection, and then re-enable concurrency. This serialized path hurts performance. An alternative is to fall back to STM, known as Hybrid TM, which complicates the design but allows more concurrency.

Hybrid TM combines the advantages of both HTM and STM. There are many ways to implement Hybrid TM. One rule to design a fast Hybrid TM system is to maintain a pure-HTM execution path as often as possible [13, 22, 101] instead of falling back to the software path. To achieve the goal, Hybrid NOrec [22] replaces part of an STM implementation with HTM. The strategy not only makes the HTM path short and unlikely to abort, but also reduces intrinsic overheads when executing the STM path. The primary challenge is to make sure the synchronization in the same path and across different paths are all correct. RH-NOrec [78] takes a step further to introduce three execution paths to increase

Algorithm 1: Example instrumentation performed by GCC

```
    // Source Code (running sequentially)
1   synchronized
2       counter++

    // Source Code
3   atomic
4       counter++

    // Transformed Code
5   if TxBegin() == $UNINST$ then
6       counter++
7   else
8       $tmp$ = TxRead(&$counter$)
9       TxWrite(&$counter$, $tmp + 1$)
10  TxEnd()
```

the scalability. Hybrid Cohorts [101] separates the HTM and STM execution via different execution phases. This eliminates complex synchronization across two different paths, but can sacrifice concurrency.

1.1.3 Case study: Transactional Memory in GCC

In this section, we describe the interfaces of TM supported by GCC. Programmers or software developers can use such interfaces to employ TM for concurrency control. Moreover, we illustrate how GCC transforms the transactional code to call its TM library. We also demonstrate a practical general-purpose TM algorithm implemented by GCC, called libitm.

GCC's TM support roughly complies with the TMTS [43]. Transaction boundaries, and memory accesses made by a transaction, map to a run-time library that provides numerous TM implementations (HTM, read-only-optimized STM, general-purpose STM, and irrevocability-supporting STM). Algorithm 1 gives an example of how GCC translates a transaction that increments a shared counter.

In the example, a call to `TxBegin` starts the transaction. Its return value indicates if the transaction can skip per-access memory instrumentation (e.g., it is using HTM or running sequentially as a `synchronized` block). If not, then each individual load and store becomes a function call, specific to the current TM mechanism. Finally, `TxEnd` completes the transaction. Despite the apparent simplicity, the ABI to support this instrumentation includes more than 180 public functions: compiler optimizations introduce 40 read functions and 30 write functions, to handle read-before-write, write-after-write, and other common access patterns for 10 primitive data types. Even when HTM is used, the ABI requires two function calls and a branch in every transaction; without HTM, there is a function call on every access. For STM, `TxBegin` must also checkpoint the thread's architectural state.

Different STM algorithms vary in how they implement `TxBegin`, `TxRead`, `TxWrite`, `TxEnd` and `TxAbort`. GCC's general-purpose STM algorithm (setting ITM_DEFAULT_METHOD = ml_wt) is a privatization-safe version of TinySTM [39]. The algorithm uses a global `ownership record` (orec) table [39] as the shared metadata implementation. Every memory location maps to an entry in this table. An `Orec` represents either the latest timestamp at which the associated memory locations have been written, or that the location is currently locked. Below are the implementation details:

`TxBegin` Set the checkpoint, read the global timestamp, and perform per-thread metadata initialization.

`TxRead` Read a value directly from memory, and validate the read-set to make sure the running transaction has a consistent view [49]. Abort the transaction if the timestamp of any memory location in its read-set changed.

`TxWrite` Store the original value in a local `undo log`. Lock the `Orec` for this memory location and update the value in-place.

`TxEnd` If it is a read-only transaction, commit. Otherwise, validate the read-set, get a new timestamp, and set it as the value in the `Orec`s of all updated locations.

`TxAbort` Write back the values in `undo log`, unlock `Orec`s in the write-set. Then reset the metadata and jump back to the checkpoint.

Logs are widely used in STM algorithms to guarantee that memory updates caused by the aborted transaction should never be seen by other transactions or the non-transactional execution. In the GCC STM algorithm, an undo log is used to capture the original value before the in-place update. If a transaction aborts, the transactional effects can be undone by replaying undo log contents. An advantage of the undo log-based design is that a read can always read directly from memory. Many other STM algorithms (e.g., NOrec [24] and RingSTM [111]) employ redo logs to buffer memory updates, writing back the redo log contents only when the transaction commits. In this case, transaction aborts are cheaper, because they only need to reset the redo log. However, all read operations need to check the redo log first to get the latest value. Such redirection could be expensive. In practice, no algorithm dominates all others in all scenarios.

Another important design choice is contention management. When transactions repeatedly fail to commit, due to repeated conflicts with other threads, a TM implementation may invoke a contention manager [103], which is responsible for delaying or aborting some transactions in order to increase the likelihood that others may complete. Although contention management policies vary, most TM implementations employ serialization as a last resort: a transaction that fails too many times will request that all other transactions abort, and no new transactions commence, until it completes. Unless the workload exhibits pathological conflicts, serialization should be rare. In GCC, the default is for software transactions to serialize after 100 attempts, and hardware transactions to serialize after two. Dynamically tuning this parameter has been shown to have a significant impact on some workloads [30]. Any nontrivial amount of serialization, however, has a terrible effect on performance, particularly because serialization delays all active transactions, even those from completely unrelated parts of the program (unlike lock-based critical sections, which are partitioned by the locks that they acquire).

1.1.4 Advanced Features

Before introducing applications of TM, it is useful to review some more advanced concepts from TM literature. We first discuss the state of the art in transactional condition synchronization. Then we discuss issues related to memory consistency, which cause the "privatization" and "publication" problems, and we describe the general solutions for both.

Condition Synchronization In most cases, a lock-based critical section in a traditional concurrent program can be replaced by a transaction. This approach does not, however, provide a means for conditional synchronization. The behavior of conditional synchronization breaks atomicity by letting uncommitted transactions wait until the precondition is fulfilled by other transactions. To that end, Harris et al. proposed a special form of self abort, called `retry` [51]. When an in-flight transaction discovers that some precondition of

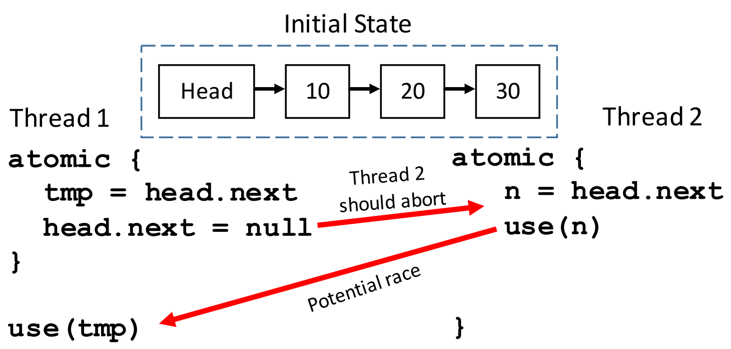


Figure 1.2: The privatization problem: If Thread 2 is executing speculatively, and is not aware that Thread 1 commits, races may occur between accesses to the node containing the value 10.

its completion does not hold, `retry` is used to immediately abort the transaction and undo its effects. Abstractly, this casts condition synchronization as a scheduling operation, with `retry` indicating that a transaction should not execute yet. Optimized implementations of `retry` track the set of locations read by the retrying transaction, and do not re-attempt the transaction until at least one of those locations is modified by another transaction. While the TMTS does not support `retry`, it can be approximated (albeit somewhat inefficiently) by using self abort from an `atomic` transaction.

Semantics and Ordering The TMTS does not separate transactional and nontransactional memory: any location can be accessed both transactionally and nontransactionally. In HTM systems, the cost of these semantics is negligible because HTM provides strong atomicity. However, nontransactional memory accesses are not tracked by STM implementations. It presents a challenge to STM implementations known as the *privatization problem* [109]: Although concurrent transactional and nontransactional access to the same location is a data race (which has undefined semantics in C++), a thread that uses a transaction to remove an object from a shared data structure and accesses it nontransactionally afterwards may conflict with another transaction that accessed the same object concurrently but is still "cleaning up". The problem is illustrated in Figure 1.2. Because Thread 2 is executing speculatively, there can be a delay between the point where Thread 1 commits, and the point where Thread 2 recognizes that it must abort. If Thread 1's transaction transitioned data to a state where transactions can no longer access it, then Thread 1 expects to be able to access the data immediately upon commit. During the window between Thread 1's commit and Thread 2's abort, simultaneous accesses to the list are unsafe.

To avoid this problem, a thread must not access a privatized object nontransactionally until every transaction that may have accessed that object has completed entirely (i.e., committed or aborted, including clean up). Since an STM cannot, in general, determine when an object is privatized, implementations typically wait after committing any writing transaction until every concurrent transaction has completed entirely; this waiting is called *quiescing*. In the GCC TM implementation, threads *quiesce* after committing. The quiesce operation involves waiting for all concurrent transactions to commit, abort *and clean up* or validate.

1.2 Applying Transactional Memory in Real-World Applications

In this section, we discuss three important applications of TM. Transactional Lock Elision (TLE) ought to be easily applied on legacy code, and it is taken for granted that using TM to replace locks could result in better scalability. Concurrent building blocks usually contain small critical sections, and they rarely have complex synchronization patterns or irrevocable operations. Thus it is promising to leverage the properties of HTM to create high performance and dedicated algorithms. Last but not the least, directly programming with Persistent Memory (PM) has been shown to be difficult and error-prone [97,104], and TM is becoming the most popular programming model for accessing data in PM due to its ability to provide atomic and durable updates of multiple objects.

1.2.1 Transactional Lock Elision

Transactional Lock Elision (TLE) is the most straightforward use of TM in real-world programs [94,96,100,102,128]. In TLE, a programmer takes as input a lock-based program with less-than-desirable performance, and replaces locks with TM, hoping that in so doing, unnecessary serialization can be avoided. The programmer in this case thinks of TM as a mechanism for achieving lock elision, and is encouraged to ignore advanced features of the

TM implementation.

We take HTM and its lock-based fall-back path as an example to illustrate how TLE works. The program executes the HTM fast path first, then falls back to a lock-based execution if the HTM cannot successfully commit. This programming model requires synchronization between the HTM and locks. Suppose a thread is executing in the lock-based critical section and delays in the middle of its execution. At this time, HTM transactions could commit if there was no synchronization between HTM and the lock-based code. Thus, concurrent execution of HTM transactions and lock-based code may introduce non-serializable behavior. To solve this problem, threads have to check the lock state at the beginning of every HTM attempt. If the lock is held by another thread, then the thread waits for the lock to be released. If the lock is available, the thread continues executing the HTM path. If the lock is acquired by another thread during the HTM execution, HTM will abort because of the change to a location (the lock) that it previously read. By doing so, the safety of the concurrent program is guaranteed. However, in order to get better performance, the execution path should remain in HTM as much as possible in order to avoid serialization.

Algorithm 2 presents the pseudocode of an HTM-based TLE retry strategy. More comprehensive applications can be found in [86]. We take Intel's HTM interface, named Restricted Transactional Memory (RTM), as an example. In lines 4, 7, and 9, RTM provides functions for `start`, explicit `abort` and successful `commit` of the transaction. The transaction begins at line 5. `Status` either indicates that the transaction successfully launched, or it contains the abort code. Line 6 to line 9 are the code executed by RTM. If at any time a transaction aborts, the execution path goes to line 11. Line 12 checks whether HTM aborted because the lock was held by another thread. Line 14 checks whether HTM aborted because of data conflicts. In line 16, excessive aborts or capacity conflicts cause the thread to execute the lock-based path (line 17–19). Otherwise, it retries the HTM path.

The retry policy in this code example is simple. There could be more complicated ones designed for specific applications. Regardless, there are two things we need to take care of: First, the algorithm needs to check the lock status right after HTM starts (line 6), this prevents lock-based code and HTM from running at the same time. Second, we need to wait

Algorithm 2: Pseudocode of Transactional Lock Elision (TLE) with Intel RTM

```
1  RetryCount ← MAX_RETRY
2  RETRY:
3      waitForLockToBeReleased()
4      status ← _xbegin()   // start the transaction by calling _xbegin()
5      if status = _XBEGIN_STARTED then
           // inside of HTM execution
           // check whether the lock has been hold
6          if check(lock) then
7              _xabort(REASON)
8          /* critical section code here */
9          _xend()  // successfully commit the transaction
10         Return
11     else
           // HTM abort
12         if status ∧ _XABORT_EXPLICIT then
13             goto RETRY // the lock is held by other thread
14         if status ∧ _XABORT_CONFLICT then
15             RetryCount ← RetryCount - 1 // data conflict
16         if status ∧ _XABORT_CAPACITY ∨ RetryCount = 0 then
               // HTM abort due to capacity or already attempted MAX times, fall back to slow path
17             acquire(lock)
18             /* critical section code here */
19             release(lock)
20             Return
21         goto RETRY
```

until the lock is available before we can try HTM (line 3). Suppose one thread is holding the lock: then all other threads that are trying HTM transactions must continuously abort due to the lock being held. Without care, they could fall back to lock-based code after reaching the maximum retry threshold, and execution would degrade to that of a lock-based program.

I/O Problems The effort to standardize TM support in C++ [3] has, to date, taken a pragmatic approach with respect to I/O. Clearly, if transactions are to replace lock-based code, then it must be possible to perform I/O operations on shared data, despite the possibility of concurrent attempts to access that same shared data. However, I/O performance has not seen much attention. In particular, it has been assumed that an I/O transaction can be statically identified by the programmer, and that it is acceptable to serialize all transactions at the time when I/O is attempted, so as to prevent concurrent accesses to the data during the I/O operation. This mechanism, broadly, is known as "irrevocability" [91, 110, 112, 118].

Irrevocability Irrevocability is a coarse-grained mechanism, which allows the execution of arbitrary operations within a transaction, even if those operations cannot be undone. Examples include accessing device registers, arbitrary system calls, communication with other threads via `volatile` and `atomic` variables, and I/O. Any student of Amdahl's law will immediately recognize global serialization of transactions as a potentially significant bottleneck. Indeed, Wang et al. inadvertently discovered as much in their exploration of transactional condition synchronization for the PARSEC benchmark suite [117]: in the "dedup" application, output by one pipeline stage eliminates all concurrency and scaling from an application whose lock-based equivalent scales well. The one-size-fits-all nature of irrevocability is costly, but its simplicity is appealing: difficult tasks are no harder with irrevocable transactions than with locks. For example, irrevocability ensures low-level atomicity and durability of output: in applications with durability constraints, it is essential that programmers control the timing of calls to `fsync`, and the atomicity of an `fsync` call with respect to preceding `write` operations, and irrevocability affords this level of control.

Deferred operations Apart from irrevocability, the only other promising approaches to transactional output rely on deferred operations. Many output operations, such as logging and error messages, can be achieved via deferred operations [14, 91, 102]. More formally, Volos et al. presented a general mechanism for deferring I/O in software transactions, via buffering and "shadow" file descriptors [115]. Unlike irrevocability, deferred output operations do not constrain concurrency. However, they suffer from two problems of their own. First, to ensure that deferral is correct, it is necessary to create an explicit copy of the data to output. This copy is in addition to any copying that occurs as part of a system call, and while it can be optimized in certain cases, it nonetheless introduces latency concerns. Furthermore, for hardware transactions, buffering may result in the working set of the transaction exceeding cache capacity, which could lead to transactions serializing. The second problem with deferred output is that real programs often care about the return value of a `write` system call. When the `write` is delayed, it seems that the continuation of the transaction must either (a) ignore the return value, or (b) be scheduled after the output, as a second transaction that is not atomic with the first.

More discussion about I/O problems and existing solutions appears in Section 1.4. In Chapter 4, we introduce our solution: `atomic deferral`, an extension to TM that allows programmers to move costly operations out of a transaction without changing program behavior. That is, the transaction and its deferred operation appear to execute as one unit from the perspective of other transactions, and the resulting execution remains serializable, for both hardware and software TM implementations.

Apart from I/O, there are many unsolved problems (e.g., third-party libraries, complex synchronization patterns) which prevent TLE from being ready to use in production. In Chapter 3, we discuss other obstacles and propose ad-hoc solutions. Our experience are gained from applying the C++ TMTS to elide locks in two real-world programs, PBZip2 and x265.

1.2.2 Facilitating the Implementation of Concurrent Data Structures

Concurrent data structures are fundamental building blocks in modern programs. In many applications, the performance of concurrent data structures determines the scalability of the program. TM facilitates the implementation of concurrent data structures for the following reasons:

- Synchronization in concurrent data structures is relatively straightforward;
- The size of critical sections fit within HTM capacity in most cases;
- Data structures rarely contain complex concurrency control like irrevocable instructions or condition variables;
- TM is particularly appealing for data structures and applications with irregular or hard-to-predict memory accesses (e.g., the rebalancing operations of a balanced binary search tree mutation), which are difficult to implement efficiently using locks.

Based on that, there are many research papers leveraging TM to craft concurrent data structures with better scalability [27, 32, 63, 75, 120].

Balanced Binary Search Trees (BST) are commonly used in a wide range of applications because they provide a logarithmic bound on search operations. In a concurrent environment, it is notoriously difficult to achieve satisfactory performance by using locks. Two reasons cause this dilemma: rebalancing and indirect deletion. Rebalancing refers to the problem where an insert or remove operation causes imbalance in the tree, and the operation must restore balance before returning in order to guarantee asymptotic complexity for future operations. Indirect deletion happens when a target node has two children. The target node has to be replaced by its successor node (based on value) and the real delete happens at a leaf node. Fine-grained locking is difficult for balanced BST because when rebalancing happens, the thread traverses upward, which is opposite to the direction the thread traverses to search the elements from root to leaf. This behavior makes lock acquires and releases difficult, because it is impossible to prevent deadlock by acquiring locks in a canonical order. Lock-free approaches are similarly difficult. Existing ways to solve the problem include relaxing the balancing conditions or using coarse-grained synchronization. Another way is to use Read-Copy-Update(RCU) [79]. However, it increases the overhead of copying existing subtrees when changes are needed, and does not allow concurrent writes. Wrapping entire BST functions in HTM transactions is one option. The scalability for this approach will largely depend on the size of the BST and the number of write operations. Siakavaras [32] proposed a new method, which combines HTM and RCU, to implement concurrent balanced BST. The algorithm achieves several appealing results: It allows concurrent update, and it introduces negligible synchronization overhead on reading operations. Although the algorithm suffers from many problems, such as live locking and memory leaks, it provides the best performance among existing algorithms for concurrent, strictly balanced BST.

In previous work [75], we also discovered that HTM can be applied in many cases to accelerate existing non-blocking data structures. For example, hardware instructions such as CAS are directly supported by x86 processors, but only for one machine word. Software emulations of CAS can support multiple words, such as k-compare-and-swap (KCAS) [76]. Wait-free KCAS is expensive but can significantly simplify the design of non blocking data structures. In this case, HTM can be applied as the simple and fast path for KCAS (atomically updating multiple memory locations). Another example is that update operations for non-blocking data structures usually are implemented by copy-on-write. Updating a bucket in nonblocking hash map [74] would require copying the whole bucket. It is expensive if the bucket contains many elements or the memory capacity is limited. HTM can provide a fast path for threads to update the bucket in-place. In both cases, synchronization between the fast path and fall-back path is straightforward.

Memory Reclamation Problems Most non-blocking concurrent data structures in research papers do not provide memory management in their implementation [11, 21, 32, 54, 75, 84, 95]. One possible reason is that memory management incurs latency. Hazard pointers [83] allow a data structure node to be reserved by a thread before it is accessed. If another threads intends to delete the node, it has to make sure no one is accessing it by checking the hazard pointer. This method introduces significant overhead because each node access will incur a write fence. Hazard pointers do not allow immediate memory reclamation when there are other threads accessing the candidate node. Epochs [41] have less latency but will delay the reclamation if one thread is blocked in an earlier epoch. It is difficult to bound the time between logical removal and physical reclamation [34], and many scalable techniques accept unbounded worst-case delay for a bounded [83] or unbounded [26] number of items. To avoid these delays, a system might fall back to complex or expensive measures when the amount of unreclaimed memory becomes too great [5, 10, 12, 19]. However, there will always remain programs whose correctness depends on memory being reclaimed immediately, hence the need for *precise* memory reclamation.

In section 1.4, we discuss the incompatibility between non-blocking techniques and precise memory management. By leveraging the immediate abort property of HTM when accessing de-allocated memory locations, we propose `revocable reclamation` to bridge the gap between them. A detailed solution and implementation are described in Chapter 2.

1.2.3 Persistent Transactional Memory

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

Due to byte-addressability and fast access, PM is likely to be accessed directly through

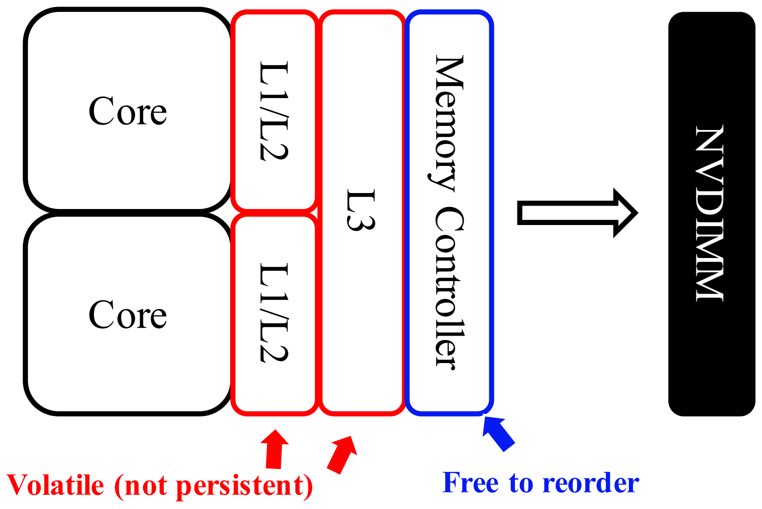


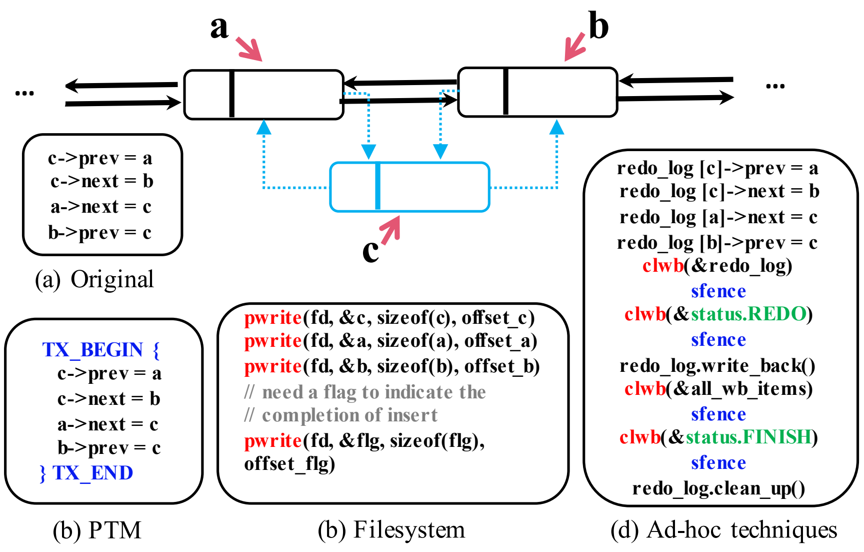
Figure 1.3: In-memory application seamlessly persists across power failures

CPU load and store instructions. Combining with non-volatility, PM can dramatically change how software persists data. As we can see in Figure 1.3, data remain volatile unless they reach persistent memory. However, the memory controller decides when and in what order the data write to PM due to hardware optimizations for writes. Without buffering and ordering methods, traditional programming models may cause the program reaching an unpredictable and unrecoverable state when a power failure happens.

Transforming a program to use PM can be nontrivial. 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 [87]. Figure 1.4 compares using a PTM library, using a file system interface, and directly programming with PM to insert a new node into a persistent doubly linked list:

- (a) represents the original volatile code on DRAM.
- (b) shows the code using PTM library. With persistent transactional memory, pro-



```
c->prev = a
c->next = b
a->next = c
b->prev = c
```

```
TX_BEGIN {
  c->prev = a
  c->next = b
  a->next = c
  b->prev = c
} TX_END
```

```
pwrite(fd, &c, sizeof(c), offset_c)
pwrite(fd, &a, sizeof(a), offset_a)
pwrite(fd, &b, sizeof(b), offset_b)
// need a flag to indicate the
// completion of insert
pwrite(fd, &flg, sizeof(flg),
offset_flg)
```

```
redo_log [c]->prev = a
redo_log [c]->next = b
redo_log [a]->next = c
redo_log [b]->prev = c
clwb(&redo_log)
sfence
clwb(&status.REDO)
sfence
redo_log.write_back()
clwb(&all_wb_items)
sfence
clwb(&status.FINISH)
sfence
redo_log.clean_up()
```

Figure 1.4: Inserting node `c` to a doubly linked list under different programming models

grammers mark the regions of code, and a run-time system tracks accesses to PM within those regions. The run-time system ensures the atomicity of transactions, using roll-forward or roll-back techniques.

- (c) represents issuing system calls to the PM-aware file systems. Implementing a file system on the PM would simplify programming on PM. Although it looks simple and familiar, many of the performance benefits of PM (such as random byte-addressable access) are lost.
- (d) demonstrates 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. To make sure the doubly linked list remains consistent across failures, four rounds of interacting with PM are necessary: Persisting the redo log; Changing the `status` to indicate all updates in the transaction can be recovered from the redo log; Writing back the content in the redo log and making them persistent; Updating the `status` to finish the operation.

For applications with irregular data access patterns, and applications that rely on ad-hoc data structures, the most promising model for interacting with PM is a transactional model. In many ways, PTM resembles software transactional memory: both need instrumentation of memory accesses. Many researchers [15, 45, 66] have implemented PTM by transforming existing STM algorithms. However, we could not find any previous research comparing the performance of different PTM algorithms and summarizing the experience via transforming them.

In section 1.4, we present preliminary findings for transforming existing STM algorithms to support data persistence. Details of our work on supporting data persistence are discussed in Chapter 5.

1.3 Dissertation Motivation

Despite significant interest and effort from the research community, TM has not yet seen widespread use in industry. The lack of adoption is particularly surprising given that (a) vendors have supported hardware TM (HTM) in commercially-available processors since 2012 [86], (b) compiler support for software TM (STM) has been present in the GCC compiler since 2012 [42], (c) a Technical Specification for using TM in C++ programs (the TMTS) was announced in 2015 [61], (d) Intel has released NVML [59], a library with a transactional interface to support programming on non-volatile memory.

To understand the problem, we conducted extensive experiments in applying TM in real-world applications. We divide applications into four categories:

- Sophisticated software such as the MySQL InnoDB engine [119], the PBZip2 parallel file compression/decompression toolkit [44], and the x265 media encoder/decoder [93].
- Primary parallel programming benchmarks such as the PARSEC Benchmark Suite [8] and the Stanford Transactional Applications for Multi-processing (STAMP) [85].
- Non-blocking and lock based concurrent data structures [36, 62, 71, 73, 74, 95].
- Benchmarks for persistent memory, such as a B+ tree microbenchmark, On-line Transaction Processing Benchmark (TPCC) [113] and Telecom Application Transaction Processing Benchmark (TATP) [107].

Some of these applications contain only small critical sections, and naively replacing locks with transactional blocks can improve performance and programmability. Even in these cases, it is possible that the performance will get worse. The costs may originate from the latency of STM, or continuous transactional aborts triggered by the workload. We can alleviate these problems by applying different STM algorithms or different retry policies. Part of the difficulty of programming with TM is insufficient infrastructure. We spent much time bypassing or reimplementing libraries in our TLE work because many standard libraries lack TM support [65].

The dissertation mainly focuses on applications that can not be transactionalized without dramatic performance degradation or significant effort. For persistent memory, the research on transactional execution time for PM is still in the early stages. Thus we evaluate the performance of existing TM algorithms with data persistence support to motive the future persistent transactional memory designs. We believe introducing these challenges and proposing techniques that tailor TM to them makes a valuable contribution to the acceptance of TM.

1.3.1 Thesis Statement

In this dissertation, we demonstrate that existing TM platforms and common parallel programming models are not compatible with each other, and even state-of-the-art TM algorithms are not optimal when transformed to support data persistence. We then propose, implement, and evaluate solutions. Our work can be summarized as diversifying the transactional programming model, extending the transactional memory API, and enhancing transactional memory implementations with more features that enable TM to fit the needs of real-world applications.

1.4 Contributions

In this section, we summarize our observations obtained from the practical experience of applying transactional memory in real-world applications. None of the applications are trivial in practice. Following each observation, we either describe our method to solve the problem or present our experience as future research guidelines so that readers can learn from them. We also present related work (if any) and compare it with the ideas we propose. Each topic is expanded in an independent chapter.

Observation 1: Non-blocking techniques are not compatible with precise memory reclamation.

In traditional lock-based code, when removing a node from the data structure, the node is guaranteed to be exclusively accessed by the removing thread. Thus it is safe to be deleted immediately. For concurrent non-blocking data structures, it is either expensive or impossible to provide information about how many threads are accessing the node when the node is unlinked from the data structure. Thus immediate reclamation of the node is not safe.

HTM not only provides hardware support for speculative execution (non-blocking), but also allows threads to access the data which already may be reclaimed by other threads (this causes HTM transactions to abort immediately). Notice that without HTM, such memory accesses lead to programs crashing (e.g., segmentation faults). This property makes immediate memory reclamation possible for non-blocking data structures implemented with HTM.

Wrapping the entire operation on a data structure in one hardware transaction would make non-blocking techniques compatible with precise memory reclamation. However, like critical sections in lock-based implementations, it is desirable to keep transactions as small as possible, in both time and space (i.e., duration and number of locations accessed): smaller transactions are less likely to conflict and abort. Furthermore, existing hardware support for transactional memory (HTM) [86] typically has capacity limits, which further makes this idea impractical.

We introduce a mechanism to link consecutive transactions by "reserving" a location at the end of a transaction and checking the reservation at the beginning of the next transaction, aborting if the location has changed since the previous transaction committed. By doing so, we link a challenge in TM to the challenge of providing memory safety for non-blocking data structures. That is, now TM must deal with the question of what happens if a reserved location is reclaimed? To avoid this problem, we introduce *revocable reservations*, which allow threads to revoke all reservations to a specified location. A subsequent transaction that checks a reservation will see that it has been revoked and therefore not attempt to access the formerly reserved location. By leveraging features of HTM, particularly the immediacy of aborts, concurrent operations are able to revoke these reservations and immediately reclaim memory, without compromising correctness. The implementation, application and evaluation of *revocable reservations* are described in Chapter 2.

Observation 2: Complex patterns of synchronization limit the applicability of a transactional programming model.

To understand the performance of TLE in real-world applications, we applied the C++ TMTS to elide locks in two real-world programs: the PBZip2 file compression tool, and the x265 video encoder/decoder. In both cases, the programs were already carefully crafted to avoid lock contention and to scale.

TLE ought to be easy: the programmer need only replace each lock-based critical section with a transaction. Unfortunately, our experience does not validate the expectation: In x265, the most important critical section was not serializable, and we could not transactionalize it without understanding several thousand lines of code, and changing the way in which threads interacted with one of the central queues in the program; In the case of PBZip2, we found that naively transactionalizing the code works. However, the performance was not competitive to the lock-based version unless we extended the C++ TMTS with a mechanism for relaxing the ordering guarantees on certain transactions.

One contribution made in Chapter 3 is proposing language-level support for transactions to dynamically disable quiescence. Prior work by Yoo et al. [122] suggests that in some workloads, quiescence can be disabled for *all* transactions. Yoo et al. also showed that in such cases, disabling quiescence for those workloads had a significant improvement on performance. Unfortunately, such an approach is not compositional: any change to the program requires whole-program analysis to determine if globally disabling quiescence remains correct. It also offers no value when transactionalizing PBZip2, because there are *few* transactions that must privatize (consumer and producer model). We studied transactional memory benchmarks and data structure benchmarks to further explore how quiescence affects the performance of TLE.

By overcoming these difficulties, Chapter 3 demonstrates the first example of the TMTS, as implemented in the GCC compiler, improving the performance of real-world code. Moreover, the improvement spanned both hardware and software implementations of TM.

Algorithm 3: Irrevocability-induced delays

```
  // Output thread
1 synchronized
2     elt.prepare(data)
3     net_send(&elt)
4     elt.sent ← true
```

```
  // Clean-up thread
1 atomic
2     if elt.sent then
3         elt.log()
4         elt.reclaim()
5     else
6         retry
```

```
  // Unrelated thread
1 atomic
2     use(other)
```

Observation 3: Serialization overheads for irrevocable and long-running operations are dramatic.

Broadly speaking, there are three causes of serialization in modern TM: irrevocability, contention management, and capacity management. Capacity management refers to situations in which a transaction's memory footprint is either too large for the TM to handle (as in the case of HTM), or causes too much logging overhead (in STM). In either case, the TM implementation may choose to serialize transactions so that the large transaction can complete efficiently.

When we transactionalized the MySQL Innodb engine and PARSEC benchmarks, we found many I/O operations in critical sections. Moreover, there are long-running computational operations in the PARSEC dedup benchmark, which become the main bottleneck for TLE. Such operations make TLE perform significantly worse than the original lock-based programs.

During profiling, we found that the composition of serialization and quiescence created unexpected latency in transactions (much worse than what was already known). Consider the code in Algorithm 3. An output thread updates an element, using some shared `data`, and then sends it over the network. The update and send are expected to be a single atomic operation. When the update communication is complete, then a clean-up thread will wake (via `retry`), log the element, and reclaim its memory. Meanwhile, some other transaction is accessing unrelated data.

When the output thread reaches line 3, GCC's STM will wait for the other transactions to complete, and its HTM support will cause the other transactions to abort. Note that the "unrelated" thread's transaction does not touch any data used by the output thread. It should neither be aborted, nor cause a delay in the output thread. However, incomplete information about which transactions might be active, and what data they might touch, necessitates conservative behavior in the TM implementation.

If *net_send* were safe to call from a transaction, we could still observe significant delays in STM. Suppose that the output thread and an unrelated thread ran concurrently. When the unrelated thread was ready to commit, quiescence would demand that it idle until the output thread completed, in order to ensure that any potential use of privatized data would not race with the output thread's upcoming reads and writes.

These sorts of delay, and the delays that arise from irrevocable execution, also manifest when transactions are serialized to ensure progress. Worse, since TM does not allow the programmer to partition transactions, any serialization or quiescence in any transaction will incur overhead proportional to the total number of active transactions at that instant. In short, I/O, long-running operations, and high contention can result in delays for all concurrent transactions.

XCall Rossbach et al. [99] were first to use locks to coordinate accesses to shared memory between transactions and non-transactional code. Volos et al. [115] followed, with a comprehensive approach to deferral focused on enabling transactional system calls (including I/O). Volos extended the OS with "sentinel" locks, which allowed software transactions to exclusively access file descriptors and other resources. Using these sentinel locks, output operations could appear to execute in the context of a transaction, but actually be deferred until after the transaction committed.

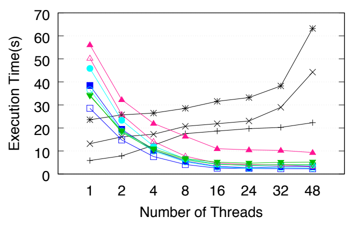
Escape actions Another approach is the use of escape actions. These may be ad-hoc [131], or formalized as open nesting [90] or transactional boosting [53]. These mechanisms provide a way to avoid logging overhead in complex operations, and also to perform I/O operations within transactions.

However, these techniques are rarely compatible with HTM [70] (an exception is the IBM POWER TM [86]). Additionally, in STM, these techniques reduce the transactional footprint, but still run in the context of an active transaction; consequently, they retain the quiescence-associated delays as we described earlier. Like traditional deferral, these techniques also require ad-hoc approaches to creating compensation actions and dealing with errors.

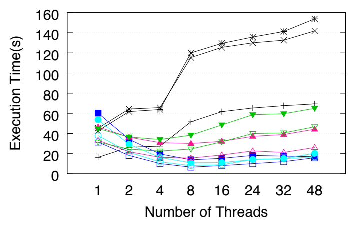
To solve the problem, we introduce *atomic deferral* in Chapter 4. Atomic deferral is a general mechanism for moving code out of transactions, but retaining serializability, through the composition of TM with two-phase locking. We successfully applied *atomic deferral* in the above applications. It not only offers an implementation-agnostic technique for performing output operations and other system calls within transactions, but also improves performance. It also enables the movement of costly operations outside of the constrained environment presented by a general-purpose TM. Our work bears resemblance to, and is inspired by, those prior works in the areas of irrevocability, deferral, transactional system calls, and escape actions. Take the XCall as an example. Volos's work is more comprehensive than ours, as it deals with a wide array of system calls, and requires less programmer effort to perform transactional I/O. On the other hand, our work is substantially simpler: it does not require a deadlock detection algorithm within the OS, or any OS modifications; it is accessible without performing system calls, and is hence compatible with HTM, and it allows the programmer to control the granularity at which operations are serialized (e.g., in the case of MySQL's file descriptor pool, where one lock abstractly covers an unbounded set of file descriptors). Our approach also makes it easier for programmers to handle timing and errors in deferred operations.

Observation 4: Although there is a mechanical transformation by which algorithms for software transactional memory can be transformed to work with persistent memory, the specifics of the programming model matter significantly for persistent transactional memory design.

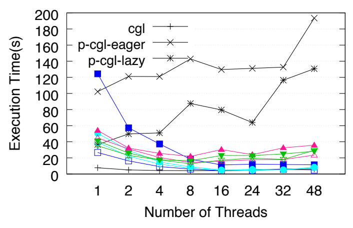
The transformation from STM to PTM is not difficult. We demonstrate two main approaches here: With *undo*, a persistent transaction (PTx) writing to location W_i must first write the current value at W_i to its undo log. Then it must persist that write (via `clwb` and a memory fence). Finally, it can 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.



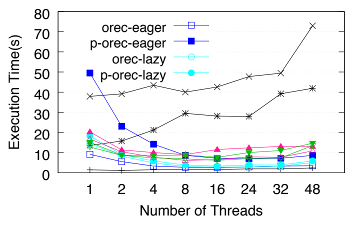
(a) genome



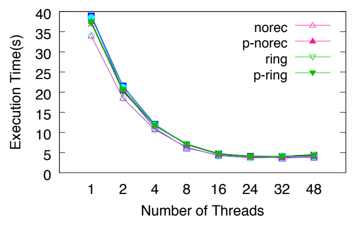
(b) Intruder



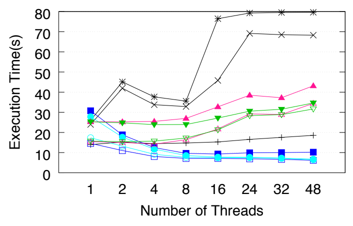
(c) kmeans (low contention)



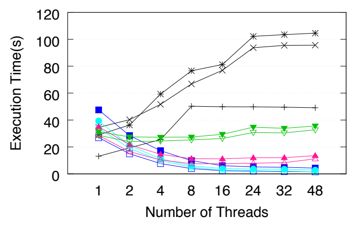
(d) kmeans (high contention)



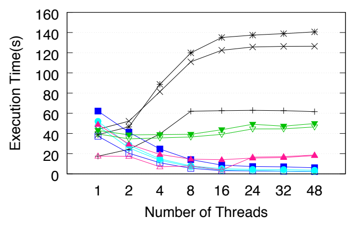
(e) labyrinth



(f) SSCA2



(g) vacation (low contention)



(h) vacation (high contention)

Figure 1.5: PTM execution on STAMP benchmarks, using the default parameters

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.

First, we show the greater potential of PTM for persistent memory compared to lock-based persistence code. We measure the impact of naively transferring STM algorithms to support data persistence 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 [39]. NOrec detects conflicts using values instead of locks [24]. Ring uses bit vectors to express the read and write sets of transactions [111]. 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 [85].

Figure 1.5 presents results for our naively implemented PTM algorithms. Our first finding is that persistence latencies discourage CGL as early as 2 threads. This is in contrast to the volatile setting, where the break-even point is 4-8 threads, and suggests that PTM is more appropriate for persistent memory than TM is for volatile memory. The second finding is that redo substantially outperforms undo, except when there is a high incidence of read-only and read-mostly transactions (genome).

There is a mechanical transformation by which algorithms for software transactional memory can be transformed to work with persistent memory. This transformation does not take into account many differences, such as the persistent and volatile programming models, the costs for flush and fence in memory instrumentations and commit, and the overhead for TM semantics.

Secondly, we observe a fundamental difference between NVM 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

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 1.1: Microbenchmark speedup for optimized PTM.

eliminate a memory fence and branch.

Table 1.1 presents preliminary speedup results for a data structure microbenchmark (RBTree, 16-bit keys, 80% lookup), with each PTM optimized according to the description. 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.

There are many optimizations we could apply to PTM exclusively, which makes the original best STM algorithms not optimal for PTM design. In chapter 5, we discuss our work supporting data persistence, which includes two programming models for PM; performance comparison between different PTM algorithms under various PM benchmarks; multiple runtime optimizations for PTM design. We are targeting at motivating the future persistent transactional memory designs.

1.5 Organization

The remainder of this dissertation is organized as follows:

- In Chapter 2, we introduce *revocable reservations*, a transactional memory mechanism to reserve locations in one transaction and check whether they are unchanged in a subsequent transaction without preventing reserved locations from being reclaimed in the interim. We describe several implementations of revocable reservations, and show how to use revocable reservations to implement lists and trees with a transactional analog to hand-over-hand locking. Our evaluation of these data structures shows that revocable reservations allow precise and immediate reclamation within transactional data structures, without sacrificing scalability or introducing excessive latency.

- In Chapter 3, we describe our experiences employing TLE in two real-world programs: the PBZip2 file compression tool, and the x265 video encoder/decoder. We discuss the obstacles we encountered, propose solutions to those obstacles, and introduce open challenges. In experiments using the GCC compiler's hardware and software support for TM, we observe that both are able to outperform the original lock-based code, potentially heralding the readiness of TM to be used more broadly for TLE, if not for truly transactional styles of programming.
- In Chapter 4, we introduce *atomic deferral*, an extension to TM that allows programmers to move long-running or irrevocable operations out of a transaction while maintaining serializability: the transaction and its deferred operation appear to execute atomically from the perspective of other transactions. Thus, programmers can adapt lock-based programs to exploit TM with relatively little effort and without sacrificing scalability by atomically deferring the problematic operations. With limited burden on programmers, our atomic deferral allows transactions to write to files, handle expensive functions, interact with the OS, and even manipulate program locks without sacrificing scalability. We demonstrate this with several use cases for atomic deferral, as well as an in-depth analysis of its use on the PARSEC dedup benchmark, where we show that atomic deferral enables TM to be competitive with well-designed lock-based code.
- In Chapter 5, we consider 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 performance. The final result is surprising: despite increased latencies due to added flush and fence instructions, required to correctly order stores to the NVM, persistent transactions achieve lower latency and higher scalability than classic memory transactions.
- Chapter 6 concludes and discusses future research directions.

Chapter 2

Supporting Precise Memory Reclamation

In this chapter, we present solutions to extend transactional memory to support precise memory reclamation on concurrent data structures without introducing performance degradation. The original work was published in “Hand-Over-Hand Transactions with Precise Memory Reclamation” at the 29th ACM Symposium on Parallelism in Algorithms and Architectures [126].

2.1 Introduction

Many operations on pointer-based data structures can be partitioned into a read-only *traversal phase* followed by an *update phase*. In the traversal phase, the operation follows a chain of nodes until it finds a node satisfying some condition. The found node is updated in the update phase. (The update phase is empty if the operation does not modify the data structure.) Some lock-based implementations reduce the size of critical sections of such operations by using *hand-over-hand locking*: An operation traverses the data structure by acquiring the lock for each node it traverses and then releasing each lock after the lock to the next node is acquired. The locks for all nodes accessed in the update phase must also be acquired, and they are not released until the end of the operation. Thus, during the traversal phase, each lock is held only while the next lock in the chain is acquired. This series of overlapping critical sections gives rise to the term hand-over-hand locking, and guarantees the atomicity of the entire operation.

Early release [55] and *elastic transactions* [40] achieve a similar effect for transactions by removing locations from a transaction's read set so that subsequent updates to the "released" locations do not cause the transaction to abort. Although they do not make transactions smaller, these techniques make transactions less likely to abort by reducing the size of their read sets. However, they cannot be applied to transactions executed by HTM, which provides no support for releasing locations.

We propose to hew more closely to hand-over-hand locking by dividing an operation's traversal phase into several read-only transactions, and executing the update phase as a single transaction. However, we cannot ensure atomicity by overlapping transactions as hand-over-hand locking overlaps critical sections. Instead, we introduce a mechanism to link consecutive transactions by "reserving" a location at the end of a transaction and checking the reservation at the beginning of the next transaction, aborting if the location has changed since the previous transaction committed. The composition of these linked transactions appears atomic because no updates are done until the final transaction.

One problem with reservations as described thus far: What happens if a reserved location is reclaimed? In that case, even reading the location in the next transaction may not be safe (e.g., it may result in a segmentation fault). We can avoid this problem by treating a reservation as a kind of hazard pointer [83], deferring the reclamation of reserved locations. Such deferral is not a significant concern for garbage-collected languages. However, in languages like C and C++, custom allocators are required, which must delay reclamation of some locations until all possible concurrent reads complete. It is difficult to bound the time between logical removal and physical reclamation [34], and many scalable techniques accept unbounded worst-case delay for a bounded [83] or unbounded [26] number of items. To avoid these delays, a system might fall back to complex or expensive measures when the amount of unreclaimed memory becomes too great [5, 10, 12, 19]. However, there will always remain programs whose correctness depends on memory being reclaimed immediately, hence the need for *precise* memory reclamation.

To avoid this problem, we introduce *revocable reservations*, which allow threads to revoke

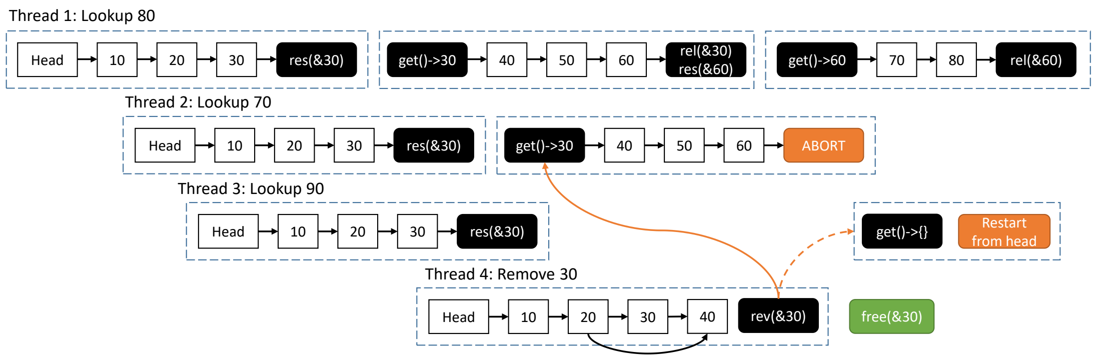


Figure 2.1: Concurrent operations on a linked list, with hand-over-hand transactions and revocable reservations. Time advances to the right. Dashed boxes indicate transactions, and filled black rectangles represent operations on the revocable reservation shared object ("res", "get", "rel", and "rev" correspond to reserving a value, getting a previously reserved value, releasing a reservation, and revoking all reservations for a specific value).

all reservations to a specified location. A subsequent transaction that checks a reservation will see that it has been revoked and therefore not attempt to access the formerly reserved location. By leveraging features of HTM, particularly the immediacy of aborts, concurrent operations are able to revoke these reservations and immediately reclaim memory, without compromising correctness.

To see how hand-over-hand transactions work with revocable reservations, consider the execution shown in Figure 2.1, in which four threads perform operations on a linked-list based set using hand-over-hand transactions. Threads T_1, T_2 and T_3 invoke *Lookup* with values 80, 70 and 90 respectively. Each of these threads executes an initial transaction that traverses the first four nodes in the list (including the head), and then commits that transaction, reserving the node N with value 30. Then T_1 and T_2 start new transactions to continue traversing the list starting from the reserved node N, and T_1 commits its transaction, releasing N and reserving a new node (with value 60). Concurrently, thread T_4 invokes *Remove*(30), finding the relevant node (N) with its first transaction. Because it wants to remove and free N, it calls *Revoke*(N) before committing, which revokes the reservations of T_2 and T_3. (By this time, T_1 has already released N and reserved a different node, so it is not affected and can complete its operation.) This revocation conflicts with T_2's use of its reservation at the beginning of its second transaction, so this transaction must be aborted. T_3 begins its second traversal transaction after T_4 commits, so when T_3 checks its reservation, it finds that its reservation has been revoked, and so retries its operation from the beginning (i.e., begins traversing from the head). Note that had T_4 removed 20 or 40 rather than 30, for example, the reservations of T_2 and T_3 would not have been revoked, so they could have continued with their operations unaffected.

We present several approaches to implementing revocable reservations, which differ in both asymptotic overhead and likelihood of conflict. We implemented and evaluated these variants in the context of singly and doubly linked lists, and unbalanced binary search trees. We found performance to be competitive with the state of the art in both transactional and nonblocking data structures, without sacrificing immediate memory reclamation.

2.2 Specification

A *revocable reservation* is a shared object that provides four methods, *Reserve*, *Get*, *Release*, and *Revoke*. Each of these operations takes a "reference" as a parameter. The object maintains a set of references for each thread. A thread adds and removes references from its set using *Reserve* and *Release* respectively. *Revoke* removes a reference from every thread's set. *Get* checks whether a reference is in the caller's set, returning **nil** if it is not. These methods must be called from within transactions, so they always appear atomic, and we can define their behavior precisely with a sequential specification, which appears in Algorithm 4.

Algorithm 4: Sequential specification for the revocable reservation shared object

```
$\mathcal{T}$ = set of all threads
$\mathcal{R}$ = set of all references
states
  $refs : \mathcal{T} \to \mathrm{Set}(\mathcal{R})$; initially $refs(t) = \{\}$ for all $t \in \mathcal{T}$
procedure Reserve($r : \mathcal{R}$) for thread $t$
  requires $r \notin refs(t)$
  $refs(t) \leftarrow refs(t) \bigcup r$
procedure Release($r : \mathcal{R}$) for thread $t$
  requires $r \in refs(t)$
  $refs(t) \leftarrow refs(t) \setminus r$
function Get($r : \mathcal{R}$) for thread $t$
  return $\begin{cases} r & \text{if } r \in refs(t), \\ \textbf{nil} & \text{if } r \notin refs(t). \end{cases}$
procedure Revoke($r : \mathcal{R}$) for thread $t$
  $\forall_{t' \in \mathcal{T}} : refs(t') \leftarrow refs(t') \setminus r$
```

Note that conflicts among operations on a revocable reservation always involve operations with same reference, one of which must be a call to *Revoke*. In particular, concurrent calls to *Reserve*, *Get*, and *Release* never conflict with each other.

The revocable reservation implementations we describe in the next section also provide a *Register* method, which is invoked by a thread before it invokes any other operation on the revocable reservation. Although not formally part of the specification, this operation is useful for maintaining the set of threads that may use the object, and thus for whom a set of references must be maintained.

2.3 Implementations

In this section, we present two families of revocable reservation implementations. The first adheres strictly to the specification in Section 2.2. The second provides a relaxed guarantee, inspired by STM: a *Get* may return **nil** even when the previously reserved reference was not revoked. To simplify the presentation, the algorithms in this section only support one reservation per thread. Extending the algorithms to support per-thread sets of reserved references is straightforward.

2.3.1 System Model

We assume a shared memory multiprocessor with coherent caches, and a TM implementation that provides a total order on transactions. The consequence of these requirements is that all writes to any single location must be ordered, and all operations performed within a transaction must appear to execute without any interleaving of transactional operations by other threads. We do not require Strong Isolation [2, 9, 106], and thus both HTM and opaque [49] STM are compatible with our algorithms. When STM is used, it must support privatization safety [81]. This is not a requirement of our algorithms, but rather a consequence of the desire for memory to be immediately reclaimed.

Algorithm 5: Basic revocable reservation algorithm (RR-FA): a linked list simulates a fully associative cache of reservations. All functions are called from an active transaction.

```
Variables ("$_t$" subscript indicates per-thread):
  $LL$   : List⟨Node⟨$\mathcal{R}$⟩⟩
  $N_t$  : Node⟨$\mathcal{R}$⟩

function Register()
    if $N_t = nil$ then
        $N_t \leftarrow$ new Node(nil)
        $LL.appendNode(N_t)$

function Reserve($r : \mathcal{R}$)
    $N_t.value \leftarrow r$

function Release()
    $N_t.value \leftarrow$ nil

function Get()
    return $N_t.value$

function Revoke($r : \mathcal{R}$)
    for $n \in LL$ do
        if $n.value = r$ then
            $n.value \leftarrow$ nil
```

2.3.2 Strict Implementations

Our first three implementations of revocable reservations resemble fully associative, direct-mapped, and set-associative caches. These implementations strictly adhere to the specification in Section 2.2. There is significant overlap among the three implementations; to save space, we only present pseudocode for the first in Algorithm 5.

Fully Associative Reservations RR-FA resembles a fully associative cache: through the *Register* method, threads "own" nodes in a linked list, and each thread t leaves its node N_t in the list from the time it first performs an operation on an RR-FA-protected list until the point where t terminates. To reserve reference r, t stores r in N_t; to release, t stores **nil** in N_t. To get its reservation, t reads from N_t. To revoke reservations on reference r, a thread traverses the list, and whenever it finds that a node stores r, it writes **nil** to that node.

To support multiple reservations per thread, we would replace the *value* field with a set. Then *Reserve* would append to the set, *Release* would remove an element from the set, and *Get* would test the set for membership. *Revoke* would remove from each thread's set, potentially increasing asymptotic complexity. All methods of the revocable reservation are performed within a transaction, which simplifies coordination of threads' accesses to these sets.

The complexity of *Revoke* is linear in the thread count. *Revoke* is also prone to low-level conflicts: From the time a revoking thread t accesses some node N_i until t's revoking transaction commits, any release or reserve by t_i will cause t's transaction to abort. On the other hand, *Reserve*, *Release*, and *Get* have $O(1)$ complexity with low constant overhead. As long as each thread's node is in a separate cache line, these methods should not experience false transaction conflicts.

Direct Mapped Reservations RR-DM (direct mapped) replaces LL with an array of unsorted, doubly linked lists, and uses a hash function to assign references to these lists. Each thread is still assigned a single node, which can be present in at most one list at any time. To revoke reservations on a reference r, a thread traverses through the list in the array position to which that reference hashes, and in any node where it observes r, it writes **nil**. To reserve a node, a thread sets the value in its node, and then inserts its node into the appropriate list. To release a reservation, the thread must set its node's value to **nil**, and should remove its node from the list. As a contention-avoiding optimization, in RR-DM, a thread can delay removing the node from its list until a subsequent transaction. The *Get* method is unchanged from RR-FA.

RR-DM reduces the common-case overhead of *Revoke*, but the worst case is unchanged. The asymptotic complexity of *Reserve* and *Release* is unchanged, but the constants are higher: each now inserts or removes from a doubly linked list. More significantly, simultaneous calls to *Reserve* and *Release* from different threads can now result in transaction conflicts on one of the reservation object's lists. To reduce contention, each list begins with a sentinel node.

Set Associative Reservations RR-SA (set associative) replaces the single array of doubly linked lists in RR-DM with A arrays. Each thread is assigned to an array via a mapping function. In *Reserve* and *Release*, the calling thread chooses the appropriate array, and then operates as in RR-DM. *Get* is unchanged from RR-DM and RR-FA. However, *Revoke* must now traverse a list in each of the A arrays, resulting in $O(A+T)$ complexity, where T is the number of threads.

RR-SA does not change the complexity of *Reserve* or *Release*, nor does it reduce the constants in these methods relative to RR-DM. However, it does reduce the likelihood that concurrent invocations of *Reserve* and *Release* will result in transaction conflicts, since these methods are unlikely to access the same lists. The cost of this improvement is additional overhead in *Revoke*, which now must traverse A lists (though the total number of entries in those lists cannot exceed the number of threads).

Correctness In the absence of concurrent calls to *Revoke*, the correctness of the cache-inspired implementations is straightforward: a thread writes a reference into a thread-specific location to reserve it, reads from that location to get it, and clears that location to release it. Thus the overall correctness of the algorithm reduces to ensuring that revoking r prevents subsequent calls to *Get* from returning r. In each algorithm, revocation traverses every possible location where a node might store r, and whenever that value is found, it is replaced with **nil**. Since every method is called from a transaction, there are no concurrent interleavings to complicate the argument.

2.3.3 Relaxed Implementations

In our relaxed implementations, after t_i reserves reference r_i, its subsequent calls to *Get* may return **nil** on account of another thread calling *Revoke*(r_j) or *Reserve*(r_j), for $r_j \neq r_i$. In exchange for this relaxation, these algorithms have smaller constant and asymptotic overhead, and less likelihood of transaction contention.

Exclusive Ownership RR-XO is inspired by the idea of ownership in STM, and is presented in Algorithm 6. Again, all functions are called from a transactional context, but now a hash function provides a many-to-one mapping from memory locations to positions in an array (OWN) of thread identifiers. Every thread is assigned a unique identifier through *Register*, starting with the value 0. The value -1 has special meaning. To reserve a reference r, a thread writes r to its thread-local variable R_t, and writes its unique identifier ID_t

Algorithm 6: Exclusive Owner Revocable Reservation algorithm (RR-XO): an array of thread IDs is used to indicate which thread currently holds a reservation for all references that hash to any given array entry.

```
Variables ("$_t$" subscript indicates per-thread):
  $OWN$  : $\mathbb{N}[]$
  $ID$   : $\mathbb{N}$     // initially 0
  $ID_t$ : $\mathbb{N}$     // initially -1
  $R_t$  : $\mathcal{R}$    // initially nil

function Register()
1     if $ID_t = -1$ then
2         $ID_t \leftarrow ID$
3         $ID \leftarrow ID + 1$

function Reserve($r : \mathcal{R}$)
4     $R_t \leftarrow r$
5     $OWN[hash(r)] \leftarrow ID_t$

function Release()
6     $R_t \leftarrow$ nil

function Get()
7     if $OWN[hash(R_t)] = ID_t$ then
8         return $R_t$
9     else
10        return nil

function Revoke($r : \mathcal{R}$)
11    $OWN[hash(r)] \leftarrow -1$
```

into the array at the position determined by *hash*(p). *Release* is a local operation, which sets R_t to **nil** but does not update the shared array of identifiers. To revoke reference r, a thread writes -1 in the shared array in the position determined by *hash*(p). To perform a *Get* of reference r, thread t must check that ID_t is still in the table at the expected position. If so, the value in R_t is returned; otherwise **nil** is returned. To support multiple reservations per thread, R_t can be replaced with a set. Since R_t is only accessed by thread t, this does not introduce new concurrency challenges.

In RR-XO, all methods run in constant time. *Release* only accesses thread-local data, and can never cause transactions to conflict. Unlike the strict algorithms, *Reserve* must write to shared memory, and two threads cannot reserve the same reference simultaneously. *Get* retains its constant-time complexity, but it must read from shared memory to determine if R_t remains valid. In exchange for these increases in overhead, *Revoke* reduces to a single constant-time write.

Shared Ownership RR-SO extends RR-XO similarly to how RR-SA extends RR-DM: it introduces multiple arrays of thread identifiers. Each thread is assigned to a specific array,

Algorithm 7: Version-based Revocable Reservation algorithm (RR-V): an array of integers is used to coordinate revocation and reservation operations.

```
Variables ("$_t$" subscript indicates per-thread):
  $V$     : $\mathbb{N}[]$
  $ID$    : $\mathbb{N}$    // initially 0
  $ID_t$  : $\mathbb{N}$    // initially -1
  $R_t$   : $\mathcal{R}$    // initially nil
  $V_t$   : $\mathbb{N}$

function Register()
1    if $ID_t = -1$ then
2        $ID_t \leftarrow ID$
3        $ID \leftarrow ID + 1$

function Reserve($r : \mathcal{R}$)
4    $R_t \leftarrow r$
5    $V_t \leftarrow V[hash(r)]$

function Release()
6    $R_t \leftarrow$ nil

function Get()
7    if $V[hash(R_t)] = V_t$ then
8        return $R_t$
9    else
10       return nil

function Revoke($r : \mathcal{R}$)
11   $V[hash(r)] \leftarrow V[hash(r)] + 1$
```

and operates identically to RR-XO for the *Get*, *Reserve*, and *Release* methods. To revoke, a thread must write -1 to the appropriate position in each of the arrays of thread identifiers. We refer to this "shared (read) ownership" variant as RR-SO.

With A ownership arrays, RR-SO increases the complexity of *Revoke* to $O(A)$. It does not change the complexity of the other operations. The main benefit is that threads will rarely cause transaction conflicts when they reserve the same reference, since they are likely to be assigned to different ownership arrays.

Versioned Reservations RR-V (Algorithm 7) uses versioning to share reservations without increasing the overhead of *Revoke*. The *OWN* array is replaced with a version array (V), which stores counters. These counters function like ownership records [28] in STM. To reserve reference r, t writes r to a thread-local field R_t, and then records the counter associated with that reference in thread-local V_t. The *Get* method checks that the value in the counter array is still V_t. To release, the thread writes **nil** to R_t, and to revoke reference r, the counter associated with r is incremented.

In RR-V, all operations have constant overhead. *Reserve* no longer writes to shared memory, and thus should not cause transaction conflicts. The use of version numbers allows any number of threads to simultaneously reserve the same reference, unlike the limits of 1 and A in RR-XO and RR-SO, respectively. *Revoke* is still $O(1)$, but must read and write to shared memory, instead of writing a constant.

Correctness Due to the use of hash functions, a revoke of r_1 could invalidate a reservation of r_2 if r_1 and r_2 hash to the same position in OWN or V. Still, we can reason about correctness in the absence of hash conflicts. In the absence of concurrency, the correctness of the three implementations follows immediately from the implementation. As in the previous subsection, the correctness of a call to *Get* in the face of a concurrent call to *Revoke* requires discussion. Calls to *Revoke* do not overwrite t's reference in R_t. However, since every call to *Get* that returns reference r accesses the metadata (in OWN or V) associated with r, and revoking r writes that metadata, the TM implementation will ensure that a conflict manifests, and a revoked r will not be used. Furthermore, RR-XO and RR-SO limit the ability of threads to concurrently hold a reservation on a reference. This limitation affects progress, but not correctness: if t_i has reserves reference r, and then thread t_j also reserves r, such that ID_i is no longer in OWN, t_i will mistake t_j's call to *Reserve* for a call to *Revoke*, but will not return an incorrect value.

2.4 Using Revocable Reservations

In this section, we briefly sketch three concurrent data structures that use revocable reservations: a singly linked list, a doubly linked list, and an unbalanced binary search tree.

2.4.1 Singly Linked List

Algorithm 8 presents a concurrent singly linked list implementation that uses revocable reservations. The behavior of this code corresponds to the illustration in Figure 2.1. We represent the common functionality of the *Insert*, *Lookup*, and *Remove* functions as the *Apply* function (lines 1–18). *Apply* takes a search key and two functions, which are run when the key is found, or is not found, respectively. Lines 19–39 provide implementations

Algorithm 8: Singly linked list with revocable reservations. Nodes consist of a value and a next pointer, and the head initially points to a sentinel node.

```
Variables:
    $RR$    : $RevocableReservation$
    $head$  : Node$\langle T \rangle$          // Head of the list
    $W$     : $\mathbb{N}$                     // Nodes to visit per transaction

function Apply($key : T, \lambda_{found}, \lambda_{notfound}$)
    while true do
        transaction
            // Initialize
            $(prev, i) \leftarrow$ (RR.Get(), 0)
            if $prev = nil$ then
                $(prev, i) \leftarrow (head, scatter(W))$
            // Traverse
            $curr \leftarrow prev.next$
            while $curr \neq nil \wedge curr.val < key \wedge i < W$ do
                $(prev, curr, i) \leftarrow (curr, curr.next, i + 1)$
            // Match
            if $curr \neq nil \wedge curr.val = key$ then
                $res \leftarrow \lambda_{found}(prev, curr)$
                RR.Release()
                return $res$
            // No Match
            if $curr = nil \vee curr.val > key$ then
                $res \leftarrow \lambda_{notfound}(prev, curr)$
                RR.Release()
                return $res$
            // Next Window
            RR.Release()
            RR.Reserve($curr$)

function Lookup($key : T$)
    $\lambda_{found} \leftarrow$ function ($prev : Node, curr : Node$)
        return true
    $\lambda_{notfound} \leftarrow$ function ($prev : Node, curr : Node$)
        return false
    return Apply($key, \lambda_{found}, \lambda_{notfound}$)

function Insert($key : T$)
    $\lambda_{found} \leftarrow$ function ($prev : Node, curr : Node$)
        return false
    $\lambda_{notfound} \leftarrow$ function ($prev : Node, curr : Node$)
        $n \leftarrow$ new Node($key$)
        $n.next \leftarrow curr$
        $prev.next \leftarrow n$
        return true
    return Apply($key, \lambda_{found}, \lambda_{notfound}$)

function Remove($key : T$)
    $\lambda_{found} \leftarrow$ function ($prev : Node, curr : Node$)
        $prev.next \leftarrow curr.next$
        RR.Revoke($curr$)
        $delete(curr)$
        return true
    $\lambda_{notfound} \leftarrow$ function ($prev : Node, curr : Node$)
        return false
    return Apply($key, \lambda_{found}, \lambda_{notfound}$)
```

of the two functions suitable for *Lookup*, *Insert*, and *Remove* operations.

Initially, the thread does not have a reservation, and the *Get* on line 3 returns **nil**. In this case, the thread will start from the head of the list, and will begin traversing forward. Typically, a thread will access a maximum of W nodes per iteration of the `while` loop. To reduce contention on revocable reservation metadata, it may be desirable to have threads shorten their initial traversal; this is achieved by the `scatter` function, which ensures that each thread's first traversal in *Apply* will be of some number between 1 and W nodes (subsequent transactions will traverse up to W nodes). During traversal, the counter i tracks the remaining nodes before a transaction must commit and start a new window. This provides the hand-over-hand behavior we desire, ensuring that traversals that progress far into the list do not conflict with modifications to nodes at the beginning of the list.

When there are no concurrent *Remove* operations, a thread will reserve its current position on line 18, commit its transaction, and then immediately get that position on line 3. If a concurrent *Remove* invalidates the traversing thread's start position (via *Revoke*), then the traversal either (a) aborts, and then discovers that its reservation is now **nil**, or (b) is between transactions, and will discover that its reservation has become **nil**. In either case, the thread will restart its search from the head of the list (line 5).

2.4.2 Doubly Linked List

Due to space constraints, we do not present full pseudocode for the doubly linked list algorithm; it is not significantly different from the singly linked list. We add a "previous" pointer to each node in the list, and during insertions and removals, we must set both the next and previous pointers; since updates are performed transactionally, the code to achieve this behavior is identical to a sequential doubly linked list. The only substantiative change relates to the removal code. In the singly linked list, the previous and current nodes are needed when unlinking a node, but in the doubly linked list, the current node suffices: its predecessor and successor are both reachable from it. This affords an optimization: rather than perform the unlinking and revoking operations from within `Apply`, a *Remove* that finds a node with matching key can reserve the node, commit the transaction, and then use a new transaction to perform both the unlinking step and the *Revoke*. If this transaction discovers that its reservation has been invalided, then it must mean that a concurrent transaction removed the same node, in which case the operation can return **false**: it can appear to happen immediately after the concurrent *Remove* operation.

The appeal of this optimization is that it avoids costly calls to *Revoke* from within traversing transactions. However, in our relaxed implementations, it is not correct: If *Get* returns **nil**, it may mean that an unrelated *Revoke* (RR-V) or even a concurrent *Reserve* (RR-XO and RR-SO) has incorrectly invalided the reservation. For these algorithms, if the final call to *Get* in a *Remove* operation returns **nil**, we must retry the entire operation.

2.4.3 Unbalanced Binary Search Tree

Lastly, we discuss an unbalanced binary search tree that uses revocable reservations. The basic pattern for the tree appears in Algorithm 9. We focus on an internal tree, and again with the list, we employ a sentinel node at the root, to simplify the case where the first element reached in a traversal is the target of a removal operation. Our implementation does not save parent pointers in tree nodes, but each node does store whether it is the left or right child of its parent.

Since the tree is not balanced, the *Lookup* and *Insert* operations are nearly identical to corresponding singly linked list code: a *Lookup* traverses until it finds its target node, and an *Insert* traverses until it either finds the value it is trying to insert, or it reaches a **nil** node, at which point it adds its value. Neither of these operations needs to revoke reservations, and both need only one reservation during their hand-over-hand traversals.

The complexity of the algorithm is in the *Remove* operation. If the value to be removed is in a leaf node, or if it is in a node that only has one child, then it can be removed in the same manner as in a singly linked list (lines 44–49). With regard to the reservation mechanism, these code paths need only revoke reservations on one node: the node to be removed. Revoking the node to remove is necessary, since another thread's search may have reserved the node that is being deleted. However, we need not revoke the node's parent or its child (if any): If a concurrent *Apply* reserved the parent, then the removal of the node cannot invalidate the *Apply*'s most recent traversal, and the removal will be detected when the operation starts its next transaction. Similarly, if the concurrent *Apply* reserved the child, then it must not be searching for the removed value, and the removal cannot have

Algorithm 9: Unbalanced binary search tree with revocable reservations. Nodes consist of a value, two child pointers, and a flag indicating whether they are the left or right child of their parent. The root is a sentinel node with only a left child.

```
Variables:
  $RR$    : $RevocableReservation$    // Shared RR object
  $root$  : Node$\langle T \rangle$              // Root of the tree
  $W$     : $\mathbb{N}$                        // Nodes to visit per transaction

function Apply($key : T, \lambda_{found}, \lambda_{notfound}$)
    while $true$ do
        transaction
            // Initialize
            $(prev, i) \leftarrow$ (RR.Get(), 0)
            if $prev = $ nil then
                $(prev, i) \leftarrow (root, scatter(W))$
            // Traverse
            if $prev.val > key$ then $curr \leftarrow prev.left$
            else $curr \leftarrow prev.right$
            while $curr \neq$ nil $\wedge\ curr.val \neq key \wedge i < W$
             do
                $(prev, i) \leftarrow (curr, i + 1)$
                if $curr.val > key$ then $curr \leftarrow curr.left$
                else $curr \leftarrow curr.right$
            // No Match
            if $curr = $ nil then
                $res \leftarrow \lambda_{notfound}(prev, curr)$
                RR.Release()
                return $res$
            // Match
            if $curr.val = key$ then
                $res \leftarrow \lambda_{found}(prev, curr)$
                RR.Release()
                return $res$
            // Next Window
            RR.Release()
            RR.Reserve($curr$)

function Lookup($key : T$)
    $\lambda_{found} \leftarrow$ function ($prev : Node, curr : Node$)
        return true
    $\lambda_{notfound} \leftarrow$ function ($prev : Node, curr : Node$)
        return false
    return Apply($key, \lambda_{found}, \lambda_{notfound}$)

function Insert($key : T$)
    $\lambda_{found} \leftarrow$ function ($prev : Node, curr : Node$)
        return false
    $\lambda_{notfound} \leftarrow$ function ($prev : Node, curr : Node$)
        $n \leftarrow$ new Node($key$)
        if $prev.val > key$ then $prev.left \leftarrow n$
        else $prev.right \leftarrow n$
        return true
    return Apply($key, \lambda_{found}, \lambda_{notfound}$)

function Remove($key : T$)
    $\lambda_{found} \leftarrow$ function ($prev : Node, curr : Node$)
        // node to delete has two children
        if $curr.left \neq$ nil $\wedge\ curr.right \neq$ nil then
            // right child's leftmost descendent, parent
            $(leftmost, parent) \leftarrow getSwapTarget(curr)$
            // get all nodes on path to leftmost
            $nodes \leftarrow \{curr \ldots leftmost\}$
            // swap value and fix leftmost's
              descendents
            $curr.val \leftarrow leftmost.val$
            $parent.left \leftarrow leftmost.right$
            // Invalidate reservations before deleting
            RR.Revoke($nodes$)
            $delete(leftmost)$
            return true
        // node to delete has < 2 children
        if $curr.left = $ nil then $child \leftarrow curr.right$
        else $child \leftarrow curr.left$
        if $curr.val < prev.val$ then  $prev.left \leftarrow child$
        else $prev.right \leftarrow child$
        RR.Revoke($curr$)
        $delete(curr)$
        return true
    $\lambda_{notfound} \leftarrow$ function ($prev : Node, curr : Node$)
        return false
    return Apply($key, \lambda_{found}, \lambda_{notfound}$)
```

affected the success of the next transaction issued by the operation, because the subtree rooted at the child has not changed.

When the value to be removed is in a node with two children, we must find a node to swap into its place. We choose the leftmost descendant of the node's right child for the swap. If that node is a leaf, it is removed from the tree by writing **nil** to its parent's left child. Otherwise, it is removed from the tree by promoting its right child to its parent's left child. We encapsulate this search as the *getSwapTarget* function on line 37. It returns the swap target and the swap target's parent. If the swap target is not a leaf, after swapping we must promote its child up one level (lines 39–40). If the swap target is a leaf, those lines write a **nil** to *parent*'s left child.

When we remove a value stored in a node with two children, we do not extract the node holding the value from the tree. Instead, we overwrite its value and then extract the leftmost descendant node of the right child from the tree. Clearly, the node that is extracted must be revoked. However, the fact that its value moves upward in the tree means that concurrent operations that have performed a *Reserve* anywhere on the path from *curr* to $leftmost$ may be invalid. Let v be the value to delete, and l be the value of the leftmost descendant of the right child. l will be written into v's node using a transaction, so there is never a time where it could appear that l is not in the tree. However, suppose a concurrent thread t_C is performing an operation with a value of l. If t_C has traversed to a point between the node holding v and the node holding l, and has reserved that node, then when it begins its next transaction(line 3), it will continue searching from a point below the point where l has been moved, and thus it will incorrectly return that l is not in the tree.

A sufficient condition to remedy this situation is for the thread removing v to perform a *Revoke* on every node in the path from v to l. The revocation causes concurrent threads on that path to restart their traversal from the root of the tree, thereby ensuring that they do not resume from a point that has become invalid.

2.5 Evaluation

To evaluate the performance of revocable reservations, we conducted a series of stress-test microbenchmarks. Our evaluation platform is a 4-core/8-thread Intel Core i7-4770 CPU running at 3.40GHz. This CPU supports Intel's TSX extensions for HTM [60]. It has 8 GB of RAM and runs a Linux 4.3 kernel. We used the TM support in the GCC 5.3.1 compiler. Results are the average of 5 trials. Across all experiments, variance was below 3%.

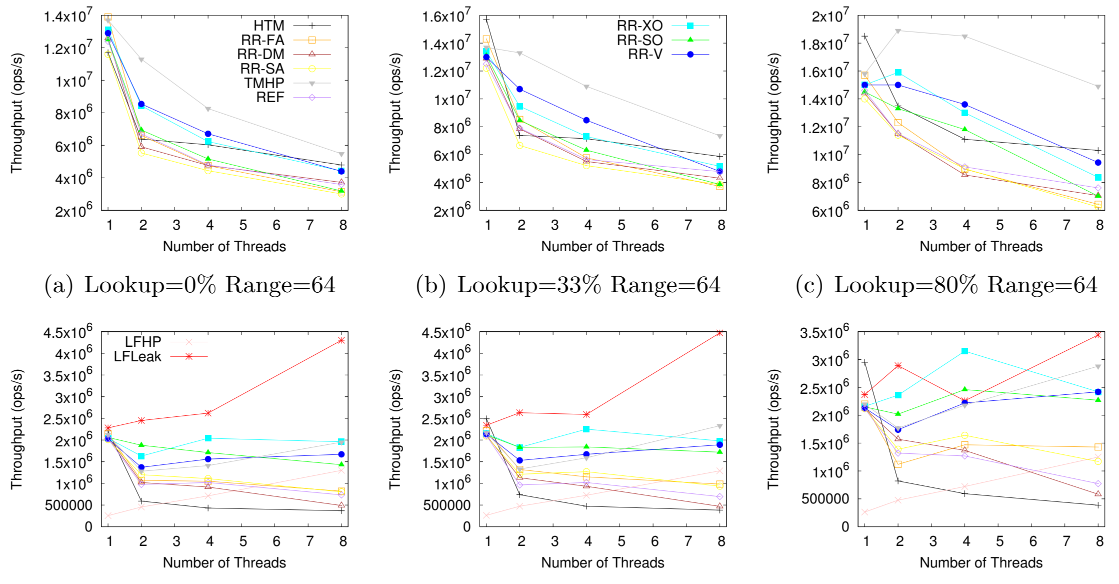
We evaluate four data structures: singly and doubly linked lists and internal and external unbalanced binary search trees. We consider the six revocable reservation implementations, and an implementation in which every data structure operation is performed within a single hardware transaction. GCC's language-level support for HTM falls back to a serial mode after hardware transactions fail twice. For the lists, this policy is adequate, but for the trees, we changed the number to 8, as it improved the performance of all implementations. For large transactions, the GCC TM implementation often must serialize these operations, and it does so after two failed attempts. In our implementations, each hand-over-hand transaction also uses GCC TM.

In the singly linked list experiments, we compare against a lock-free list based on the work of Harris [50] and Michael [82]. We provide two versions of the list: one that never reclaims memory, and one that uses hazard pointers [83]. The former approximates the best-case performance of an epoch-based allocator or garbage collector, but has no bounds on memory overheads. The latter is significantly more expensive, but can bound memory consumption more tightly. Our implementations adhere to the C++11 standard. With hazard pointers, performance is best when threads only reclaim after 64 deletions, so we report that result.

Our list experiments also include a transactional hazard pointer implementation. Algorithmically, operations are identical to those in Algorithm 8, except that memory reclamation is deferred via hazard pointers, instead of being performed immediately. We also provide a reference-counted version of this implementation. These algorithms most closely resemble work by Liu et al. [75], and benefit from only accessing hazard pointers once per internal transaction.

We consider both internal and external trees. This affords an opportunity to compare against an existing lock-free unbalanced search tree [88], taken from SynchroBench [47]. Note that this algorithm leaks memory.

We discovered two sources of sensitivity when running the experiments. First, there is a relationship between the number of threads and the optimal transaction window size (variable W in Algorithm 8). We determined the best window size for each thread count and data structure, and used these values. Second, the choice of memory allocator had a significant impact on scalability. This is a known issue with TM [4]. We performed each experiment with three allocators: the Linux system allocator, Hoard [7] and jemalloc [38].



(a) Lookup=0% Range=64 (b) Lookup=33% Range=64 (c) Lookup=80% Range=64

(d) Lookup=0% Range=1024 (e) Lookup=33% Range=1024 (f) Lookup=80% Range=1024

Figure 2.2: Singly linked list microbenchmark

In our charts, we report performance for whatever allocator was most stable: Hoard for the lists, jemalloc for the trees. We also observed significant improvements for all implementations when memory allocation and reclamation was performed outside of transactions. This suggests TM-aware allocators as a future research topic.

2.5.1 Linked Lists

Figure 2.2 presents a singly linked list benchmark. In each experiment, we pre-populate the list to a 50% filled state, and then perform 1M operations per thread. We vary the key range (6-bit or 10-bit) and operation mix (0, 33, or 80% lookups, with the remaining operations split evenly among inserts and removes). In the 6-bit experiments, we omit results for the lock-free implementations: hazard pointers (LFHP) perform the worst, and the lock-free list without any memory reclamation (LFLeak) performs best, but neither scales.

With small key ranges, transactions cannot scale: any list modification operation is likely to access a location that has recently been read by a concurrent transaction. However, hand-over-hand transactions exceed the baseline single-transaction implementation (HTM) in most cases, especially when lookups do not dominate. The experiments also show that when transactions are small, the cost of *Revoke* is important: the implementations with

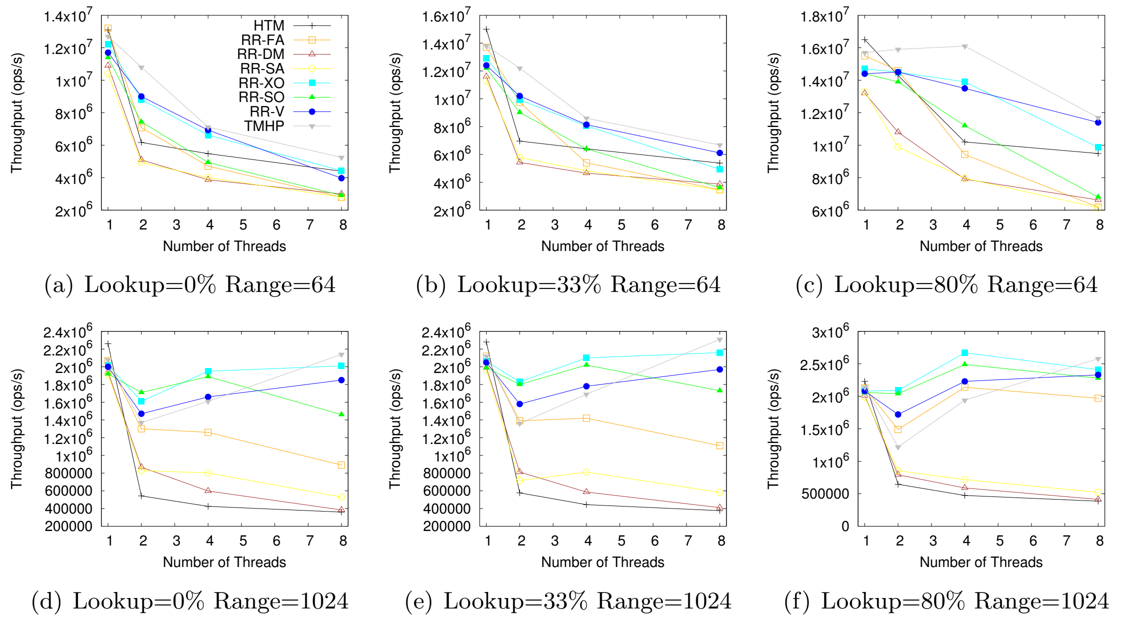
$O(1)$ *Revoke* (RR-XO, RR-SO, and RR-V) perform significantly better than those with $O(T)$ overhead (RR-FA, RR-DM, RR-SA).

Even with `scatter` optimizations, transactional reference counting (REF) performs poorly. This is despite optimizations that put reference counts in separate cache lines, read only for the first and last node of each transaction. Additionally, for small lists we can quantify the cost of precise reclamation by contrasting performance with transactional hazard pointers (TMHP). By deferring reclamation and performing reclamation in batches, reclamation exhibits more locality with allocator metadata, and hence has lower latency. Furthermore, since TMHP does not write to shared reservation metadata, it scales better for small key ranges.

With 10-bit key ranges, the bottlenecks in many of our reservation-based algorithms decrease. *Revoke* represents a smaller fraction of total execution time, and the overall performance of each reservation implementation becomes a function of the propensity for *Reserve* and *Release* to cause conflicts: RR-DM performs worst, since threads must insert and remove their nodes from doubly linked lists, and RR-SA performance varies between RR-DM and RR-FA, depending on the value of A (in the charts, $A = 8$).

For high lookup ratios, the relaxed implementations perform best. They avoid the per-access overheads of the leaky list, and their costs relative to transactional hazard pointers are amortized over a longer transaction execution. Interestingly, this workload experiences the longest reclamation delays for the hazard pointer and epoch-based reclamation strategies. Revocable reservations eliminate all reclamation delay while delivering good performance.

Figure 2.3 presents results for the doubly linked list. We no longer report reference counting, since it performs poorly, and we do not report lock-free doubly linked list performance: the only known algorithms make use of simulated multi-word compare-and-swap, and perform significantly worse than the lock-free singly linked list. The main difference between the two list algorithms is that *Remove* operations can unlink in a separate transaction from the one that finds the target node. This is beneficial for scaling, since the writing transaction is smaller. It also reduces conflicts in the reservation mechanism: if a call to *Revoke* aborts due to conflicts with a concurrent *Reserve*, the enclosing transaction retries immediately, without performing a traversal.



(a) Lookup=0% Range=64 (b) Lookup=33% Range=64 (c) Lookup=80% Range=64

(d) Lookup=0% Range=1024 (e) Lookup=33% Range=1024 (f) Lookup=80% Range=1024

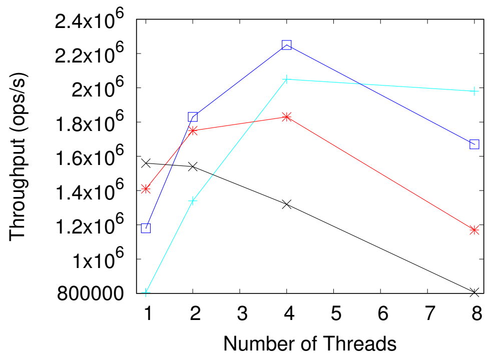
Figure 2.3: Doubly linked list microbenchmark

Overall, the doubly linked list trends are similar to the singly linked list. We observe a slightly smaller gap between the reservation mechanisms and TMHP, suggesting that the separate unlink-and-revoke transaction reduces conflicts and contention within the reservation mechanism. The remaining differences stem from TMHP's ability to batch its deferred reclamation.

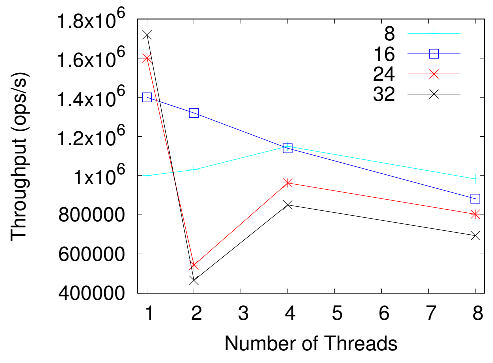
2.5.2 Window Size

Figure 2.4 shows the impact of window size on our algorithms. We highlight RR-FA and RR-XO, which are representative of the strict and relaxed techniques. The experiments use 10-bit keys and a 33% lookup ratio. On the one hand, smaller windows are less likely to result in transactional conflicts. On the other, smaller windows increase latency, since there are overheads at each transaction boundary. In addition, for RR-XO, scattering the initial window size is an important optimization, since threads will otherwise conflict when reserving nodes.

At one thread, there are no conflicts, and all transactions fit within the hardware capacity, even with a window size of 32. The advantage of a large window diminishes rapidly, especially in RR-FA, where revocation is more likely to cause false conflicts with calls to



(a) RR-XO



(b) RR-FA

Figure 2.4: Impact of window size

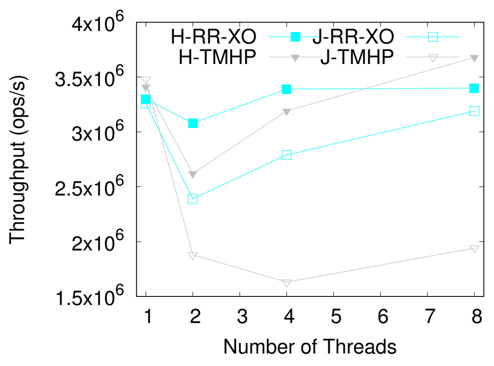
Reserve and *Release*. Since the likelihood of conflicts increases with the thread count, higher counts favor smaller windows. In addition, at 8 threads, each hardware transaction's capacity is effectively halved, since our CPU has four two-way threaded cores. Up to 4 threads, a window size of 16 is best. At 8 threads, the balance tips in favor of a window size of 8.

This experiment suggests future work in dynamic tuning of the window size. Doing so will entail hand-crafting the transactions, instead of using GCC TM support: GCC TM does not expose the fact of an abort, or its cause, to the programmer. Without that information, it is not possible to implement a data structure-specific tuning mechanism.

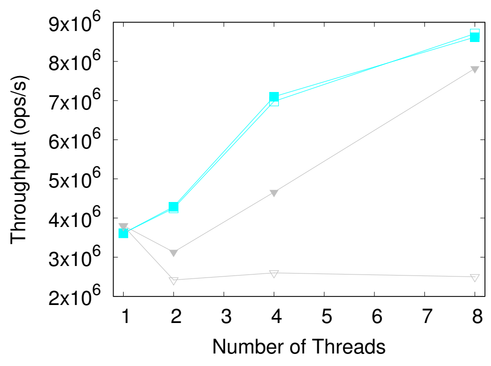
2.5.3 The Impact of Allocator Algorithm

To illustrate the impact of the memory allocator on performance, Figure 5.1 contrasts the performance of transactional hazard pointers (TMHP) with the RR-XO algorithm for a doubly linked list. We conduct two experiments, with 0% and 98% lookup ratios, on a list holding 9-bit keys. Curves prefixed with "J-" use jemalloc, and curves prefixed with "H-" use Hoard.

This case is the most extreme that we observed in all of our experiments: TMHP exposes a pathological behavior in jemalloc, resulting in poor scalability. This is especially peculiar at the 98% lookup ratio, where memory allocation and deallocation are rare. The impact of the allocator was roughly the same for our six implementations of revocable reservations on the doubly linked list.



(a) Lookup=0% Range=512



(b) Lookup=98% Range=512

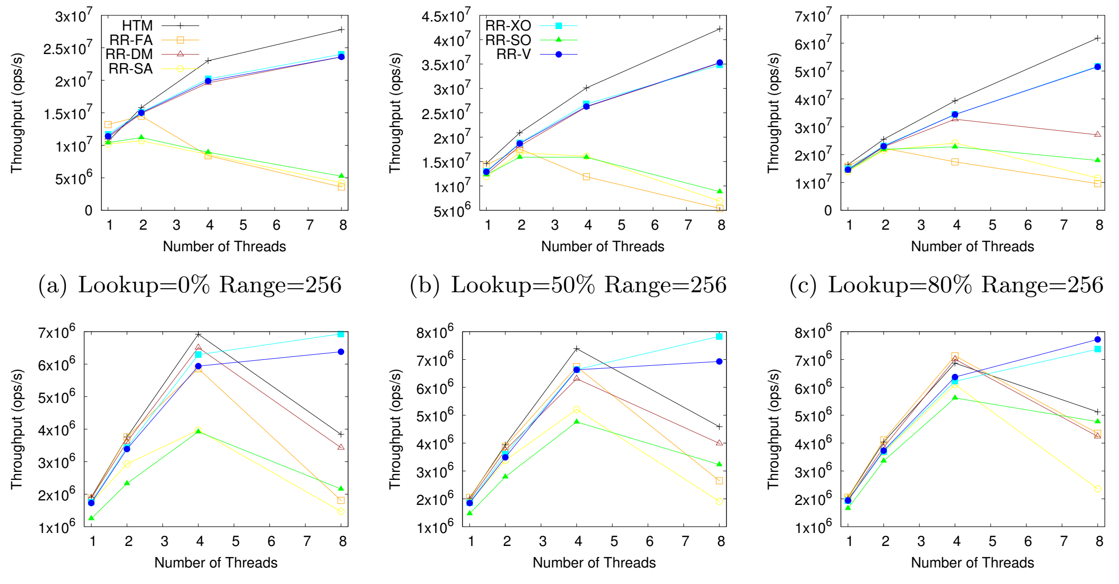
Figure 2.5: Impact of allocator

2.5.4 Unbalanced Search Trees

We now turn our attention to binary search trees. When a tree has logarithmic depth, performing operations within a single transaction should deliver good performance: even with 2M entries in the tree, the average traversal should only touch about 22 nodes, and should fit in the hardware cache. Thus the potential for reservations to improve performance is diminished. However, since the trees are not balanced, occasional large traversals are possible. Without hand-over-hand transactions, these traversals are likely to exceed cache size, and cause program-wide serialization of transactions.

Figure 2.6 presents an internal tree microbenchmark, which has mixed (0, 50, or 80%) lookups. We now consider 8-bit and 21-bit keys. In each experiment, the data structure is pre-populated with random keys to reach a 50% fill rate. We are not aware of internal trees that use hazard pointers, and the lock-free tree in SynchroBench is an external tree, so we can only compare our six algorithms against an algorithm where each data structure operation is a single transaction (HTM).

In the small (8-bit) key range experiment, our best implementations use a large window at low thread counts, and the entire operation fits in a single transaction. Thus differences relative to the HTM curve at one thread indicate the cost of reservations, and differences at higher thread counts reveal bottlenecks or contention due to the reservation mechanism. As in the list curves, we see that the relaxed implementations RR-XO and RR-V offer the best performance: all overheads are constant, and threads are unlikely to reserve the same



(a) Lookup=0% Range=256 (b) Lookup=50% Range=256 (c) Lookup=80% Range=256

(d) Lookup=0% Range=2M (e) Lookup=50% Range=2M (f) Lookup=80% Range=2M

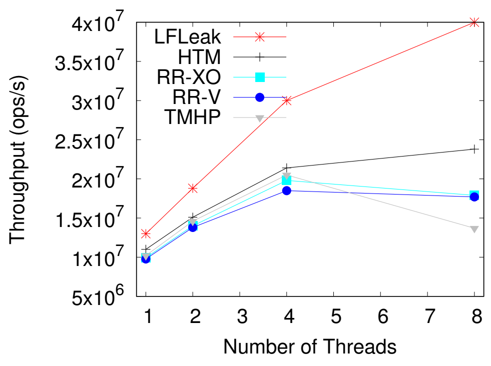
Figure 2.6: Internal binary search tree microbenchmark

nodes.

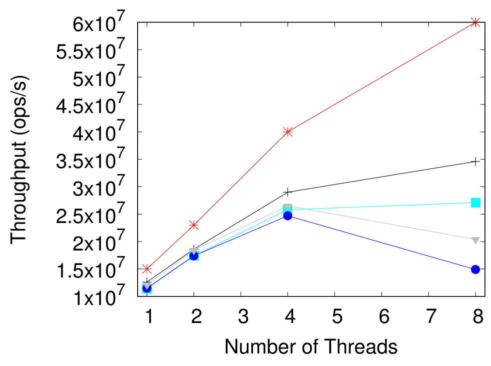
For the large key range experiment, there is an inflection point after 4 threads, due to hardware multithreading. The HTM algorithm exceeds the cache, and serializes. In contrast, reservation-based algorithms can use a smaller window size and complete. However, only RR-XO and RR-V scale well. The cause of poor performance in the other algorithms is the overhead of calling *Revoke* for multiple references. Recall that in the tree, a removal must revoke reservations on the path between the found node and the node whose value will be swapped. In RR-SA and RR-SO, each call to *Revoke* visits A locations ($A = 8$). In RR-DM, k *Revoke* operations result in $O(k)$ unique accesses.

Lastly, Figure 2.7 presents the performance of an external binary search tree that uses revocable reservations. We also include results for a nonblocking external tree [88] that leaks memory (LFLeak), and a tree that uses hand-over-hand transactions and hazard pointers (TMHP).

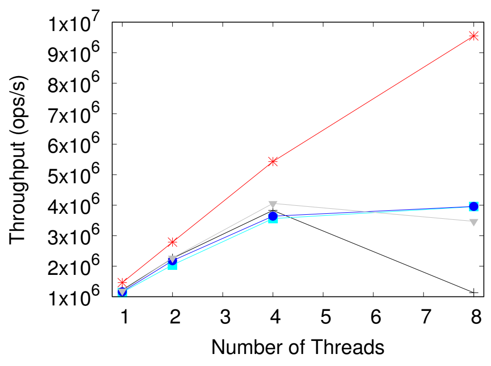
With no memory reclamation overheads, no overheads on transaction boundaries, and a highly optimized lock-free implementation, LFLeak performs significantly better at all thread levels, and scales linearly. The difference between LFLeak and the other algorithms makes it difficult to observe their performance, so we omitted the weaker-performing reser-



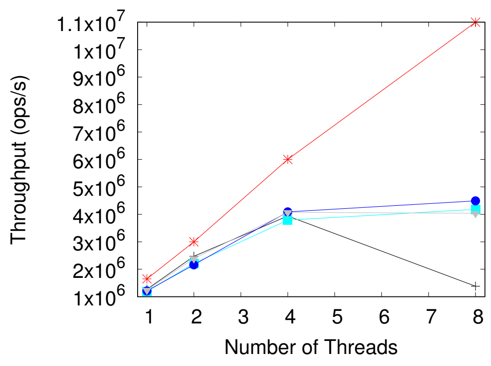
(a) Lookup=0% Range=256



(b) Lookup=50% Range=256



(c) Lookup=0% Range=2M



(d) Lookup=50% Range=2M

Figure 2.7: External binary search tree microbenchmark

vation algorithms. The best reservation algorithms, RR-XO and RR-V, exhibit the same relationship to HTM as in the internal tree experiments. In addition, we see that the performance of TMHP is almost indistinguishable from these algorithms. In the absence of multiple calls to *Revoke*, the other reservation algorithms performed better than in the internal tree, though still below RR-XO and RR-V.

2.6 Conclusions

This chapter introduced revocable reservations, which allow concurrent data structure designers to make use of hand-over-hand transactions and still immediately reclaim memory. We presented six implementations of revocable reservations, three of which allow for spurious revocation of reserved references. We also presented concurrent list and tree data structures that employ revocable reservations and hand-over-hand transactions. We found that the relaxed algorithms performed best, often enabling hand-over-hand transactions to provide better scalability and resilience than coarse-grained transactions, with minimal cost relative to hazard pointer implementations that sacrifice immediate reclamation.

Overall, we found the use of revocable reservations to be straightforward, and their application to lists and unbalanced trees did not require much data structure redesign. Based on this experience, we believe they will be a valuable technique for other concurrent data structures, such as balanced trees and hash tables, for which existing scalable algorithms rely on deferred memory reclamation.

Chapter 3

Supporting Complex Patterns of Synchronization

In this chapter, we describe our experiences employing Transactional Lock Elision (TLE) in two real-world applications: the PBZip2 file compression tool, and the x265 video encoder/decoder. The original work was published in "Practical Experience with Transactional Lock Elision" at the 46th International Conference on Parallel Processing [129].

3.1 Introduction

In this chapter, we focus on TLE. TLE plays an important role in the long-term adoption of TM: until it is shown to be useful, there is little incentive for HTM and STM implementations to provide support for the advanced features needed by transactional programming models, such as self-abort [51, 91], deferred actions [127], OS support for transactional system calls [115], escape actions and open nesting [53, 90, 131], and nuanced contention management [103].

The primary vehicles for our study are two open-source applications: the PBZip2 parallel file compression/decompression toolkit [44], and the x265 media encoder/decoder [93]. Both programs are large, robust, and mature, with many locks, non-trivial input and output operations, and subtle protocols for sharing memory. Neither was designed with TM in mind, and both have been heavily optimized. In both cases, we explore whether it is possible for TLE to improve program performance. Our TLE versions of these benchmarks were produced via a hand instrumentation that applied the C++ TMTS [61]. In this manner, we believe our findings have the most potential to generalize, both to other TLE endeavors, and to efforts to improve the TMTS. Since they conform to the TMTS, it is also relatively easy to test these benchmarks with a variety of HTM and STM implementations.

While there are many small discoveries reported in this chapter, two experiences stand out. In the case of PBZip2, we found that both HTM and STM were able to improve performance relative to locks, but STM could only do so when the C++ TMTS was extended with a mechanism for relaxing the ordering guarantees on certain transactions. We propose a dynamic mechanism for achieving this relaxation, which is reminiscent of the `memory_order_relaxed` feature of the C++ memory model [6]. In x265, we found that the most significant critical section in the program did not obey two-phase locking, and was incompatible with TLE. A simple refactoring solved the problem, but raises an important question: is two-phase locking a necessary or sufficient condition when using TLE? As with PBZip2, both HTM and STM were able to improve the performance of x265, relative to the original lock-based program. Our peak improvement was 9%.

In Section 3.2, we describe the high-level behavior of PBZip2 and x265. Section 3.3 discusses the quiescence mechanism used by GCC's STM to ensure lock-based semantics, and presents situations in which quiescence overheads are avoidable. Section 3.4 discusses problematic lock-based code in x265, which is not immediately transactionalizable. Section 3.5 briefly discusses additional challenges these applications present, and describes our workarounds. Section 3.6 presents performance results for the two applications, and shows that with modest effort, TM is able to outperform the original lock-based code. Section 3.7 concludes.

3.2 PBZip2 and x265

Our study focuses on the transactional elision of locks in two programs, described below.

PBZip2 PBZip2 is a parallel version of the BZip2 file compression algorithm. Whereas the original BZip2 algorithm takes as input a complete file stream, and then compresses it, PBZip2 splits a file into multiple streams, and compresses those streams in parallel. The user is able to specify the size of each stream, to balance the amount of work per thread with the number of threads. Internally, the program follows a serial-parallel-serial pipeline pattern. A producer thread creates stream descriptors and passes them to consumer threads. Consumer threads compress or decompress streams, based on the descriptors they pull from the queue. Their output is passed to the serial write stage, which produces the output file by assembling its input in the correct order.

The implementation employs six locks and six condition variables. The critical sections are friendly to transactionalization, in that they are small and do not make system calls. The input, output, compression, and decompression operations are performed outside of critical sections. The main source of contention is for the locks protecting the inter-stage queues.

x265 x265 encodes and decodes video streams and images to and from the HEVC/H265 compression format. The encoding and decoding algorithms divide each frame into sequences of macro-blocks called "slices", which are passed to worker threads. Each slice consists of a sequence of CTUs (Coding Tree Units), which can be encoded by making reference to another unit in the same frame (intra-picture prediction) or in another frame (inter-picture prediction). The output frame is stored in a decoded frame buffer to be used for the prediction of other frames.

x265 takes advantage of as much parallelism as possible to improve performance. With frame-level parallelism, independent frames can be encoded simultaneously. Each video frame is also divided into "slides", which can be independently processed. Additional parallelism is achieved via a wavefront algorithm used in individual frames. Within each CTU, CUs (Coding Units) distribute their analysis work to threads, which provide CU-level parallelism. At the lowest level, vector instructions can be enabled to process adjacent pixels of a frame. To manage parallelism more abstractly, x265 includes wrappers over traditional synchronization objects. These include a thread pool and a condition variable wrapper, as well as a wrapper around mutex locks. A depiction of the wavefront and row decomposition appears in Figure 3.1.

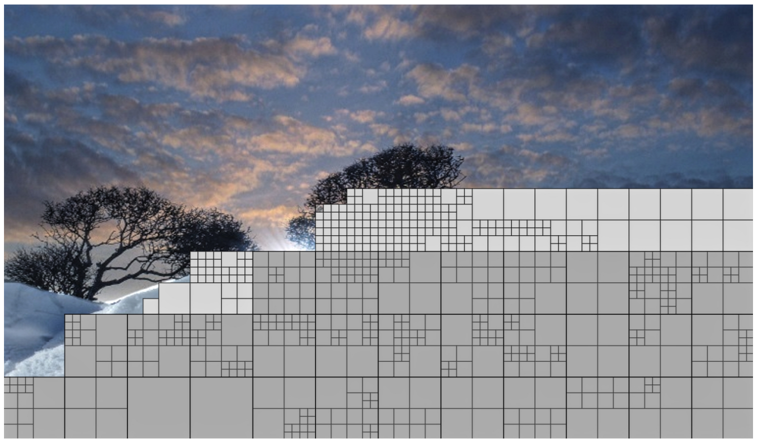


Figure 3.1: HEVC wavefront parallel processing [93].

There are three main lock objects in x265:

- The lookahead lock prevents concurrent access to shared input and output queues of frames; in essence, it mediates inter-frame parallelism.
- The CTURows lock is used by the wavefront processing algorithm to mediate communication from a completed CTU to the CTUs that depend on it.
- The EncoderRow lock protects shared data when multiple threads work on the same row within a slice of a frame.

There are additional locks, to include the “bonded task group” lock, which governs the allocation of jobs to threads; a “parallel motion estimation” lock, which protects searches for reference frames during motion searches; and a “cost lock”, which protects performance metadata and metrics.

3.3 Quiescence and Lock Elision

Quiescence ensures that whenever a thread commits a transaction, it waits until all concurrent threads commit or abort *and clean up* before the thread is permitted to execute the code that follows the transaction. While some STM algorithms have quiescence support built-in [24, 29, 68, 111], the STM algorithm in GCC does not, and requires a committing transaction to execute code similar in spirit to a user-space RCU Epoch [26].

The C++ TMTS uses the notion of transactional sequential consistency [23] to describe a memory model that includes locks, `atomic` variables, and transactions. The memory model requires a global total order on synchronization operations, program order on synchronization operations within a thread, and a global total order on all transactions. The model does not handle explicit self-abort within transactions, which Shpeisman et al. previously showed to be a source of significant additional complexity [105].

In this memory model, quiescence ensures that when a transaction transitions data between shared and thread-private states, concurrent transactions accessing that data do not race with legal nontransactional accesses. In HTM, such accesses are not possible. In STM, they can result from delayed undo or write-back operations. In C++, publication safety (i.e., ensuring the absence of races when transitioning data to a state in which it can be accessed by transactions) is guaranteed for race-free programs, and thus quiescence is only required for privatization safety (i.e., ensuring the absence of races when transitioning data to a state in which it is no longer accessed by transactions) among STM transactions.

3.3.1 Problems with TLE and Quiescence

When the C++ TMTS is used to achieve lock elision, it entails a form of lock erasure: whereas the original program may contain many locks that disjoint regions of memory, all elided locks become transactions over a single shared heap. As an example, if a program contained a queue protected by lock L_1, and a stack protected by lock L_2, the transactional version would contain one class of transactions used to protect both the queue and the stack. As a global synchronization operation, quiescence forces a transaction on the stack to delay after it commits, waiting until any concurrent transaction touching the queue *or* the stack has committed or aborted. Since the locks are erased during TMTS-based transactional lock elision, the granularity of quiescence becomes unnecessarily coarse.

In addition, quiescence has the potential to result in transaction congestion. Consider two transactions, T_1 and T_2, each of which takes U units of time to complete. Suppose that T_1 begins at time 0, and T_2 begins at time $U/2$. When T_1 completes, it must wait for T_2 to commit or abort. This waiting does not increase the likelihood of T_2 aborting, because T_1 has already committed. However, if T_1 and T_2 execute in tight loops, then after one

Algorithm 10: Proxy privatization: When the privatizer's transaction commits and invalidates the vector update thread's transaction, the potential race is between line 4 of the *proxy thread* and the rollback of the vector thread's transaction.

```
// Vector update thread
1 atomic
2     for $k \in 0 \ldots size$ do
3         $update(vec[k])$
```

```
// Privatizer thread
1 atomic
2     $msg \leftarrow vec$
3     $vec \leftarrow$ null
```

```
// Proxy thread
1 atomic
2     if $msg = null$ then
3         retry
4 $use(msg)$
```

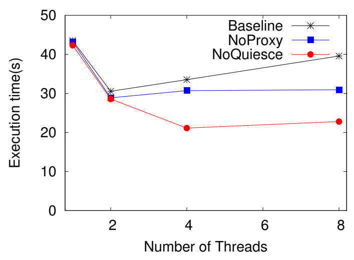
iteration, the interval between when the next T_1 and next T_2 begin is likely to be less than $U/2$. Quiescence in T_1 results in future congestion.

While HTM does not incur quiescence overheads, STM must. Furthermore, these overheads are growing increasingly expensive. Prior to 2016, GCC's STM implementation performed quiescence after every writing transaction. This, however, does not support proxy privatization (see, e.g., Algorithm 10). Since 2016, *every* STM transaction quiesces after committing in GCC.

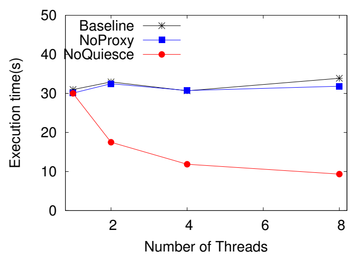
To show the overheads caused by unnecessary quiescence in GCC, we apply various levels of quiescence-avoidance on the STAMP TM benchmark suite [85].

Figure 3.2 presents performance for 8 common configurations of STAMP. We omit the "Bayes" benchmark, which has nondeterministic behavior, and the "labyrinth" benchmark, whose transactions are too infrequent to affect performance. We compare three levels of quiescence avoidance for GCC's STM. The Baseline version quiesces on every transaction, even read-only transactions. It represents the default behavior of GCC STM after 2016. However, it is overkill, since there is no proxy privatization in STAMP. NoProxy avoids quiescence for transactions that do not perform writes. This is equivalent to the pre-2016 behavior of GCC. Finally, NoQuiesce removes quiescence overheads for all transactions.

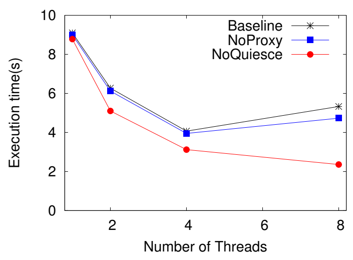
The benefit of quiescence avoidance is dramatic, and increases with thread count. NoQuiesce always outperforms the baseline, and only Yada fails to show separation between NoProxy and NoQuiesce. This lack of separation is because read-only transactions were the only opportunity we identified to avoid quiescence in Yada. For all benchmarks, we see that unnecessary quiescence is a dominant overhead for STM. When we dynamically remove quiescence from transactions, benchmarks that did not scale become scalable, and



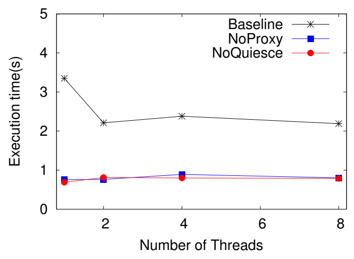
(a) Genome



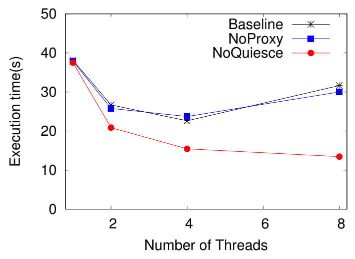
(b) Intruder



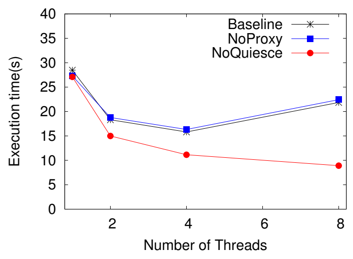
(c) SSCA2



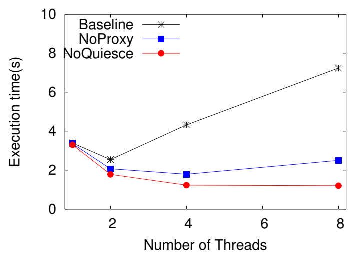
(d) Yada



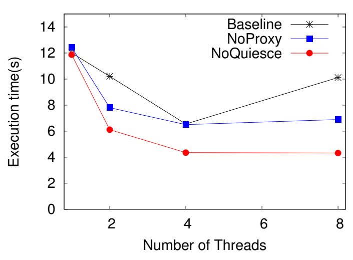
(e) Vacation (high contention)



(f) Vacation (low contention)



(g) KMeans (high contention)



(h) KMeans (low contention)

Figure 3.2: STAMP performance with varying levels of quiescence-avoidance.

benchmarks that appeared to run out of concurrency instead scale to the full core and thread count of the machine.

3.3.2 Programatically Avoiding Quiescence

Alternatives to maximal quiescence draw from the observation that privatizing in C++ always involves at least one transaction. Two sufficient criteria arise: (a) require quiescence in the transaction that transitions the data to a nontransactional state, or (b) require quiescence in the last transaction executed by a thread before it accesses data nontransactionally. In practice, neither approach is straightforward: When transactions are nested, or have complex memory access patterns, it is not feasible to expect programmers to know which transactions privatize. Indeed, some transactions might only privatize under certain circumstances (consider a consumer who reads from a producer/consumer queue: if the queue is empty, there is no data to privatize). Furthermore, proxy privatization (itself a reasonable idiom, e.g., for a producer/consumer workload with per-consumer queues) cannot support marking privatizing transactions (criteria "a" above) without added overhead.[1]

In the proxy privatization case, a writer privatizes the data, and then a transaction in another thread executes before the data is accessed nontransactionally. In the non-proxy privatization case, a writer is the last transaction to modify the data before nontransactional access. In both cases, it is easier to achieve the second sufficient criteria: we could mark the last transaction before the private access.

With complex control flows, nested transactions, and separate compilation, we do not believe that programmers will be able to correctly identify the minimal set of transactions that require quiescence. However, one simple heuristic can capture a fair portion of the times when quiescence is not needed: if transactions T_1 and T_2 are executed sequentially by the same thread, then T_1 requires quiescence only if the thread's memory accesses between T_1 and T_2 might include data that was accessible by transactions prior to T_1's execution.

Prior work by Yoo et al. [122] suggests that in some workloads, quiescence can be disabled for *all* transactions. Yoo et al. also showed that in such cases, disabling quiescence for those workloads had a significant improvement on performance. Unfortunately, such an approach is not compositional: any change to the program requires whole-program analysis to determine if globally disabling quiescence remains correct. It also offers no value when *few* transactions privatize.

We propose a new TM API function: `TM.NoQuiesce`. When called within a transaction, this function indicates that the transaction should not quiesce after it commits. The call has no meaning for strongly isolated HTM implementations, or for STM implementations that do not require quiescence. The STM implementation is also free to ignore the API call. Two examples are when the transaction making the call is nested within another transaction, in which case its programmer is unlikely to know the privatization behavior of the parent transaction, and when the transaction `free`s memory (certain TM-aware memory managers require quiescence before returning memory to the operating system [57]).

[1]The issue is that existing STM algorithms would require writing transactions to quiesce *before* releasing ownership of locations.

Algorithm 11: Producer/consumer workload

```
// Producer thread
while true do
    atomic
        if $\neg c.full()$ then
            $c.insert(produce())$
        $TM.NoQuiesce()$
```

```
// Consumer thread
while true do
    atomic
        if $\neg c.empty()$ then
            $tmp \leftarrow c.get()$
        else
            $tmp \leftarrow$ nil
            $TM.NoQuiesce()$
    if $tmp \neq nil$ then
        $use(tmp)$
```

Algorithm 11 demonstrates the use of `TM.NoQuiesce`. The producer need never quiesce, since it never privatizes data, and the consumer need only quiesce if it succeeds in extracting an element from the collection (c). This example offers additional benefits: for single-producer, multi-consumer workloads, the producer is more likely to be the bottleneck, and avoids quiescence. Furthermore, when a consumer finds no work, it does not wait unnecessarily before looking again.

3.3.3 Pitfalls

`TM.NoQuiesce` has the potential to significantly increase scalability: quiescence can introduce cache misses linear in the number of threads, to determine when each thread is no longer at risk of racing with a subsequent nontransactional access; and long-running transactions can lead to a quiescence operation blocking unrelated threads' committed transactions for the duration of the long-running operation. However, when used incorrectly, `TM.NoQuiesce` transforms an otherwise correct program into a racy program.

The problem is that Transactional Sequential Consistency demands a global total order among transactions, and the transitive closure of transaction order and program order must establish happens-before relations. Quiescence delays committing transactions long enough to be certain of transitivity with program order across threads. In contrast, `TM.NoQuiesce` asserts that data and/or control-flow dependencies within a specific transaction, or among specific dynamic instances of transactions within the thread, are enough to provide happens-before. When the assertion is faulty, the program becomes erroneous because there are accesses to shared memory that may not be compatible with any global total order on transactions. We expect these errors to be easy to identify and fix using transactional race detectors. For example, T-Rex [64] is able to identify all races that arise when a TM library fails to provide privatization safety. Extending T-Rex to understand implicitly privatization-safe STM with selective disabling of privatization appears to be straightforward.

3.4 Two Phase Locking and x265

Part of the appeal of TM is that it ought to be *easier* than using fine grained locks. By extension, TLE ought to be easy: the programmer need only replace each lock-based critical section with a transaction. Past work has revealed this task to be laborious, but not thought-intensive. For example, in transactional memcached [102], the effort was in identifying which transactions caused unnecessary serialization, and then creating transaction-safe variants of the standard library functions that were responsible for the serialization.

In memcached, critical sections obeyed two-phase locking [37], by ensuring that all lock acquires preceded all lock releases within each critical section. The only complication was when a critical section also read a C++ `atomic` variable. The solution in that work was to model these accesses as mini-transactions, and subsume them within the transaction that replaced the critical section. In memcached, critical section behavior did not depend on an atomic variable changing between accesses, and hence this was safe.

The transactionalization of PBZip2 mirrored past work with memcached: lock-based critical sections were replaced with transactions, and as appropriate, functions were annotated for transaction safety. During the transactionalization of x265, we found a situation in which the pattern of lock acquisitions and releases was clearly not two-phase locking, and hence the program could not be naïvely transactionalized: if the outer lock was replaced with a transaction, the program could not complete.

Fortunately, the violation of two-phase locking in x265 was fixable. The specific behavior was that a producer thread would acquire a lock on its output queue, then produce elements, then unlock the queue (Algorithm 12). During element production, several smaller critical sections ran, with inter-thread communication between the critical sections. These critical sections could not be subsumed by the transaction on the output queue. Our solution was to

Algorithm 12: A non-serializable critical section in x265.

```
   // Lock held during produce stage
   // of pipeline, to ensure ordering
1  out_queue.lock()
2  element ← new queue_node()
3  out_queue.enqueue(element)
4  produce(element)
5  out_queue.unlock()
   // Lock acquired during final stage of pipeline
6  out_queue.lock()
7  e ← out_queue.dequeue()
8  out_queue.unlock()
9  use(e)

   // First step of produce uses m_lock
   produce(element)
10     m_lock.lock()
11     use(element)
12     m_lock.unlock()
       ...
13     produce2()
14     wait()

   // Second step of produce also uses m_lock
   produce2(element)
       ...
15     m_lock.lock()
16     use2(element)
17     m_lock.unlock()
       ...
```

Algorithm 13: A `ready` flag avoids lock nesting, facilitating transactionalization.

```
   // Lock no longer held during produce stage
1  out_queue.lock()
2  element ← new queue_node()
3  out_queue.enqueue(element)
4  element.ready ← false
5  out_queue.unlock()
6  produce(element)
7  out_queue.lock()
8  element.ready ← true
9  out_queue.unlock()

   // Lock acquired during final stage of pipeline
10 e ← nil
11 out_queue.lock()
12 if out_queue.peek().ready then
13     e ← out_queue.dequeue()
14 out_queue.unlock()
15 if e = nil then goto10
16 use(e)
```

embed a ready flag in each queue node, rather than keep the queue locked for the duration of the program (Algorithm 13). Across several workload configurations and thread counts, we confirmed that this modification did not affect performance. With the change in place, each of the critical sections could be separately transactionalized.

Our experience introduces two research questions, which we leave as future work:

- Can it be proven that naïve transactionalization is safe for critical sections that obey two-phase locking?
- Under what conditions will naïve transactionalization of non-two-phase locking code remain safe?

To the best of our knowledge, no prior work has dealt with these problems, or explored transactionalization of programs that did not obey two-phase locking.

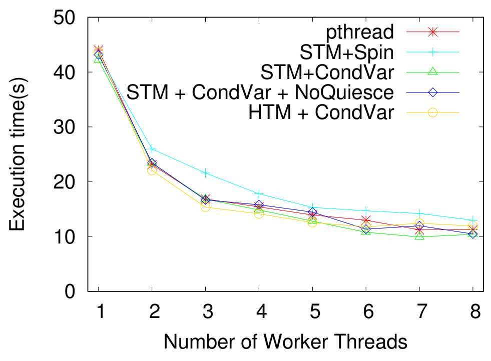
3.5 Additional Considerations

Library and compiler support for the TMTS remains inconsistent, and we encountered three categories of code that caused transactions to serialize unnecessarily or behave incorrectly. For completeness, we discuss each problem and its resolution below.

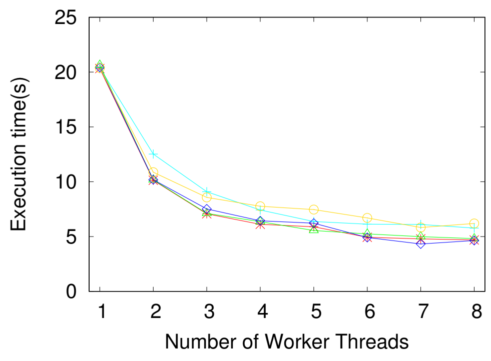
Logging Overheads Both applications can be configured to produce diagnostic output to logs while locks are held. Such output cannot be rolled back, and hence ought to serialize transactions. These outputs are similar to log output in the transactional versions of memcached [102] and Atomic Quake [132]. In those applications, the programs did not require any ordering among logging operations: log messages are timestamped, the order can be determined post-mortem, and the return values of any syscalls during logging are ignored. Consequently, those log operations could either be executed unsafely (and possibly more than once) by STM, or deferred until the end of the transaction. In our applications, we chose to defer output [127]: if we marked the output as unsafe, it would still lead HTM to serialize.

Conditional Synchronization In order to support its soft real-time guarantees, x265 uses timeouts whenever a thread waits on a condition variable. To support this behavior, we first refactored the relevant critical sections to be compatible with Wang's transaction-safe condition variable library [117]. However, the library did not support timeouts. We extended the condition variable library to allow timed wait operations via POSIX semaphores. We verified that this change had no impact on the behavior of the original lock-based program. However, in our experiments, condition variables did present a common source of serialization, especially for HTM. We leave exploration of this problem as future work.

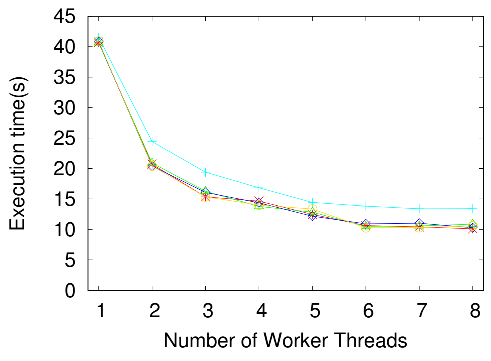
Vector Instructions Lastly, x265 can be configured to make use of vector instructions (e.g., Intel SSE) during rendering. In all, there are over 50 distinct SSE instruction types used by the program, all of which cause STM implementations to serialize. By analyzing each SSE call, we were able to determine that the compiler correctly instrumented SSE memory accesses, at which point the remaining SSE arithmetic operations did not require



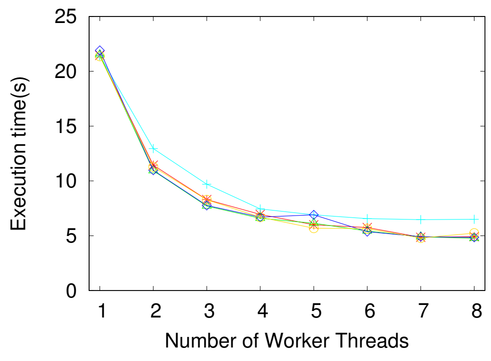
(a) Compress (100K)



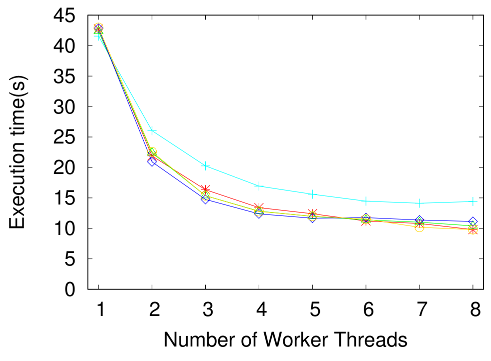
(b) Decompress (100K)



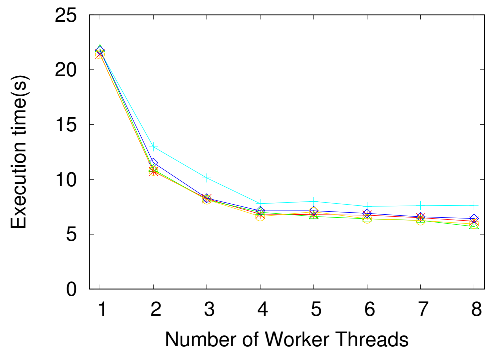
(c) Compress (300K)



(d) Decompress (300K)



(e) Compress (900K)



(f) Decompress (900K)

Figure 3.3: Performance of Transactionalized PBZip2

instrumentation. Our solution was to use the (deprecated) `transaction_pure` annotation to prevent these operations from causing the compiler to insert serializing instructions. However, this is not a satisfactory long-term approach.

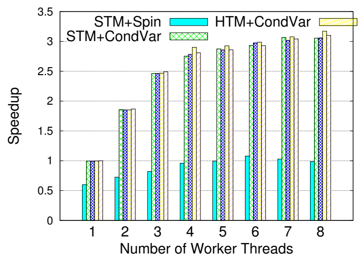
3.6 Evaluation

In this section, we evaluate the use of TMTS-based transactional lock elision in PBZip2 and x265. As much as possible, we preserved the original structure of the source code. The only exceptions are (1) our "ready" flag in x265, which allowed us to transform the code to adhere to two-phase locking, (2) the addition of `TM.NoQuiesce` calls, and (3) minor refactorings of transactions that wait as part of condition synchronization. We use Wang's transaction-friendly condition variables [117], but these require waiting transactions to be enclosed in a loop, and rewritten so that a waiting transaction always performs its wait as its last instruction. Since the TMTS does not officially support these condition variables, we also considered the use of this refactored code *without* conditional waiting, in which case threads repeatedly poll their wait condition within a small transaction.

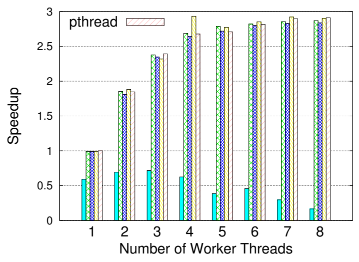
All experiments were conducted on a 4-core/8-thread Intel Core i7-4770 CPU running at 3.40GHz. This CPU supports Intel's TSX extensions for HTM, includes 8 GB of RAM, and runs a Linux 4.3 kernel. We used the GCC 5.3.1 compiler, and only modified its TM implementations enough to support transaction-friendly condition variables (the default HTM implementation does not, due to a lack of support for deferred actions). Results are the average of 5 trials. The STM results use ml_wt algorithm (a privatization-safe version of TinySTM [39]. The HTM results fall back to a serial mode after hardware transactions fail twice. We did not pin threads to specific cores: since our tests are on a single-chip machine, pinning did not offer any significant benefit.

3.6.1 PBZip2

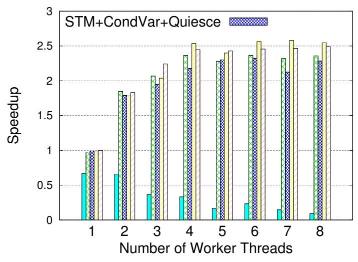
PBZip2 offers two independent operations, Compress and Decompress. We tested each, using a 650MB test file. Within PBZip2, we varied the number of worker threads, as well as the size of the blocks that were processed in parallel; all other configuration parameters were left at their defaults. In our experiments, we varied the number of worker threads from 1 to 8, and considered block sizes of 100K, 300K, and 900K (the range is 100K to 900K, with 900K being the default). Apart from the worker threads, there is a main thread, which runs the benchmark harness but does not participate in the computation.



(a) Netflix film (3810M)



(b) Medium test file (735M)



(c) Small test file (38M)

Figure 3.4: Performance of Transactionalized x265

We compare five algorithms. The baseline is the original code, which uses pthread mutex locks. We then consider three STM algorithms: STM + Spin uses GCC's ml_wt algorithm, with spin waiting when the baseline would wait on a condition variable. STM + CondVar uses transaction-friendly conditional variables. STM + CondVar + NoQuiesce adds dynamic disabling of quiescence for selected transactions. Lastly, the HTM + CondVar executes the transaction using GCC's HTM support.

The main use of critical sections in PBZip2 is to protect queue metadata. Therefore, the average size of critical sections is small. Each thread can access the queue metadata after it finishes compressing or decompressing its block. Conflicts among critical sections are rare: for a 650MB test file, we observed between 950 and 1100 transactions, of which 0.1% aborted at least once in STM. In the HTM experiments, 13% to 18% of transactions aborted twice and fell back to serial mode. Since current HTM support does not report the size of the working set on transaction abort, it would be beneficial for programmers to be able to suggest retry policies on a transaction-by-transaction basis: for queues that are expected to be un-contended, more retries before serialization might be appropriate.

Figure 3.3 shows the performance of the TM algorithms on PBZip2. STM + Spin performs the worst in all conditions except for Figure 3.3(b). This is because spinning not

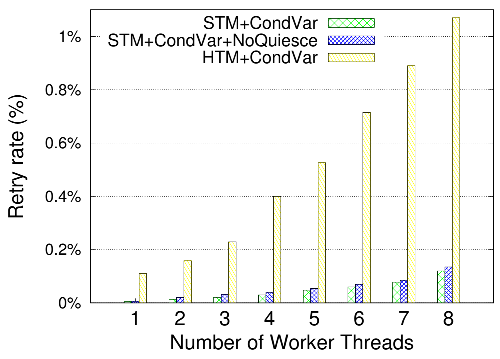


Figure 3.5: Retry rate for Netflix test (10M Transaction Commits)

only wastes compute resources, but also increases contention (between caches and between transactions). In Figure 3.3(b) HTM + CondVar performs worse than STM + Spin in some cases because nearly 20% of HTM transactions fall back to the serial path. Note, however, that at higher thread counts STM + CondVar and STM + CondVar + NoQuiesce both outperform the baseline in Figures 3.3(a) and 3.3(f). At low threads, conflicts are rare, and STM instrumentation overheads dominate.

Disabling quiescence offers mixed results. There is, necessarily, extra tracking and instrumentation overhead, which much be offset. In Figures 3.3(b) and 3.3(d), disabling quiescence offers the best performance at high concurrency levels, which correspond to the scenarios in which the most gain is expected. Note, too, that HTM + CondVar often outperforms the baseline, achieving a peak speedup of 8.5% in Figure 3.3(a). In this case, the fallback rate remains high (15%-18%), suggesting that finely tuning fallback strategies would offer even better performance.

3.6.2 x265

To evaluate x265, we considered three file sizes: small (38MB), medium (735MB), and a real movie downloaded from Netflix (3810M). The application defaults to a pool of 8 worker threads, 3 frame threads, and a main thread. In our experiments, we varied the number of worker threads, and again considered the five algorithms from above. The impact of spinning was disastrous in this workload, even at low thread counts. To maintain readability in Figure 3.4, we plot speedup relative to the single-thread pthread execution, instead of execution time.

The peak performance of HTM was 9.5% better than pthreads at 4 threads (Figure 3.4(b)). Moreover, HTM outperformed pthreads in almost every case. Again, this was with the untuned GCC HTM support: the abort rates in Figure 3.5 suggest that better performance is possible by tuning the fallback policy. We did not observe a significant or consistent impact on performance when we disabled quiescence. In the worst cases, disabling quiescence even decreased performance relative to STM + CondVar. Surprisingly, Figure 3.5 shows that disabling quiescence resulted in slightly higher abort rates for the STM execution. Whereas we expected quiescence to cause bursty transaction start times, the fact that transactions were usually small meant that quiescence delays were not a significant contributor to latency.

Across all experiments, we observed many situations in which STM + CondVar and HTM outperformed the pthread baseline. This result is in spite of the lock erasure effects of TMTS-based transactional lock elision. However, we required support for conditional synchronization, which is currently lacking in the TMTS. We were particularly surprised by the performance of STM; its overheads at transaction boundaries, and on every access of shared memory, were still less than the gain in performance. A variety of optimizations could take this result even further, such as reducing latency for small transactions [33] or making quiescence avoidance conditional on the number of threads (to reduce bookkeeping costs at low thread counts).

3.6.3 Is Quiescence Overhead Important?

Given our past experience with transactional workloads, we expected eliminating quiescence to have a more significant impact on performance. To gain a deeper understanding of its potential benefits, we conducted targeted microbenchmark experiments on a larger machine, with two Xeon X5650 CPUs, each of which has 6 cores/12 threads and runs at 2.67GHz. This test machine includes 12GB of RAM, and runs a Linux 4.4.0 kernel. We used the GCC 5.3.1 compiler. Results are the average of three 10-second trials. This CPU does not support hardware TM. The STM results use the same GCC STM algorithms and configurations as prior experiments.

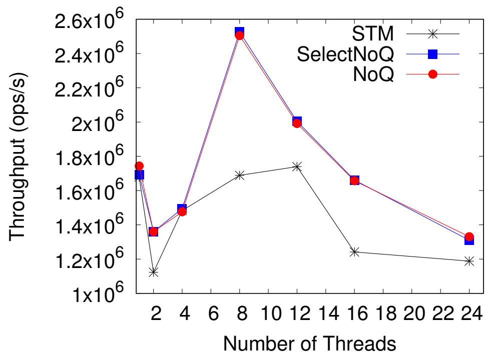
Figure 3.6 presents experiments for three data structure microbenchmarks and two configurations. We limit our focus to high-contention scenarios with small transactions: a list-based set storing 6-bit keys, a hash-based set storing 8-bit keys, and a tree-based set storing 8-bit keys. Transactions execute randomly-selected operations using randomly-selected keys. On the left side of the figure, operations are split evenly between inserts and removes. On the right side, half of the operations are lookups, with the remainder split evenly between inserts and removes.

We consider three STM implementations: the baseline GCC implementation (STM), with quiescence after every transaction; an implementation in which no transactions quiesce (NoQ); and our implementation that uses TM.NoQuiesce to selectively disable quiescence (SelectNoQ). Note that eliminating all quiescence is not correct: whenever a removing transaction frees memory, the GCC TM requires it to quiesce before returning that memory to the system allocator.

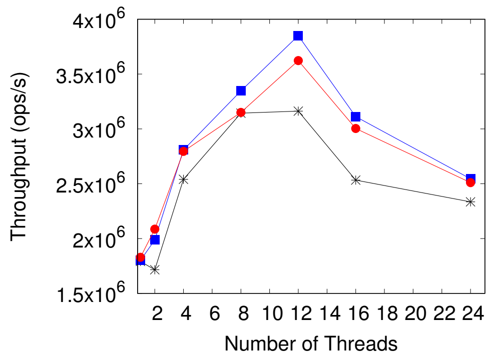
In the list experiments (Figure 3.6a and 3.6b), selectively disabling quiescence offers the same benefit as the unsafe NoQ option. Note that in this workload, little scaling is expected. Note, too, that the dip in performance at two threads is expected; it results from inter-chip communication on a global counter within the GCC STM implementation. Surprisingly, with 50% lookups, selectively disabling quiescence outperforms globally disabling quiescence. In the experiment, the list is initially 50% full, and hence at any time a transaction has a 12.5% chance of performing a successful remove operation and quiescing. Since the benchmark executes transactions in a tight loop, any quiescence represents a period with less contention. In particular, for transactions that must traverse to the end of the list, quiescence by concurrent threads provides a chance to make forward progress. In essence, a small amount of quiescence provides congestion control.

In the hash and tree workloads, conflicts are much less likely, since transactions access disjoint regions of the data structure. In these cases, SelectNoQ performs on par, though slightly below NoQ, and both outperform the baseline STM. Moving from hash to tree, conflicts become more likely. As they do, we see the same trend as with the list: at high contention, occasional quiescence gives expensive transactions a chance to complete.

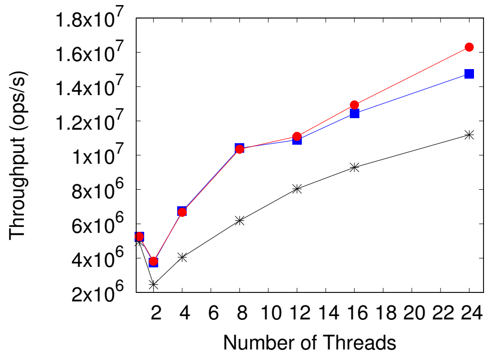
In summary, we observe that removing quiescence has a benefit that is proportional



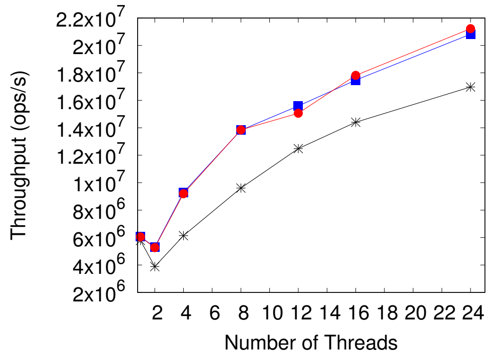
(a) List(Lookup=0%)



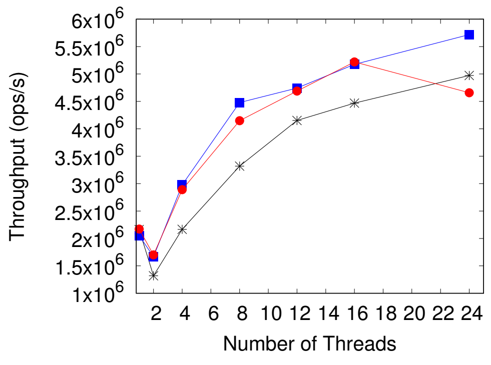
(b) List(Lookup=50%)



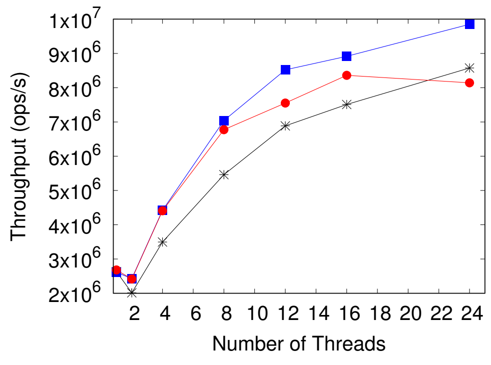
(c) Hash(Lookup=0%)



(d) Hash(Lookup=50%)



(e) Tree(Lookup=0%)



(f) Tree(Lookup=50%)

Figure 3.6: Performance of TM.NoQuiesce on microbenchmarks

to the frequency of transactions and their length. Even when transactions are infrequent or tiny, quiescence is worth eliding. When transactions become more common and larger, quiescence becomes a dominant overhead, and removing it correctly and safely is a valuable optimization. These experiments also explain the variability in STM performance in PBZip2 and x265: when an STM implementation does not provide any other form of contention management, quiescence becomes a congestion control tool. As the TMTS expands to include programmer-specified policies for how to handle conflicts, this behavior should become less likely.

3.7 Conclusions

In this chapter, we applied the C++ TMTS to elide locks in two real-world programs, PBZip2 and x265. In both cases, the programs were already carefully crafted to avoid lock contention and to scale. Nonetheless, transactional lock elision improved performance by up to 9%. To the best of our knowledge, this is the first example of the TMTS, as implemented in the GCC compiler, improving the performance of real-world code. Moreover, the improvement spanned both hardware and software implementations of TM.

Unfortunately, our experience does not validate the expectation that TLE will be easy. In x265, the most important critical section was not serializable, and we could not transactionalize it without understanding several thousand lines of code, and changing the way in which threads interacted with one of the central queues in the program. There is exciting future work in this area, exploring the conditions under which an unmodified critical section can and cannot be transactionalized. Our intuition is that two-phase locking is a sufficient condition, but a more formal study is needed.

We also showed that quiescence avoidance need not be thought of as an all-or-nothing proposition. Specifically, TMTS-based lock elision introduces orderings that a fine-grained locking program would not display. Allowing programmers to avoid these overheads, without sacrificing composability, will speed up STM executions of a program. However, our experiments show that quiescence currently serves as a form of implicit congestion control, and eliminating it can lead to increased abort rates and decreased performance. We believe that the TMTS should allow programmers to specify contention management policies, so that the effect of quiescence can be more predictable.

Chapter 4

Supporting Irrevocable and Long-Running Operations

This chapter introduces *atomic deferral*, an extension to TM that allows programmers to move long-running or irrevocable operations out of a transaction while maintaining serializability. The original work was published in "Extending Transactional Memory with Atomic Deferral" at the 21st International Conference on Principles of Distributed Systems [125], and in "Brief Announcement: Extending Transactional Memory with Atomic Deferral" at the 29th ACM Symposium on Parallelism in Algorithms and Architectures [124].

4.1 Introduction

There are only a few examples of TM being used in "real" software [63, 102]. Why is TM not more widely adopted? One reason is that TM implementations often do introduce significant overhead, especially when transactions are large and there is no hardware support. Another is that adapting a program to use TM is not always a simple matter of replacing lock-based critical sections with transactions, because transactions cannot execute some kinds of operations (e.g., I/O and certain system calls), and most TMs do not support condition synchronization and other important synchronization patterns. Achieving good performance with TM often requires significant changes to the code, both to reduce the size and number of large transactions and to move "unsafe" operations within critical sections

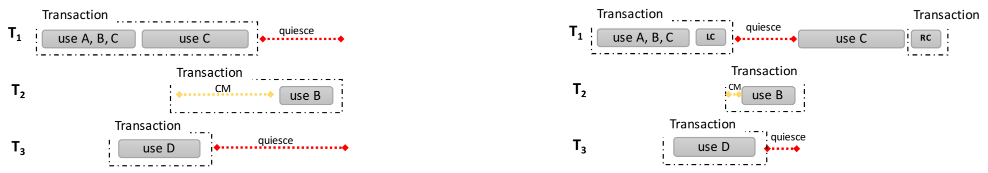


Figure 4.1: Motivation for atomic defer. On the left, T_1's transaction includes a long running operation using C. On the right, C is locked, and then the operation on C is deferred until after the transaction commits. The use of locking and deferral of the operation on C enables the operations by threads T_2 and T_3 to progress more quickly, without violating serializability.

out of transactions while still ensuring correct synchronization.

In this Chapter, we introduce language and run-time support for the *atomic deferral* of operations in transactions: deferred operations do not execute until after the transaction commits. Unlike prior work on deferred operations, atomic deferral does not violate serializability: concurrent transactions cannot observe an intermediate state in which the transaction's updates are complete but its deferred operation's updates are not. This property is particularly important for operations that perform output: if the output fails, compensating or retrying operations can be performed as part of the deferred operation so that it appears to be atomic with the deferring transaction.

We implement atomic deferral by introducing *transaction-friendly locks*, that is, locks that can be acquired and released within transactions, and to which transactions can "subscribe". With these locks, programmers can "mix and match" lock-based and transaction-based synchronization, using whichever is appropriate to the need. We use these locks to protect shared data accessed by deferred operations. The atomicity of the transaction and its deferred operation is preserved by acquiring the appropriate locks before committing the transaction.

4.2 A Motivating Example

To motivate atomic deferral, consider the execution depicted on the left side of Figure 4.1, which captures behavior we observed when transactionalizing the PARSEC dedup kernel [8]. T_1 executes a transaction that first accesses locations A, B and C, and then does a lengthy operation that accesses only C. Concurrently, T_2 executes a transaction that accesses B. Because these transactions conflict, either one of them must abort, or T_2 must wait until T_1 commits before it can proceed with the part of its transaction that accesses B, which is what happens in this case. Because T_1 may privatize some memory, after it commits, it must quiesce, waiting for T_2 to finish (either commit or abort).

The situation is even worse for T_3, which accesses a completely different location, and so does not conflict with either T_1's or T_2's transaction. Nonetheless, T_3 might privatize some memory, and thus it must quiesce until all concurrent transactions complete, so it must wait for T_1's lengthy operation to complete, and then for T_2's transaction to complete afterwards , before it can proceed.

Note, however, that T_1 is only accessing C in the lengthy operation at the end of its transaction. If it could defer that operation until after it commits, then T_2 could start the section of its code that accesses B earlier, and likely commit before T_1 completes its lengthy operation on C. T_3 can also stop quiescing earlier (i.e., when T_2 commits). This case is depicted one the right side of Figure 4.1.

One problem with doing this, however, is that a thread accessing C after T_1 commits the initial part of its transaction but before T_1 finishes its final lengthy operation on C will see an intermediate state of T_1's transaction, violating atomicity. To avoid that, we should prevent other threads from accessing C in that interval. This is represented by the small "LC" and "RC" operations (for "lock C" and "release C" respectively). Achieving this is the core conceptual contribution of this paper, and we show how to do it in the next section.

Prior studies of concurrent applications [102, 117, 127] found that output operations and long-running operations occur often while locks are held. The consequences of such operations are less severe in lock-based code than in programs with TM, primarily because the lock-based programs use many locks: a long-running operation protected by lock L_1 does not impede a thread executing a critical section protected by L_2. However, long-running operations tend to hold as few locks as necessary.

4.3 Extending TM with Atomic Deferral

We support atomic deferral using two new keywords: the `deferrable` annotation on classes, and the `atomic_defer` function, which takes as arguments a function and a list of objects, each of which must be an instance of a `deferrable` class. To defer an operation, a programmer calls `atomic_defer` with a function implementing the deferred operation and a list of all the shared objects that this function may access. Fields of `deferrable` objects must not be accessed directly, but only through getters and setters (a recommended software engineering practice in any case). Thus, if o is an object with a `deferrable` class type and an *expensive* method, then we can defer the execution of that method within a transaction by writing:

$\lambda \leftarrow ()\ \{\ o.\mathit{expensive}()\ \}$

$\texttt{atomic\_defer}(\lambda, o)$

The deferred operation will be executed immediately after the enclosing transaction commits, and in such a way that no other transactions can see a state that reflects the effects of the transaction but not those of its deferred operation. A deferred operation will see any effects of the transaction that occur after the call to `atomic_defer`. If `atomic_defer` is called multiple times within a single transaction, the deferred operations will be executed in the order of their respective calls to `atomic_defer`, and the effects of earlier deferred operations will be visible to later ones.

4.3.1 Implementing Atomic Deferral

We implement atomic deferral by using locks to protect accesses to `deferrable` objects. To provide atomicity, we acquire the locks required by the deferred operation before the transaction commits. We also need a way to notify transactions that access a `deferrable` object (directly as part of the transaction, not deferred) when the lock protecting it has been acquired (by a transaction that calls `atomic_defer` with the object): such transactions must abort and retry after the deferred operation has completed (and the corresponding locks released).

Algorithm 14: Implementation of atomic deferral

```
  // Extensions to classes annotated as deferrable
  deferrable class T
    lock : TxLock // implicit per-instance lock
    ...   // programmer-defined fields
  function    transaction_safe Method(...)
      // subscribe to the implicit lock
      TxLock.Subscribe(lock)
      // programmer-defined logic
      ...

  function atomic_defer(l : λ, objs: Deferrable ...)
      // Use transaction to acquire locks without deadlock
1     transaction
2         for o : objs do
3             TxLock.Acquire(o)
4     deferred_ops.append(⟨l, objs⟩)
```

```
  Additional Per-Thread TM Metadata:
    // all deferred operations for current transaction
    deferred_ops    : list⟨λ, list⟨Deferrable⟩⟩

  function TxEnd()
      // Standard STM Commit; HTM uses a special
        instruction
1     ValidateReadsFinalizeWrites()
      // STM-only: ensure transaction finishes before λs
        run
2     Quiesce()
      // Reset thread's TM metadata
3     move(tm_free_list, local_frees)
4     move(deferred_ops, local_defers)
5     ResetLists()
      // Execute deferred operations
6     for ⟨l, objs⟩ ∈ local_defers do
7         l.execute()
8         for o ∈ objs do TxLock.Release(o)
      // Reclaim memory, reset lists
9     for ptr ∈ local_frees do free(ptr)
```

To this end, we designed *transaction-friendly locks*, which can be acquired and released within a transaction, and which provide a *subscribe* method. The subscribe method must be called from within a transaction, which blocks (or aborts) until the lock is either free or held by the subscribing thread. Multiple threads can subscribe to a lock if it is free. We describe how we implement transaction-friendly locks in Section 4.3.2.

Pseudocode for implementing atomic deferral appears in Algorithm 14. In addition to providing implementations for `atomic_defer` and `deferrable`, we modify the commit operation `TxEnd`. This code assumes that `TxLock` is a class of transaction-friendly locks, and that `Deferrable` is a base class for all `deferrable` classes.

For `deferrable` classes, we add a field that maintains a transaction-friendly lock that protects the class, and we inject a call to `TxLock.Subscribe` for this lock as the first instruction of the transaction-safe version of every member function.[1]

The `atomic_defer` function first acquires the locks of all the `deferrable` objects passed to it, and then appends all of its arguments (the function representing the deferred operation and the list of `deferrable` objects it may access) to *deferred_ops*, a thread-local list of deferred operations. This list will be used when the transaction commits, as described below.

[1]STM implementations typically maintain two versions of each transaction-safe function, one that is called within transactions and one that is called when not in a transaction.

When committing the transaction, we first proceed as usual, validating the read set, finalizing the writes, and then quiescing to avoid privatization problems. Remember that any object that might be accessed by deferred operations has already been locked (i.e., when `atomic_defer` was called with the object), so no other transaction can see any writes to it. We then execute the deferred operations in order, releasing the locks on the `deferrable` objects associated with each deferred operation after that operation is complete. (If an object is accessed by multiple deferred operations, each of them would have acquired the corresponding reentrant lock, and so it is not actually released until the last such operation completes.)

The enclosing transaction may have freed memory, which is normally deferred by the TM after the transaction has quiesced. Because deferred operations may refer to memory that was subsequently freed by the transaction, we delay the freeing of that memory a bit more, until all the deferred operations have completed.

Because deferred operations may use transactions internally, we need to make *deferred_ops* and *tm_free_list* available for their use. Thus, we copy them into local variables before executing any of the deferred operations.

To argue that this implementation is correct, that is, that a transaction and its deferred operations appear atomic, we draw an analogy with two-phase locking, a well-understood technique known to guarantee atomicity. Specifically, a transaction can be thought of as acquiring and holding a single global lock until the transaction commits. Because the lock for every object accessed by deferred operations is acquired before the transaction commits, there is an initial phase in which locks are only acquired (i.e., up to the point that the transaction commits), and a concluding phase in which locks are only released (including the implicit global lock released by the transaction on commit). So all locks are held between the time that the commit operation is invoked and the time that the commit actually occurs. We must also ensure that every access is protected by the appropriate lock, which is why the programmer must provide, when calling `atomic_defer`, all the objects that the deferred operation may access. If a deferred operation accesses some object not passed to `atomic_defer`, then a data race may occur.

Algorithm 15: A transaction-friendly, reentrant mutex lock

```
Fields of TxLock Object:
  owner   : transaction_id    // Lock holder ID
  depth   : Integer           // For reentrancy

function TxLock.Acquire(l)
1   atomic
        // Common case: lock is unheld
2       if $l.owner = nil$ then
3           $l.owner \leftarrow me$
4           $l.depth \leftarrow 1$
5           return
        // Handle reentrancy/nesting
6       else if $l.owner = me$ then
7           $l.depth \leftarrow l.depth + 1$
8           return
        // Wait (spin or yield CPU) until lock is
          released
9       spin()
10      retry

function TxLock.Subscribe()
    // Must be in transaction to call
1   if $owner \neq nil \land owner \neq me$ then
2       retry

function TxLock.Release(l)
1   atomic
        // [Optional] Forbid handoff of held lock
2       if $l.owner \neq me$ then
3           fatal error
        // Handle reentrancy/nesting
4       else if $l.depth > 1$ then
5           $l.depth \leftarrow l.depth - 1$
6           return
        // Else no reentrancy/nesting
7       $l.depth \leftarrow 0$
8       $l.owner \leftarrow$ nil
```

4.3.2 Transaction-Friendly Mutex Locks

The heart of our atomic deferral mechanism is a transaction-friendly mutual exclusion lock, whose pseudocode appears in Algorithm 15. The `TxLock` is reentrant, storing an owner and a count of the locking depth. In this manner, a thread that holds the lock may re-acquire it by incrementing the count. Before any other thread can acquire the lock, the current owner must release the lock as often as needed to ensure $depth = 0$. Since the implementation uses transactions, the *owner* and *depth* fields need not be packed into a single machine word: they are only accessed within transactions. A thread that is currently in a transaction may acquire and/or release `TxLock`s, because it is correct in C++ to nest transactions. Among other things, this means that a thread can acquire multiple locks in a deadlock-free fashion, even without a global locking order: it need only issue all acquisitions inside of a transaction.

`TxLock`s are elidable within transactions, via the `Subscribe` method: a transaction that subscribes to a `TxLock` blocks until the lock is either unheld, or held by the calling thread. Subscription only reads the *owner* field, which allows concurrent subscription by multiple threads. When any thread acquires the `TxLock`, all subscribing transactions will conflict with the new lock owner, and will abort. When the `TxLock` is acquired, the C++ TMTS ensures correct fence semantics: since the transaction accesses shared memory, the TM implementation is required to guarantee that memory accesses preceding the transaction order before it, and memory accesses following the transaction order after it.

If the C++ TMTS adds efficient support for `retry`, transactions could yield the CPU if they attempt to acquire or subscribe to a lock that is held by another thread, and would be woken automatically when the lock is released. In the meantime, we implement `retry` by placing an `atomic` transaction inside a while loop, replacing the `retry` instruction with an exception throw, and adding a `break` as the last statement in the transaction.

4.3.3 Practical Concerns

Deferring operations creates a nonlinear control flow within a program. This nonlinearity is not observable to concurrent threads: the transaction and its deferred operations appear to be a single, serializable operation. However, within the transaction, the programmer must be mindful of a few challenges.

First, the state of the object and thread-private data at the time when the `atomic_defer` keyword appears is not immutable, and may change in the suffix of the transaction that executes before the deferred operation. In addition, the deferred operation does not execute transactionally, and thus races can occur if the deferred operation accesses shared data not protected by the associated `TxLocks`.

Second, the programmer must encapsulate shared objects carefully. Consider a deferred operation that performs a `write` of byte stream B to file descriptor F. If F is shared, then it should be a field of a `Deferrable` object. If B is shared, then it, too, should be a field of a `Deferrable` object. Programmers must decide if B and F should be fields of the same `deferrable` object, or of multiple objects.

Third, since system calls made within a deferred operation happen immediately, some possibility for performance bottlenecks remains. For example, an `fsync` within a deferred operation is often necessary. With `atomic_defer`, the `fsync` and any associated error recovery can be atomic with the transaction, and will not block *all* transactions. However, lengthy deferred operations will still block concurrent transactions that call a method of the associated `deferrable` objects.

Algorithm 16: Diagnostic logging from a critical section

```
  // Version with irrevocable transactions
1 synchronized
      // x is a mutable string
      // i is a mutable integer
2     ...
3     fprintf(stderr, str, x, i)
4     free(x) // Optional
5     ...

  // Deferrable for the log file's descriptor
  class defer_fprintf: public Deferrable
      fd : file     // output file descriptor
      ...
```

```
  // global instance of log file descriptor
  df : defer_fprintf(fd ← stderr)

  // Version with atomic_defer
1 atomic
2     ...
3     tmp ← sprintf(str, x, i)
4     λ ← ()
5         fprintf(df.fd, tmp)
6         free(tmp)
7         free(x) // Optional
8     atomic_defer(λ, df)
9     ...
```

4.4 Programming With Atomic Defer

We now present examples of `atomic_defer` in real applications. The examples depict common use cases, and show the deferral of increasingly complex operations without sacrificing atomicity or resorting to serialization.

4.4.1 Basic Logging

In programs such as memcached [102] and Atomic Quake [132], critical sections occasionally perform logging operations, such as error messages and diagnostic writes to per-thread logs. The program does not require any ordering among logging operations: they are timestamped, and the order can be determined post-mortem. The return values of the output operations are typically ignored. An example appears in Algorithm 16.

When the values to be logged (`x` and `i`) are mutable shared data, existing programs resort to irrevocability or they skip the logging operation. When the values can be encapsulated in a `Deferrable` object, `atomic_defer` is a straightforward transformation: the output string is prepared within the transaction, and the output is deferred until the end of the transaction. Note that this approach ensures ordering of all logging operations on the encapsulated file descriptor. A simpler approach, when ordering is not needed, is to pass **nil** as the second argument on line 8. This approach causes serialization only among transactions that use *df*.

Because transactional versions of existing programs tend to omit this instrumentation

Algorithm 17: Durable output with guaranteed order

```
   // Deferrable wrapper for file descriptors
   class defer_fd: public Deferrable
       fd : file    // output file descriptor
       ...

   // Deferrable objects
   $fd_{D1}$ : defer_fd(fd ← ...)
   $buff_{D1}$ : defer_buffer(buf ← ..., flag ← false)

   // Durable output to $fd_{D1}$
   atomic
1      ...
2      λ ← ()
3          write($fd_{D1}$.fd, $buff_{D1}$.buffer)
4          fsync($fd_{D1}$.fd)
5          $buff_{D1}$.flag ← true
6      atomic_defer(λ, $fd_{D1}$, $buf_{D1}$)
```

```
   // Deferrable wrapper for output buffer
   class defer_buffer: public Deferrable
       buf : buffer      // buffer data
       flag: boolean     // is buffer written?

   // Deferrable objects
   $fd_{D2}$ : defer_fd(fd ← ...)
   $buff_{D2}$ : defer_buffer(buf ← ..., flag ← false)

   // Conditional durable output to $fd_{D2}$
   atomic
7      Subscribe($buff_{D1}$)
8      if $buff_{D1}$.flag then
9          λ ← ()
10             write($fd_{D2}$.fd, $buff_{D2}$.buffer)
11             fsync($fd_{D2}$.fd)
12             $buff_{D2}$.flag ← true
13         atomic_defer(λ, $fd_{D2}$, $buf_{D2}$)
```

in order to avoid serialization, we did not observe a performance impact when applying `atomic_defer` to memcached. However, atomic deferral keeps the code robust and complete without adding too much burden on programmer, and it makes it easier to debug programs during development, by enabling non-serializing `printf` debugging.

4.4.2 Durable Output

Programs often rely on the `fsync` system call to persistent output. In some cases (e.g., durable database operations), it is necessary to order outputs based on the timing of `fsync` calls, such that file F_2 is not updated until after F_1's updates have reached the disk. Simply deferring an `fsync` operation in this case is insufficient. With `atomic_defer`, we can encapsulate the completion status of the `fsync` in a `Deferred` object that is associated with the deferred `fsync` operation.

In Algorithm 17, one thread executes the transaction (T_1) on lines 1 to 6, and another executes the transaction (T_2) on lines 7 to 13. We wish to ensure that T_2 does not write $buff_{D2}$ to file fd_{D2} unless T_1's write of $buff_{D1}$ to file fd_{D1} has been persisted to disk. Since the flag indicating the completion of T_1's `fsync` is encapsulated in a `Deferrable` object, and T_1 sets that flag in an operation that has been deferred, we know that $buff_{D1}$'s implicit lock will be held during the time that the flag is set, and will not be released until after

Algorithm 18: MySQL critical sections in file pool management that are used in asynchronous I/O

```
    // atomic_defer: types and variables
    class file_system_t: public Deferrable
      ...
      space_list : file_space_t

    // wrap the file system as a deferrable object
    file_system : file_system_t

    mySQL_initialize (...)
        // open tablespace data files
1       atomic
            ...
2           $\lambda \leftarrow ()$
3               for $space \in space\_list$
4                   for $node \in space$
5                       $node \leftarrow open(\ldots)$
6           $atomic\_defer(\lambda, file\_system)$
            ...

    mySQL_destroy (...)
        // close tablespace data files
7       atomic
            ...
8           $\lambda \leftarrow ()$
9               for $space \in space\_list$
10                  for $node \in space$
11                      $close(node)$
12          atomic_defer($\lambda$, $file\_system$)
            ...

    mySQL_io_prepare (...)
13      close_more :
14      atomic
            // check system states and select files
            ...
15          $\lambda \leftarrow ()$
16              if $close(file) = -1$
17                  error
18              $n\_open \leftarrow n\_open - 1$
19              if $(n\_open \geq max\_n\_open)$
20                  $need\_close \leftarrow$ true
21                  goto end
                // check the node to do I/O
22              if $\neg node.open$
                    // get file size, do an open and close
                    // save metadata for future I/O
23                  if $node.size = 0$
24                      $node \leftarrow open()$
25                      $offset \leftarrow lseek(file, 0, SEEK\_END)$
26                      $success \leftarrow pread(two\ pages)$
27                      $close(node)$
28                  $node \leftarrow open()$
29              end :
30          atomic_defer($\lambda$, $file\_system$)
            ...
31      if $(need\_close)$
32          goto close_more
```

the `fsync` returns. Consequently, when T_2 executes line 7, three cases are possible: (1) T_1 has not yet executed line 6, in which case line 7 returns, and then line 8 evaluates to false; (2) T_1 has executed line 6 but has not completed lines 3 to 5, in which case T_2 will call `retry` and ultimately land in the third case; or (3) T_1 has completed its deferred execution of lines 3 to 5, in which case T_2 can subscribe to $buff_{D1}$, and then observe a **true** value on line 8. Note that T_2 can only perform its write in the third case, which orders lines 3–5 before lines 10–12, and thus the deferred outputs occur and reach the disk in the required order.

4.4.3 Opening Files as Output

Our final example comes from the MySQL InnoDB storage engine. InnoDB maintains a pool of file descriptors, which is protected by a lock. Metadata is associated with each file descriptor, and allows updates to files to be performed via asynchronous I/O. For example, to append new records to the end of a file, a thread locks the pool, updates the size of the file, and then unlocks the pool. It then issues an asynchronous write. Subsequent appends will follow the same protocol, and hence will appear at a later point in the file, even if their writes reach the disk earlier.

While reads and writes do not occur in critical sections, and hence would not serialize a transactional version of InnoDB, the management of the pool depends on the ability to open and close files dynamically, in order to stay below a pre-set maximum number of open files. If a file must be opened when the pool is at capacity, then a thread will lock the pool, close some other files that do not have outstanding accesses in-flight, and then open the new file. In transactional InnoDB[2], this operation requires irrevocability, and serializes all memory transactions, to include those performing read-only queries of data within the database.

With `atomic_defer`, the pool becomes a `Deferrable` object. On any modification to file descriptor metadata, a thread uses a transaction that subscribes to the pool. Thus, file operations can proceed fully in parallel, since they use asynchronous I/O to perform their file accesses, and transactions to operate on disjoint file metadata regions. In the uncommon cases where files are opened and closed, the system calls are deferred from a transaction. While the system calls are in-flight, concurrent accesses to the pool stall (via `retry`). Once the pool is returned to a usable state, any suspended threads resume.

4.5 Performance Evaluation

We now present experiments that demonstrate the benefit of `atomic_defer`. We conduct tests on two platforms. In charts depicting scalability up to 8 threads, the platform is a 4-core/8-thread Intel Core i7-4770 CPU running at 3.40GHz. This CPU supports Intel's TSX extensions for HTM, includes 8 GB of RAM, and runs a Linux 4.3 kernel. Experiments with larger thread counts were conducted on a machine with two 18-core/36-thread Intel E5-2699 V3 CPUs running at 2.30GHz. This CPU also supports TSX, includes 128 GB

[2]Unfortunately, adding TM to InnoDB revealed a bug in GCC, which produces an internal compiler error.

of RAM, and runs a Linux 4.8 kernel. Our extensions were implemented in GCC 5.3.1. Results are the average of 5 trials.

4.5.1 Performance of atomic_defer on a Transactional I/O Microbenchmark

One motivation for `atomic_defer` is to avoid the serialization of `synchronized` transactions, while allowing output that is atomic with respect to the transaction. We begin with a microbenchmark study to observe the behavior of transactions that perform irrevocable operations on files. Our microbenchmarks are patterned after work by Demsky and Tehrany [25]. Whereas they required custom instrumentation of system calls in order to make them transaction safe, we run I/O operations without instrumentation, using either irrevocability or `atomic_defer`. Algorithm 19 presents the general behavior of our microbenchmarks: a transaction produces content and identifies a file to update. It then performs I/O, which includes opening a file, reading the file length, and appending data to the file. The I/O can be deferred or executed irrevocably. To use `atomic_defer`, we encapsulate the I/O streams in `deferrable` objects, and then use `atomic_defer` to delay the operation on line 16. Figure 4.2 presents experiments with four configurations of the microbenchmark. In each case, threads cooperate to complete a total of 1M operations. The figure presents results for STM, but trends for HTM are the same.

Figure 4.2 (a) explores the overhead of atomic deferral when there is only one file, and hence no concurrency, by comparing performance when transactions are replaced with a coarse-grained lock (CGL), and when transactions use irrevocability (irrevoc), or atomic deferral (defer). We see that the baseline GCC TM implementation (irrevoc) is well-tuned to handle irrevocability: it serializes transactions early, avoids instrumentation, and achieves performance comparable to CGL. In contrast, `atomic_defer` pays a constant overhead per transaction to support rollback, even though no rollbacks occur. As the thread count increases, overheads due to `retry` cause additional slowdown. This is partly a result of our `retry` implementation aborting and immediately retrying, instead of de-scheduling the transaction until it can make progress. Until the C++ TMTS includes efficient `retry`, this cost is unavoidable.

Algorithm 19: An example of deferring I/O and system calls

```
   // Encapsulate streams in a Deferrable object
   class defer_file: public Deferrable
       input    // input stream
       output   // output stream

   // An array of files
   $dfs$: $defer\_file[]$

   // Operation to be deferred
1  $\lambda \leftarrow$ (id, content)
       // Read File
2      if $\neg dfs[id].input.open()$ then error
       // Get the length of the file
3      $dfs[id].input.seekg(0, end)$
4      $len \leftarrow dfs[id].input.tellg()$
5      $dfs[id].input.close()$
       // Write to the file and close
6      $tmp \leftarrow format(content, len)$
7      $dfs[id].output.write(tmp)$
8      $dfs[id].output.close()$

   // Irrevocable version of benchmark
9  synchronized
10     $content \leftarrow \ldots$
11     $id \leftarrow \ldots$
12     $\lambda(id, content)$

   // atomic_defer version of benchmark
13 atomic
14     $content \leftarrow \ldots$
15     $id \leftarrow \ldots$
16     $atomic\_defer(\lambda(id, content), dfs[id])$
```

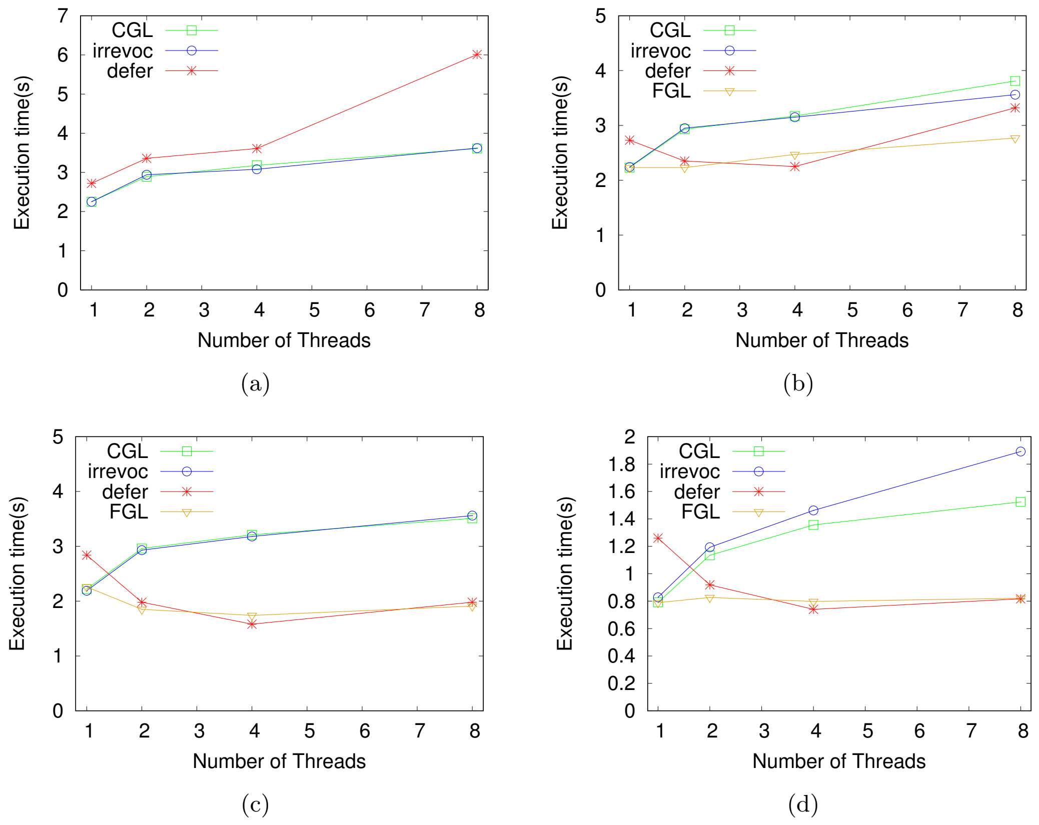
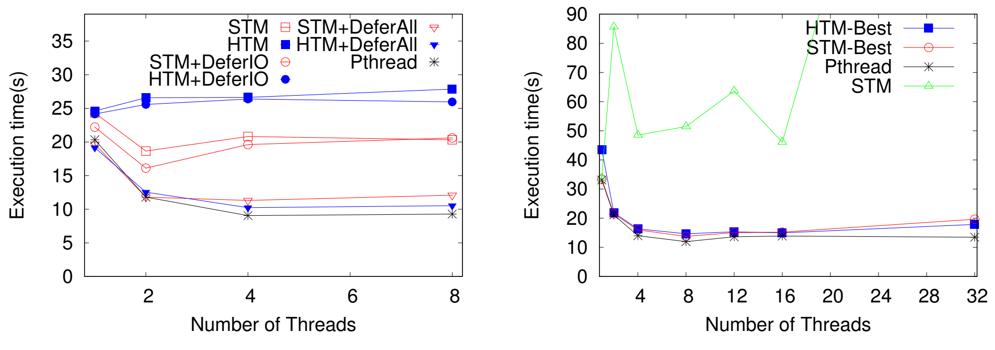


Figure 4.2: Atomic_defer performance on an I/O microbenchmark

Figures 4.2 (b) and 4.2 (c) expand the number of files to 2 and then 4, and threads update files with equal probability. We include another non-transactional baseline, with one fine-



(a) (b)

Figure 4.3: Performance of PARSEC dedup with `atomic_defer`

grained lock (FGL) per file. We again see that single-threaded code has higher overhead when using `atomic_defer`, due to instrumentation and the management of lambdas. While the behavior of CGL and irrevoc is unchanged, deferral now shows scaling on par with fine-grained locks, achieving indistinguishable performance at 2 and 4 threads. When the thread count greatly exceeds the potential concurrency (e.g., 8 threads and 2 files, Figure 4.2 (b)), we still see extra overheads from `retry`. However, when there is enough concurrency in the workload (e.g., 4 files), `atomic_defer` scales well.

Finally, in Figure 4.2 (d) transactions append to files that are kept open throughout the experiment. There are still 4 files, but the smaller critical sections reveal an overhead in the irrevocability mechanism: when one transaction becomes irrevocable, the others block, possibly yielding the CPU. When the irrevocable transaction is brief, the overhead of yielding becomes visible, and irrevoc degrades worse than CGL. Meanwhile, FGL has flat performance, and defer overcomes latency at 1 thread to be competitive with FGL.

4.5.2 Performance of atomic_defer on PARSEC Dedup

Wang et al. reported [117] that PARSEC's dedup kernel [8] ceased to scale when transactions replaced locks. Dedup is a pipeline application, and the original file output stage performs output while holding a lock; Wang's version replaces that lock with an irrevocable transaction. When the irrevocable transaction executes, it must serialize all concurrent transactions.

When we rewrote dedup's output operation to use `atomic_defer`, irrevocability ceased

Algorithm 20: Deferring reliable output in PARSEC dedup

```
function pipeline_out(buf, len, fd)
    // fd may be unreliable, so monitor progress of writes
1   $(p, nsent, rv) \leftarrow (buf, 0, 0)$
2   while $nsent < len$
3       $rv \leftarrow$ write$(fd, p, len - nsent)$
4       if $transient\_error(rv)$ then
5           continue
6       if $fatal\_error(rv)$ then error
7       $nsent \leftarrow nsent + rv$
8       $p \leftarrow p + rv$
9   $fsync(fd)$
10  $free(buf)$
```

```
// Version with irrevocable transactions
1 synchronized
      ...
2     $pipeline\_out(packet.buf, len)$
      ...

// Version with atomic_defer: packet is now deferrable
deferrable class packet
1 atomic
      ...
2     $\lambda \leftarrow ()$
3         $pipeline\_out(packet.buf, len)$
      ...
4     $atomic\_defer(\lambda, packet)$
```

to cause performance degradation, but the benchmark still scaled poorly. A sketch of the code transformation appears in Algorithm 20. Since the buffer to be output was already encapsulated in a struct ("packet"), we made that struct `deferrable` and ensured that its fields were accessed through getters and setters. Deferring the operation was then a one-line change, which preserved the ordering of `fsync` operations and error handling with respect to output and subsequent concurrent accesses. The performance of dedup with this change appears in Figure 4.3 (a), as "+DeferIO".

We discovered that the `Compress` function was marked as `pure`, because it does not access any shared memory. Marking the function `pure` indicates to the compiler that the function can be run without instrumentation, lacks side effects, and can be run from a non-irrevocable context even when the compiler cannot prove that irrevocability is not needed. `Compress` is a long-running function, and in HTM, it accesses more memory than can be tracked by the HTM; the HTM execution serializes whenever a call to `Compress` exceeds the capacity of the hardware. In STM, the call to `Compress` represents a period of time during which other transactions can commit, but will delay in their quiesce operations. While the run-time behaviors are different, the consequence is the same: when one transaction calls `Compress`, other transactions cannot make progress.

Since `Compress` is pure, it can be deferred. We encapsulated the compressed buffer as a `deferrable` object, so that the run-time system can suspend transactions when they attempt to access a buffer that is locked for deferred compression. This has a profound impact on both STM and HTM. In HTM, the transaction ceases to overflow hardware capacity, and serialization is avoided. In STM, compression ceases to impede quiescence, and concurrent threads can make forward progress. In Figure 4.3 (a), we see that the "+DeferAll" curves for both HTM and STM now compete with pthread locks, representing a 1.7x speedup for STM and 2.7x speedup for HTM.

Lastly, we measure the impact of `atomic_defer` on the 36-core system, to see how performance scales across chips and in the face of significantly more hardware parallelism. In Figure 4.3 (b), the performance of the baseline HTM is not shown: the 32-thread STM performance exceeds 270 seconds, and HTM never scales. When we employ `atomic_defer` to move output and pure functions out of transactions, both STM and HTM improve by roughly 10x compared to their respective TM baselines, reaching the same performance as pthread locks. While these optimizations require more careful reasoning about the program, we contend that these optimizations are still easier than reasoning about fine-grained locking.

4.6 Conclusions

In this chapter, we presented a technique for atomically deferring operations in memory transactions. The key feature of our work is that concurrent transactions cannot detect that an operation was deferred: the operation appears atomic with the corresponding transaction, which retains serializability. The fundamental technique to enable atomic deferral is composing transactions with locks and `retry`-based condition synchronization, to facilitate a form of two-phase locking. With the deferred operation, transactions may perform complex operations and access a subset of shared memory. Using atomic deferral allows transactions to perform output without serializing, and was the foundation for a dramatic improvement in the performance of the PARSEC dedup benchmark.

Atomic deferral requires more complex reasoning by programmers than irrevocability, and is less general. However, when applicable, it eliminates serialization overheads, and shortens the time that transactions spend quiescing. In our view, the additional programmer overhead to use atomic deferral is small, and more than justified by the benefits. For example, we presented a scenario where atomic deferral can avoid serialization when managing file descriptor pools in MySQL, and another where files can be updated in order, while obeying strong persistence requirements.

Chapter 5

Supporting Data Persistence

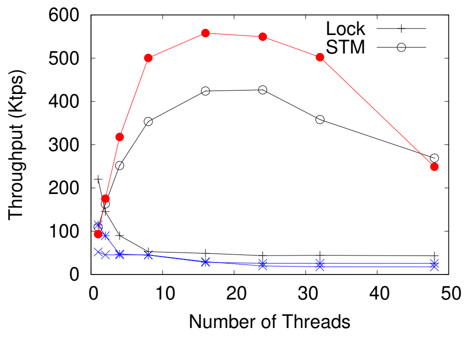
This Chapter studies the performance of persistent transactional memory (PTM) algorithms and introduces runtime optimizations and contention management for PTM under different programming models. The work first was published in "Brief Announcement: Optimizing Persistent Transactions" at the 31th ACM Symposium on Parallelism in Algorithms and Architectures [130].

5.1 Introduction

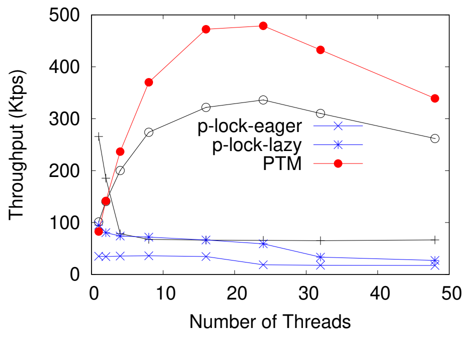
In many ways, PTM resembles software transactional memory (STM), an approach to creating high-performance concurrent programs. STM promised to simplify the creation of scalable programs by raising the level of abstraction for programmers: instead of thinking about explicit fine-grained locks, programmers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

STM promised high scalability with limited run-time latency. By and large, it did not succeed. Single-thread latency was often $3\times$ that of sequential code, and the programming model for STM necessitated overheads within the STM implementation that prevented the levels of scalability needed to justify its cost.

In this chapter, we show that PTM is not destined to face the same fate as STM.



(a) TPCC-HashTable



(b) TPCC-B+Tree

Figure 5.1: Transactional TPCC benchmark performance. When the program data is in DRAM, synchronization is achieved using a single coarse lock or transactions. When the program data is in NVM, synchronization is achieved using a coarse lock + undo (eager), a coarse lock + redo (lazy), or persistent transactional memory.

Consider the experiments in Figure 5.1, which show 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, and all program data is in a contiguous segment of RAM. When we use a global mutex ("lock") to implement transactions, single-thread throughput is 2× to 3× that of STM. STM requires 4–8 threads to match the throughput of the lock-based code, and at its peak, has only 2× the throughput.

We can make the lock-based system persistent by either deferring all updates of memory to the end of each critical section (p-lock-lazy), or by updating an undo log prior to each write to the NVM within the critical section (p-lock-eager). These systems also require explicit flushing of data from the cache to memory, and explicit memory fences. The eager approach is almost 10× slower than the non-persistent case, and the lazy approach is 2× slower. Neither scales. In contrast, by applying the techniques discussed in this chapter, persistent transactional memory (PTM) is able to achieve 90% of the performance of the persistent lock-based code, and scale to 5× its throughput.

The experiment suggests that the promise of PTM is far greater than that of STM. There are two reasons. First, the fundamental overheads of persistence, such as flushing and fencing, mask the instrumentation overheads of a well-designed PTM. Second, the PTM programming model is fundamentally different than the STM programming model, affording new optimizations that can reduce latency and improve scalability.

In this chapter, we study the relationship between PTM and STM. We introduce a universal transformation from STM to PTM, characterize fundamental overheads associated with different programming models for PTM, and present optimizations for PTM within these programming models. In particular:

- We show that PTM can be faster than STM, and more scalable, for realistic workloads.
- We explain why the conventional wisdom about what makes a "good" STM algorithm does not always apply to PTM.
- We show that PTM cannot use STM techniques to ensure progress, and we show how to easily ensure progress for PTM.
- We demonstrate the importance of the persistence model on the performance of PTM algorithms.
- We introduce run-time optimizations specific to PTM, which raise performance by as much as 60%.

The remainder of this chapter is organized as follows. In Section 5.2, 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 5.3, we present a general transformation for turning an arbitrary STM algorithm into a PTM algorithm. We also present baseline performance numbers for PTM versus STM. Section 5.4 presents a set of optimizations, some of which are only applicable to the more restricted model, others of which apply to both models. We also measure the impact of each optimization, in isolation. Then, in Section 5.5, we measure the impact of combining optimizations. Finally, Section 5.6 summarizes our conclusions.

5.2 System Model 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 not be invalid. At a high level, 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 the following factors:

5.2.1 Hardware Support for Persistence

Marathe et al. [77] describe three hardware persistence domains. The simplest (persistence domain 0, or PDOM-0) only contains the NVM RAM 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 for a PDOM-2 system, then when the machine is powered back, it can resume immediately, with no loss of data. 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 a `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 `clwb`s, to move data from the memory controller to the DIMMs.

Current and upcoming Intel systems provide PDOM-1, and instructions related to PDOM-0 have been deprecated [58]. In Intel's PDOM-1 systems, 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. As a result, 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. As a result, any transaction, even one that is not concurrent, requires instrumentation on every load and store of persistent memory.

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

Table 5.1: Overhead of self-referential pointers

5.2.2 Position-Independence

Typically, a persistent region is achieved by mapping a named, contiguous range of physical addresses in NVM into a program's virtual address space via `mmap` [17]. When a program restarts and reloads such a region, its virtual-to-physical mappings may change. The system may require that a data structure stored in NVM use position-independent pointers. These can either consist of two machine words (to represent a file ID and offset within the file) [59] 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`). Position-independence simplifies recovery: when a program re-starts, it can load the persistent region and use it immediately. Without position independence, it is necessary to walk the entire persistent region and re-write pointers. Note that rapid recovery requires position independent pointers and also a persistent allocator.

Table 5.1 depicts the increase in latency that position-independent pointers introduce in a non-persistent program. The experiment was conducted by using our transactional instrumentation (discussed in Section 5.3) to dynamically treat each pointer in the benchmark as a self-referential pointer. As such, this experiment is a low estimate of the true cost of position independence, as it does not consider the additional `clwb` and `sfence` instructions that a persistent allocator would introduce. When we consider that persistent regions are rarely loaded, this cost seems excessive, and thus we focus on non-position-independent pointers.

5.2.3 Hardware Memory Diversity

In systems with NVM RAM, it is also possible to have traditional DRAM. One example is Intel Apache Pass. In these systems, a fundamental question is whether a persistent transaction is allowed to read and write to the DRAM, or only the persistent RAM. It may be difficult to statically enforce a requirement that transactions operate exclusively on one type of memory or the other. The distinction is important, because a persistent memory region is typically allocated with `mmap`, and deallocated with `munmap`. An allocator that runs within the region may create and reclaim ranges of memory within the region, but it cannot return portions of the region to the operating system. In contrast, allocators for DRAM can return individual pages of virtual memory to the operating system when they are no longer in use.

STM literature establishes that when transactions are able to execute speculatively, then a transaction that frees memory cannot simply defer the call to `free` until after the transaction commits: freeing might return a page to the operating system, and a concurrent transaction (which is destined to abort) may be in the process of accessing a location on that page. If the freeing thread does not wait for all concurrent transactions to reach a safe point, or epoch, then those threads may incur a segmentation fault. This behavior is a subset of a larger pattern called *privatization*. A PTM that allows transactions to access DRAM and NVM must incur small privatization overheads at the boundaries of every transaction (and optionally also whenever an in-flight transaction checks the consistency of its read set). It must also incur large overheads when committing a transaction that privatizes memory. Privatization patterns are sufficiently complex that the default is for every transaction to incur this overhead whenever it commits.

5.2.4 Static Separation

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. In legacy systems, this may mean that 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. In contrast, 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. It is natural, then, to require that every store be part of a transaction.

Going a step further, we can require that every load from a persistent region is also part of a transaction (note that micro-transactions make the overhead of such a design minimal [33]). The resulting "static separation" model [1] is able to track memory at a coarser granularity than permitted by transactional programming models for DRAM [122].

5.2.5 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 [17]. For the purposes of this Chapter, 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.

5.2.6 Models Considered in this Chapter

From the above, we focus on two models. 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 and (b) remap pointers within the persistent region. Upon this base, the general persistence model (GP) assumes that main memory consists of both NVM and DRAM, that any single transaction may access both kinds of memory, and that it 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.

5.3 Universal Persistent TM Transformation

STM algorithms typically contain five functions, which interact with program addresses and STM metadata:

- `Begin`: Starts a transaction by snapshotting the thread's architectural state and possibly reading some global metadata.
- `Write(a, v)`: Attempt to write the value `v` to address `a`. `Write` may directly modify the value at address `a`, or might store the `v` in a private buffer, to "redo" at a later 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 and return the value at `a`. Like `Write`, `Read` may cause a validation. To ensure processor consistency, it may need to check if `a` is in the redo log managed by `Write`.
- `Commit`: Finishes a transaction by finalizing the writes only if the reads all remain valid.
- `Abort`: Rolls back any writes, clears all thread-local metadata, and restores the checkpoint from `Begin` to re-try the transaction.

The most general STM algorithms rely on global metadata through which transactions can lock locations before modifying them. These locks may be explicit readers/writer locks [29], or ownership records (orecs) [28] 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 [24, 92] or bit vectors [111] 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 violate processor consistency by failing 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 merely resets its local lists, whereas an eager transaction must use its undo logs to restore the values of locations it wrote, then releases locks.

5.3.1 Ensuring Correctness for Common-Case Transactions

Algorithm 21 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 persistent occurs in the `Commit` function. 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

Algorithm 21: Universal transformation from 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 ()
```

clean up (line 13).

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 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.

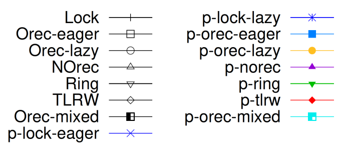
5.3.2 Ensuring Progress and Instrumentation

A simple and easy strategy for ensuring the progress of DRAM transactions is through the use of irrevocability [118]. The most common irrevocability mechanism is to use a form of readers/writer locking, in which transactions acquire read permission before starting/restarting, and release it upon commit/abort. A transaction that acquires the lock in write mode is guaranteed to run without any active concurrent transactions.

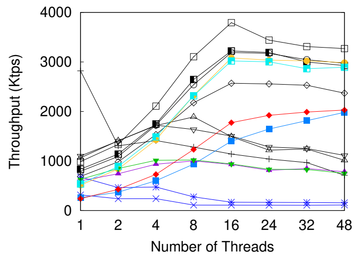
Transactional compilers will make a transaction irrevocable if it attempts to do something that cannot be undone (e.g., an I/O system call). The rationale is that irrevocable transactions cannot abort, and hence the call will thus be safe. These compilers also optimize irrevocable transactions: since they are running without concurrency, they do not need any instrumentation on reads or writes, nor do they need undo and redo logging. Once a transactional compiler and STM library support irrevocability, it also provides a simple tool for ensuring progress: a transaction with more than k consecutive aborts, for some threshold k, can become irrevocable. Without concurrency or instrumentation, the transaction can then finish as quickly as possible.

Irrevocability is unsuitable for PTM, because a fault during an irrevocable transaction cannot be recovered: the transaction's writes are not reflected in an undo log, and any irrevocable operation, like a system call, may not be reversible. To transform an STM into a PTM, the irrevocability code must be removed, and a transaction that requires irrevocability due to system calls must be rewritten.

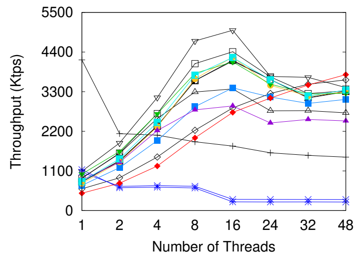
Without irrevocability, we require a new mechanism to guarantee progress. We use the "hourglass" scheduler [72]. The key idea behind the hourglass is to reduce contention, but in a less rigid way than irrevocability. Let hg be an atomic boolean variable. A thread with more than k consecutive aborts can try to transition hg from `false` to `true`. If it succeeds, it is said to own the hourglass. When the thread's next transaction commits, it sets hg back to false.



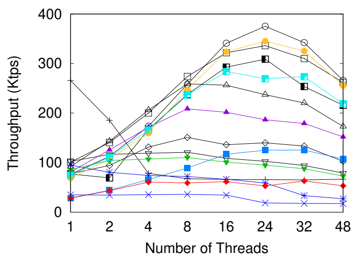
(a) Legends



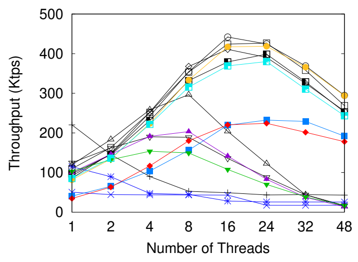
(b) B+Tree Insert



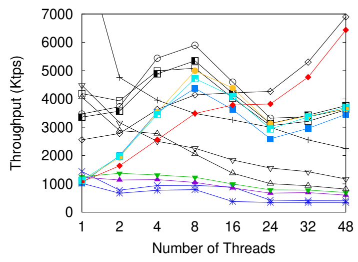
(c) B+Tree Mix



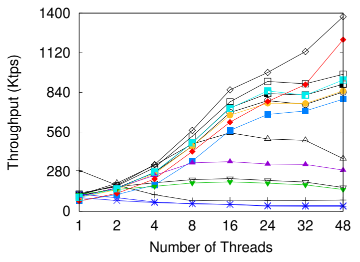
(d) TPCC-B+Tree



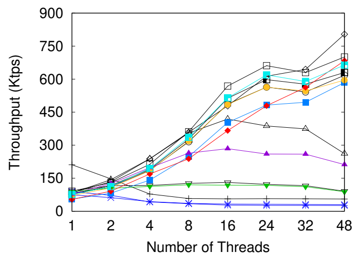
(e) TPCC-HashTable



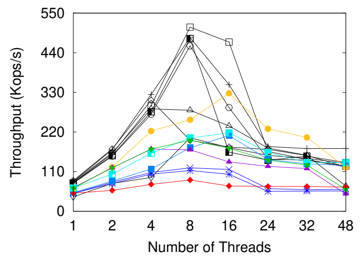
(f) TATP



(g) Vacation (low)



(h) Vacation (high)



(i) Memcached

Figure 5.2: Performance comparison of STM to naive PTM (general model).

The nuance in the hourglass comes from its use in the begin function. After line 1 (or line 22), the thread that owns the hourglass proceeds immediately to the next line. If a thread does not own the hourglass, it spins until the hourglass is false, then proceeds.

The hourglass is simpler and more concurrent than irrevocability. With irrevocability, a distressed transaction waits on concurrent transactions to finish, so that it will never abort again; with the hourglass, a distressed transaction may abort after owning the hourglass. However, as other threads complete their transactions, they will become unable to start new transactions until after the distressed transaction completes.

Since correctness is not compromised if the distressed transaction (which is still instrumented) runs concurrently with other transactions, the check of hg in the begin function does not require memory fences or readers/writer locking. This makes it more scalable than irrevocability in the common case, and incurs less latency.

5.3.3 Performance of Naive PTM and STM

Figure 5.2 presents the performance of a variety of 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 implementation where all transactions are protected by a single global lock. “Orec-eager” refers to a STM that uses ownership records and undo logging, similar to that used by GCC TM [39]. “Orec-lazy” is identical to “orec-eager”, except it acquires locks at commit time and uses redo logging [28,108]. “Orec-mixed” uses redo logging, but still acquires locks early, like orec-eager [116]. 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” [24] 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” [29] is an eager algorithm with carefully-crafted readers/writer locks. “Ring” [111] is a lazy algorithm that uses a log of 1024-entry bit vectors to capture the history of committed writer transactions. Readers can validate by intersecting their for fast but imprecise validation of active transactions against newly committed transactions. The persistent versions of the above algorithms are indicated by the “p” prefix. They were created via transformation in Listing 21. 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 [16]. Excluding “lock” algorithms, all STM and PTM algorithms use our hourglass scheduler to ensure progress.

We instrumented code using an open-source LLVM extension for STM [123], and we integrated the 7 STM and 8 PTM algorithms into it. This 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 extension that did not instrument reads. We also employed Link Time Optimization (LTO) to eliminate 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 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 consider every open-source multi-threaded PTM benchmark we could find, which includes (i) one real world application, Memcached [80]; (ii) write-only benchmarks from DudeTM [69]: the TPCC transaction processing benchmark, TATP telecom application benchmark, and a B+ tree data structure microbenchmark; (iii) the "vacation" travel reservation benchmark [85, 87]. 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 this experiment 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 is consistently among the best in Figure 5.2(c)(f)(g)(h), as is the persistent version. The success of persistent TLRW is due to our optimizations, which carefully order the explicit readers/writer locking in TLRW to avoid additional `sfence`s for persistence. Another reason is TLRW's unique, scalable approach to privatization safety: Other algorithms incur privatization 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 [22, 78]. The `clwb` instruction is incompatible with hardware TM. Without the promise of a hardware-accelerated future, p-norec does not appear viable.

For orec-mixed, which was implemented in Mnemosyne [116], 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.

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.

5.4 PTM Optimizations

5.4.1 Captured Memory

From the earliest days of STM research, systems avoided instrumentation for accesses to memory on stack frames whose lifetime was limited by the scope of the transaction. In addition, Riegel et al. [98] and Dragojevic et al. [35] developed techniques for avoiding instrumentation of "captured memory", that is, 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

	TPCC-HashTable	TPCC-B+Tree	B+Tree (Insert)
p-lock-eager	1.674	1.543	1.235
p-lock-lazy	1.055	1.045	1.041
p-orec-eager	1.674	1.443	1.126
p-orec-lazy	1.115	1.101	1.048
p-norec	1.107	1.081	1.061
p-ring	1.068	1.093	1.011
p-tlrw	1.626	1.358	1.127
p-orec-mixed	1.118	1.125	1.088

	Vacation (low)	Vacation (high)	Memcached
p-lock-eager	1.190	1.139	1.01
p-lock-lazy	1.026	1.021	1.01
p-orec-eager	1.179	1.177	1.272
p-orec-lazy	1.067	1.069	1.12
p-norec	1.052	1.024	0.999
p-ring	1.003	1.092	1.031
p-tlrw	1.173	1.162	1.264
p-orec-mixed	1.099	1.058	1.026

Table 5.2: Speedup from the last allocation tracking optimization (single-thread).

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 begin 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 additional branch before lines 15 and 20 for the lazy algorithm in Listing 21, 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

	TPCC-HashTable	TPCC-B+Tree	TATP	B+Tree (Insert)
p-lock-eager	1.229	1.519	1.049	1.366
p-lock-lazy	1.086	1.227	1.296	1.226
p-orec-eager	1.185	1.464	1.224	1.406
p-orec-lazy	1.041	1.124	1.084	1.113
p-norec	0.423	0.315	0.905	0.750
p-ring	1.022	1.121	1.107	1.158
p-tlrw	1.251	1.347	1.116	1.357
p-orec-mixed	1.088	1.113	1.052	1.063

	Vacation (low)	Vacation (high)	Memcached
p-lock-eager	1.231	1.197	1.11
p-lock-lazy	1.218	1.205	1.03
p-orec-eager	1.185	1.196	1.16
p-orec-lazy	1.104	1.120	1.114
p-norec	0.651	0.567	1.02
p-ring	1.076	1.058	1.022
p-tlrw	1.155	1.177	1.139
p-orec-mixed	1.144	1.097	1.055

Table 5.3: Speedup of aligned memory and coarse-grained logging for single thread execution

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 5.2 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 a startling 13% or higher. 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 (4% 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.

5.4.2 Memory Alignment and Logging Granularity

Both the GP and IP models assume that addresses in the NVM will only be accessed via transactions. Since persistent memory is given to the program at the granularity of pages, the models both permit for a coarser granularity of management than in 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). The problem is that adjacent addresses may be accessed by a concurrent, non-transactional thread. With undo logs, this means that the entries in the log must have variable granularity. 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 to indicate which bytes of the entry should be written back. In addition, when ensuring processor consistency during reads (lines 15–16), the bitmap dictates which bytes of an entry should affect the return value of the read. In a general-purpose STM implementation that supports C++ casting and mixed-granularity access, the lookup in Listing 21 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 recommend developers to think more carefully about granularity, so that it can be a tunable parameter in future systems.

Table 5.3 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

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

Table 5.4: Speedup of fence pipelining for TLRW on single thread execution

the same cache line, such problems do not manifest in our 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 program 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 requires the transaction to maintain two redo logs: one for NVM addresses, with coarse granularity, and one for DRAM addresses, with STM granularity.

5.4.3 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 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

	TPCC-HashTable			TATP			B+Tree (Insert)		
Threads	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
p-orec-lazy	1.026	1.052	1.007	1.018	1.056	1.050	1.010	0.998	1.008
p-norec	1.108	1.167	1.147	1.301	1.245	1.229	1.086	1.141	1.169
p-ring	1.041	1.028	1.130	1.152	1.211	1.457	1.085	1.145	1.152
p-orec-mixed	0.998	1.003	1.012	0.979	1.002	1.034	1.002	1.017	1.047

	Vacation (low)			Memcached		
Threads	4	8	24	4	8	24
p-lock-lazy	1.035	1.032	1.085	1.005	1.33	1.354
p-orec-lazy	1.012	1.017	0.998	1.057	1.368	1.485
p-norec	1.061	1.087	1.100	0.992	1.256	1.272
p-ring	1.027	1.014	1.017	1.111	1.184	1.631
p-orec-mixed	0.971	0.980	0.991	1.201	1.271	1.321

Table 5.5: Speedup of Deferred Flushing

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 5.4 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.

5.4.4 Deferred Flushing

Our naive transformation from STM to PTM prolongs the time spent holding locks: all `clwb` operations are performed while locks are held. In Listing 21, there are four rounds of interaction with the NVM while locks are held. The benefit of this strategy is simplified recovery: 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 13. As a result, recovery never needs to worry about ordering the outstanding write-backs of two completed transactions.

In our deferred flushing optimization, we replace the *active* status word with an integer timestamp representing the transaction's commit order (0 means *inactive*). Then, we split line 10, such that the write-back occurs *without* `clwb`s. After write-back completes, we release locks (part of line 13), then we issue the `clwb`s, then clear the status, and then run the rest of line 13. 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, the delayed `clwb` will flush the new value.

Essential to the above optimization is the ability to produce a correct total order on writer transaction commits. In all but the TLRW algorithm, some global counter, or single global lock, is used to order all writers. We use it as the timestamp value. In TLRW, we use the CPU's high-resolution timestamp counter (`rdtscp`), which is coherent across cores on the x86.

Table 5.5 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× the unoptimized 24-thread throughput. For some workloads, we observe a small slowdown (up to 3%), due to the shorter critical sections leading to transactions committing in different orders. However, the overall impact is positive.

5.5 Combining Optimizations

We conclude our evaluation by measuring the impact of optimizations, in combination, for each benchmark. We are 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 approximate the worst case, our GP implementations of PTM omit optimizations that are inappropriate

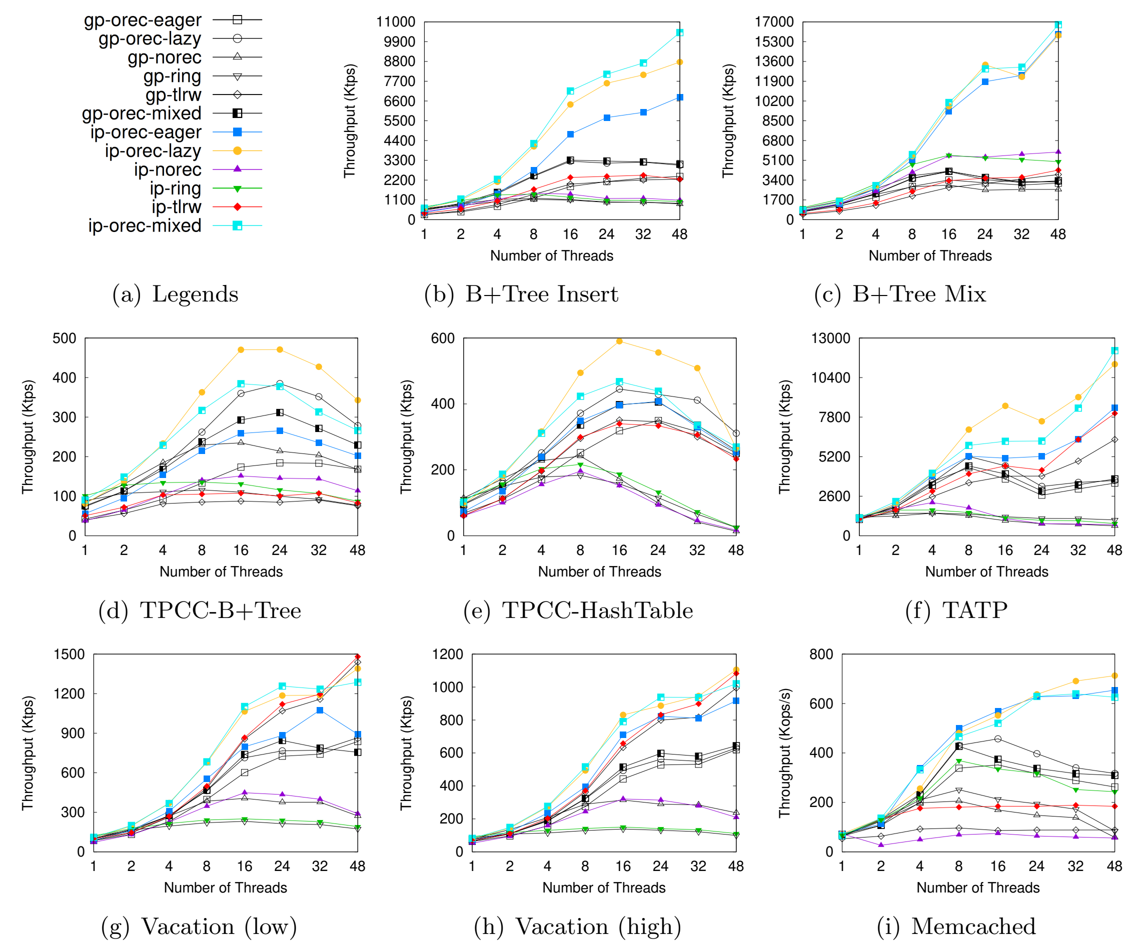


Figure 5.3: Optimized performance of PTM algorithms. The GP prefix indicates that the PTM uses the optimizations that are appropriate in the General Persistence model. IP indicates that additional optimizations were applied, based on the more limited Ideal Persistence model.

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 [20, 116] in requiring quiescence when unmapping a persistent region, so that individual transactions do not need to quiesce.

In summary, 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 the "tlrw" algorithm does not require quiescence.

5.5.1 Performance in the General Persistence Model

In Figure 5.3, algorithms optimized for the GP model are prefixed with `gp`. After applying the optimizations, the overall performance for each algorithm improves, often by a substantial margin. The peak performance speedup of the best choice, when comparing with the naive transformation from STM to PTM (from Figure 5.2), is from 1.1× (b) to 1.3× (i), with a geometric mean of 1.22× across all benchmarks. 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 performance. Note that the decision may not be difficult, since either gp-orec-lazy or gp-tlrw is near the top in every workload. Indeed, if only one algorithm could be used for all programs, gp-orec-lazy appears to provide the best overall performance.

In assessing the improvements to other PTM algorithms, we see that fence pipelining had a significant effect on eager TLRW. Fence pipelining caused 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. Both are appealing, because they achieve privatization safety without quiescence. However, neither could match the performance of gp-orec-lazy at high thread counts. Although the implementation of RingSTM presented in the charts did not scale well, we were able to make it somewhat more competitive at low thread counts by using its relaxed commit order optimization [111]. However, this optimization sacrifices ring's privatization safety, and is offset by a need for quiescence. In the case of NOrec, performance degraded relative to STM. There were two causes. The first is that committing transactions in NOrec (STM) cause active transactions to block. Adding `clwb` instructions to the commit sequence for PTM increases latency at this most critical point. Secondly, logging granularity was not able to improve performance, due to its impact on read set validation for NOrec.

5.5.2 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 [116] occasionally outperforms our orec-lazy, but when orec-lazy performs better, it is by a larger margin, indicating 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$\times$ (d) to 4.06$\times$ (c), with a geometric mean of 1.92$\times$ across all benchmarks.

5.6 Conclusions

In this chapter, we studied the performance of persistent transactional memory (PTM) algorithms. Our study 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, designed to reduce the cost of fences, which are the most significant source of latency in PTM relative to the software transactional memory algorithms (STM) on which they are built.

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 could perform well, especially at low thread counts, whereas in the ideal model, a single algorithm was best. A critical question is whether the ideal model is realistic: at the time of this writing, there are no commercially-available NVM-only systems, but there are also no production-worthy applications that use STM for transactions over DRAM. While we show that PTM over NVM provides compelling performance at as few as two threads, making it a superior choice to lock-based concurrency, the verdict on STM is muddier, and further clouded by incompatibility of hardware TM (HTM) with NVM.

Chapter 6

Conclusions and Future Work

In this dissertation, we applied transactional memory in various real-world applications to demonstrate that existing TM platforms and common parallel programming models are not compatible with each other, and even state-of-the-art TM algorithms are not optimal when transformed to support data persistence. We then proposed, implemented, and evaluated our solutions: In Chapter 2, we showed that non-blocking techniques are not compatible with precise memory reclamation. We then introduced *revocable reservations* to bridge the gap between them. In Chapter 3, we described our experiences employing TLE in two real-world programs. We discovered the effects of quiescence for TLE and proposed language-level support to let programmers dynamically disable quiescence. We also raised questions regarding formalizing the sufficient and necessary conditions for TLE. In Chapter 4, we analyzed the cost for long-running or irrevocable operations in transactions, and introduced *atomic deferral* to allow programmers to move long-running or irrevocable operations out of a transaction while maintaining serializability. In Chapter 5, We demonstrated how to build concurrent persistent transactional memory from traditional software transactional memories and introduced general and programming model-specific optimizations that can substantially improve performance.

All of the above work is targeted at making transactional memory more appealing and applicable, and thus increasing the chance for TM to be used in production code. As for future work, we summarized our previous work and listed the possible directions to expand the work in each chapter.

In Chapter 2, our experiments showed the value of adjusting the number of locations accessed per transaction. We used the thread count as a heuristic, but contention would be a better metric. We plan to explore techniques wherein language-level transactions can reliably and safely expose abort counts and abort causes to the programmer, to enable such optimization in a standards-compliant way. We also found that our concurrent data structures were sensitive to the choice of allocator, with unpredictable and occasionally pathological behaviors. These findings suggest that there is still opportunity to improve on the state of the art in memory allocation, possibly by considering TM-aware allocation strategies and algorithms.

To further improve the work in Chapter 3, we are interested in creating tools for automatically transforming output operations into deferred operations, and studying the relationship between atomic deferral and nested transactions. We are also interested in crafting a more formal correctness argument, which may influence the use of transaction-friendly locks in a greater range of workloads.

For supporting data persistence in Chapter 5, we plan to delve deeper into the relationship between HTM and PTM, with a particular focus on using HTM to prefetch or pre-compute results, even if those results must be flushed using a software protocol. We also plan to look at niche 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 are currently developing 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 when threads are not constrained to a single socket. 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 [52].

Lastly, our experience suggests that much more work is needed before programmers can use TM easily. Library support remains inconsistent, and even a fully-implemented specification is insufficient to address third-party libraries. We encourage continued effort in this direction.

Bibliography

[1] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of Transactional Memory and Automatic Mutual Exclusion. In *Proceedings of the 35th ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January 2008.

[2] Martín Abadi, Tim Harris, and Mojtaba Mehrara. Transactional Memory with Strong Atomicity Using Off-the-Shelf Memory Protection Hardware. In *Proceedings of the 14th PPoPP*, Raleigh, NC, February 2009.

[3] Ali-Reza Adl-Tabatabai, Tatiana Shpeisman, and Justin Gottschlich. Draft Specification of Transactional Language Constructs for C++, February 2012. Version 1.1, http://justingottschlich.com/tm-specification-for-c-v-1-1/.

[4] Alexandro Baldassin, Edson Borin, and Guido Araujo. Performance Implications of Dynamic Memory Allocators on Transactional Memory Systems. In *Proceedings of the 20th PPoPP*, San Francisco, CA, February 2015.

[5] Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zablotchi. Fast and Robust Memory Reclamation for Concurrent Data Structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, Asilomar State Beach, CA, July 2016.

[6] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ Concurrency. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages*, Austin, TX, January 2011.

[7] Emery Berger, Kathryn McKinley, Robert Blumofe, and Paul Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, November 2000.

[8] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th PACT*, Toronto, ON, Canada, October 2008.

[9] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Subtleties of Transactional Memory Atomicity Semantics. *Computer Architecture Letters*, 5(2), November 2006.

[10] Anastasia Braginsky, Alex Kogan, and Erez Petrank. Drop the Anchor: Lightweight Memory Management for Non-Blocking Data Structures. In *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures*, Montreal, Quebec, Canada, July 2013.

[11] Anastasia Braginsky and Erez Petrank. A Lock-Free B+tree. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, Pittsburgh, PA, June 2012.

[12] Trevor Brown. Reclaiming Memory for Lock-Free Data Structures: There has to be a Better Way. In *Proceedings of the 34th ACM Symposium on Principles of Distributed Computing*, Portland, OR, June 2015.

[13] Irina Calciu, Justin Gottschlich, Tatiana Shpeisman, Gilles Pokam, and Maurice Herlihy. Invyswell: A Hybrid Transactional Memory for Haswell's Restricted Transactional Memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*, Edmonton, AB, Canada, August 2014.

[14] Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. The Atomos Transactional Programming Language. In *Proceedings of the 27th PLDI*, June 2006.

[15] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *ACM SIGPLAN Notices*, volume 49, pages 433–452. ACM, 2014.

[16] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. Atlas: Leveraging Locks for Non-Volatile Memory Consistency. In *ACM SIGPLAN Notices*, volume 49, pages 433–452. ACM, 2014.

[17] Joel Coburn, Adrian Caulfield, Ameen Akel, Laura Grupp, Rajesh Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, March 2011.

[18] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM Sigplan Notices*, 46(3):105–118, 2011.

[19] Nachshon Cohen and Erez Petrank. Efficient Memory Management for Lock-Free Data Structures with Optimistic Access. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, Portland, OR, June 2015.

[20] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 271–282. ACM, 2018.

[21] Tyler Crain, Vincent Gramoli, and Michel Raynal. A Speculation-Friendly Binary Search Tree. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, New Orleans, LA, February 2012.

[22] Luke Dalessandro, Francois Carouge, Sean White, Yossi Lev, Mark Moir, Michael Scott, and Michael 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, March 2011.

[23] Luke Dalessandro and Michael Scott. Strong Isolation is a Weak Idea. In *Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing*, Raleigh, NC, February 2009.

[24] Luke Dalessandro, Michael Spear, and Michael L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming*, Bangalore, India, January 2010.

[25] Brian Demsky and Navid Tehrany. Integrating File Operations into Transactional Memory. *Journal of Parallel and Distributed Computing*, 71(10):1293–1304, 2011.

[26] Mathieu Desnoyers, Paul McKenney, Alan Stern, Michel Dagenais, and Jonathan Walpole. User-Level Implementations of Read-Copy Update. *IEEE Transactions on Parallel and Distributed Systems*, 23(2):375–382, 2012.

[27] Dave Dice, Yossi Lev, Virendra Marathe, Mark Moir, Marek Olszewski, and Dan Nussbaum. Simplifying Concurrent Algorithms by Exploiting Hardware TM. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.

[28] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*, Stockholm, Sweden, September 2006.

[29] David Dice and Nir Shavit. TLRW: Return of the Read-Write Lock. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.

[30] Nuno Diegues, Paolo Romano, and Luis Rodrigues. Virtues and Limitations of Commodity Hardware Transactional Memory. In *Proceedings of the 23rd PACT*, Edmonton, AB, Canada, August 2014.

[31] Edsger W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Communications of the ACM*, 8(9):569, 1965.

[32] Siakavaras Dimitrios, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. Combining htm and rcu to implement highly efficient balanced binary search trees. In *Parallel Architectures and Compilation Techniques*, Portland, Oregon, USA, September 2017.

[33] Aleksandar Dragojevic and Tim Harris. STM in the Small: Trading Generality for Performance in Software Transactional Memory. In *Proceedings of the EuroSys2012 Conference*, Bern, Switzerland, April 2012.

[34] Aleksandar Dragojevic, Maurice Herlihy, Yossi Lev, and Mark Moir. On The Power of Hardware Transactional Memory to Simplify Memory Management. In *Proceedings of the 30th ACM Symposium on Principles of Distributed Computing*, San Jose, CA, June 2011.

[35] Aleksandar Dragojevic, Yang Ni, and Ali-Reza Adl-Tabatabai. Optimizing Transactions for Captured Memory. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures*, Calgary, AB, Canada, August 2009.

[36] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking Binary Search Trees. In *Proceedings of the 29th ACM Symposium on Principles of Distributed Computing*, Zurich, Switzerland, July 2010.

[37] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, 1976.

[38] Jason Evans. jemalloc memory allocator, 2017. http://http://jemalloc.net/.

[39] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In *Proceedings of the 13th PPoPP*, Salt Lake City, UT, February 2008.

[40] Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Elastic transactions. In *International Symposium on Distributed Computing*, pages 93–107. Springer, 2009.

[41] Keir Fraser. *Practical Lock-Freedom.* PhD thesis, King's College, University of Cambridge, September 2003.

[42] Free Software Foundation. Transactional Memory in GCC, 2012. http://gcc.gnu.org/wiki/TransactionalMemory.

[43] Free Software Foundation. Transactional Memory in GCC, 2016. http://gcc.gnu.org/wiki/TransactionalMemory.

[44] Jeff Gilchrist. Parallel BZIP2 (PBZIP2) Data Compression Software, 2016. http://compression.ca/pbzip2/.

[45] Ellis R Giles, Kshitij Doshi, and Peter Varman. Softwrap: A lightweight framework for transactional support of storage class memory. In *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, pages 1–14. IEEE, 2015.

[46] Justin Gottschlich and Hans-J. Boehm. Generic Programming Needs Transactional Memory. In *Proceedings of the 8th ACM SIGPLAN Workshop on Transactional Computing*, Houston, TX, March 2013.

[47] Vincent Gramoli. More Than You Ever Wanted to Know about Synchronization. In *Proceedings of the 20th PPoPP*, San Francisco, CA, February 2015.

[48] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a Theory of Transactional Contention Managers. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, Las Vegas, NV, July 2005.

[49] Rachid Guerraoui and Michal Kapalka. On the Correctness of Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, February 2008.

[50] Tim Harris. A Pragmatic Implementation of Non-Blocking Linked Lists. In *Proceedings of the 15th International Symposium on Distributed Computing*, Lisbon, Portugal, October 2001.

[51] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable Memory Transactions. In *Proceedings of the 10th PPoPP*, Chicago, IL, June 2005.

[52] Tim Harris, Mark Plesko, Avraham Shinar, and David Tarditi. Optimizing Memory Transactions. In *Proceedings of the 27th ACM Conference on Programming Language Design and Implementation*, Ottawa, ON, Canada, June 2006.

[53] Maurice Herlihy and Eric Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the 13th PPoPP*, Salt Lake City, UT, February 2008.

[54] Maurice P. Herlihy, Victor Luchangco, and Mark Moir. Obstruction free synchronization: Double-ended queues as an example. In *Proceedings of 23rd International Conference on Distributed Computing Systems*, May 2003.

[55] Maurice P. Herlihy, Victor Luchangco, Mark Moir, and William 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, July 2003.

[56] Maurice P. Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th ISCA*, San Diego, CA, May 1993.

[57] Richard L. Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin Hertzberg. A Scalable Transactional Memory Allocator. In *Proceedings of the International Symposium on Memory Management*, Ottawa, ON, Canada, June 2006.

[58] Intel. NVDIMM Block Window Driver Writer's Guide. http://pmem.io/documents/NVDIMM_DriverWritersGuide-July-2016.pdf.

[59] Intel Corporation. Nvml: Implementing persistent memory applications. https://www.snia.org/sites/default/files/.

[60] Intel Corporation. Intel Architecture Instruction Set Extensions Programming (Chapter 8: Transactional Synchronization Extensions). February 2012.

[61] ISO/IEC JTC 1/SC 22/WG 21. Technical Specification for C++ Extensions for Transactional Memory, May 2015.

[62] Louis Jenkins, Tingzhe Zhou, and Michael F. Spear. Redesigning go's built-in map to support concurrent operations. In *Parallel Architectures and Compilation Techniques*, PACT'17, Portland, Oregon, USA, September 2017.

[63] Tomas Karnagel, Roman Dementiev, Ravi Rajwar, Konrad Lai, Thomas Legler, Benjamin Schlegel, and Wolfgang Lehner. Improving In-Memory Database Index Performance with Intel Transactional Synchronization Extensions. In *Proceedings of the 20th HPCA*, Orlando, FL, February 2014.

[64] Gokcen Kestor, Osman Unsal, Adrian Cristal, and Serdar Tasiran. T-Rex: A Dynamic Race Detection Tool for C/C++ Transactional Memory Applications. In *Proceedings of the EuroSys2014 Conference*, Amsterdam, The Netherlands, April 2014.

[65] Matthew Kilgore, Stephen Louie, Chao Wang, Tingzhe Zhou, Wenjia Ruan, Yujie Liu, , and Michael Spear. Transactional Tools for the Third Decade. In *Proceedings of the 10th TRANSACT*, Portland, OR, June 2015.

[66] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. High-performance transactions for persistent memories. *ACM SIGOPS Operating Systems Review*, 50(2):399–411, 2016.

[67] Benjamin C Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. Phase-change technology and the future of main memory. *IEEE micro*, 30(1), 2010.

[68] Yossi Lev, Victor Luchangco, Virendra Marathe, Mark Moir, Dan Nussbaum, and Marek Olszewski. Anatomy of a Scalable Software Transactional Memory. In *Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing*, Raleigh, NC, February 2009.

[69] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, and Jinglei Ren. Dudetm: Building durable transactions with decoupling for persistent memory. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Xi'an, China, April 2017.

[70] Yujie Liu, Stephan Diestelhorst, and Michael Spear. Delegation and Nesting in Best Effort Hardware Transactional Memory. In *Proceedings of the 24th SPAA*, Pittsburgh, PA, June 2012.

[71] Yujie Liu, Victor Luchangco, and Michael Spear. Mindicators: A Scalable Approach to Quiescence. In *Proceedings of 33rd International Conference on Distributed Computing Systems*, Philadelphia, PA, July 2013.

[72] Yujie Liu and Michael Spear. Toxic Transactions. In *Proceedings of the 6th ACM SIGPLAN Workshop on Transactional Computing*, San Jose, CA, June 2011.

[73] Yujie Liu and Michael Spear. Mounds: Array-Based Concurrent Priority Queues. In *Proceedings of the 41st International Conference on Parallel Processing*, Pittsburgh, PA, September 2012.

[74] Yujie Liu, Kunlong Zhang, and Michael Spear. Dynamic-Sized Nonblocking Hash Tables. In *Proceedings of the 33rd ACM Symposium on Principles of Distributed Computing*, Paris, France, July 2014.

[75] Yujie Liu, Tingzhe Zhou, and Michael Spear. Transactional Acceleration of Concurrent Data Structures. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, Portland, OR, June 2015.

[76] Victor Luchangco, Mark Moir, and Nir Shavit. Nonblocking k-compare-single-swap. In *Proceedings of the 15th ACM Symposium on Parallel Algorithms and Architectures*, San Diego, CA, June 2003.

[77] Virendra Marathe, Achin Mishra, Amee Trivedi, Yihe Huang, Faisal Zaghloul, Sanidhya Kashyap, Margo Seltzer, Tim Harris, Steve Byan, Bill Bridge, and Dave Dice. Persistent memory transactions. In *arXiv preprint arXiv:1804.00701*, 2018.

[78] Alexander Matveev and Nir 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, March 2015.

[79] Paul E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels.* PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004.

[80] memcached.org. Memcached, 2014. http://memcached.org/.

[81] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard Hudson, Bratin Saha, and Adam Welc. Practical Weak-Atomicity Semantics for Java STM. In *Proceedings of the 20th SPAA*, Munich, Germany, June 2008.

[82] Maged Michael. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures*, Winnipeg, Manitoba, Canada, August 2002.

[83] Maged Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004.

[84] Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, May 1996.

[85] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford Transactional Applications for Multi-processing. In *Proceedings of the IEEE*

International Symposium on Workload Characterization, Seattle, WA, September 2008.

[86] Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged M. Michael, and Hisanobu Tomari. Quantitative Comparison of Hardware Transactional Memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In *Proceedings of the 42nd ISCA*, Portland, OR, June 2015.

[87] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly 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*, ASPLOS'17, Xi'an, China, April 2017.

[88] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, Orlando, FL, February 2014.

[89] Newsroom, Intel. Intel and Micron produce breakthrough memory technology, 2018. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[90] Yang Ni, Vijay Menon, Ali-Reza Adl-Tabatabai, Antony Hosking, Rick Hudson, Eliot Moss, Bratin Saha, and Tatiana Shpeisman. Open Nesting in Software Transactional Memory. In *Proceedings of the 12th PPoPP*, San Jose, CA, March 2007.

[91] Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. Design and Implementation of Transactional Constructs for C/C++. In *Proceedings of the 23rd OOPSLA*, Nashville, TN, USA, October 2008.

[92] Marek Olszewski, Jeremy Cutler, and J. Gregory 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, September 2007.

[93] Parabola Research. HEVC Wavefront Animation, December 2013. https://www.parabolaresearch.com/blog/2013-12-01-hevc-wavefront-animation.html.

[94] Martin Pohlack and Stephan Diestelhorst. From Lightweight Hardware Transactional Memory to Lightweight Lock Elision. In *Proceedings of the 6th ACM SIGPLAN Workshop on Transactional Computing*, San Jose, CA, June 2011.

[95] William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33:668–676, June 1990.

[96] Ravi Rajwar and James R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34th IEEE/ACM International Symposium on Microarchitecture*, Austin, TX, December 2001.

[97] Jinglei Ren, Qingda Hu, Samira Khan, and Thomas Moscibroda. Programming for non-volatile main memory is hard. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, September 2017.

[98] Torvald Riegel, Christof Fetzer, and Pascal Felber. Automatic Data Partitioning in Software Transactional Memories. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[99] Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Aditya Bhandari, and Emmett Witchel. TxLinux: Using and Managing Transactional Memory in an Operating System. In *Proceedings of the 21st SOSP*, Stevenson, WA, October 2007.

[100] Amitabha Roy, Steven Hand, and Tim Harris. A Runtime System for Software Lock Elision. In *Proceedings of the EuroSys2009 Conference*, Nuremberg, Germany, March 2009.

[101] Wenjia Ruan and Michael Spear. Hybrid Transactional Memory Revisited. In *Proceedings of the 29th International Symposium on Distributed Computing*, Tokyo, Japan, October 2015.

[102] Wenjia Ruan, Trilok Vyas, Yujie Liu, and Michael Spear. Transactionalizing Legacy Code: An Experience Report Using GCC and Memcached. In *Proceedings of the 19th ASPLOS*, Salt Lake City, UT, March 2014.

[103] William N. Scherer III and Michael 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, July 2005.

[104] Seunghee Shin, James Tuck, and Yan Solihin. Hiding the long latency of persist barriers using speculative execution. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, Toronto, ON, Canada, June 2017.

[105] Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Robert Geva, Yang Ni, and Adam Welc. Towards Transactional Memory Semantics for C++. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures*, Calgary, AB, Canada, August 2009.

[106] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Kate Moore, and Bratin Saha. Enforcing Isolation and Ordering in STM. In *Proceedings of the 2007 ACM Conference on Programming Language Design and Implementation*, San Diego, CA, June 2007.

[107] Simo, N. and Antoni, W. and Markk, M. and Vilho, R. Tpc- c benchmark v5, 2015.

[108] Michael Spear, Luke Dalessandro, Virendra J. Marathe, and Michael 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, February 2009.

[109] Michael Spear, Virendra Marathe, Luke Dalessandro, and Michael Scott. Privatization Techniques for Software Transactional Memory (POSTER). In *Proceedings of the 26th PODC*, Portland, OR, August 2007.

[110] Michael Spear, Maged M. Michael, and Michael L. Scott. Inevitability Mechanisms for Software Transactional Memory. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Transactional Computing*, Salt Lake City, UT, February 2008.

[111] Michael Spear, Maged M. Michael, and Christoph von Praun. RingSTM: Scalable Transactions with a Single Atomic Instruction. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[112] Michael Spear, Michael Silverman, Luke Dalessandro, Maged M. Michael, and Michael L. Scott. Implementing and Exploiting Inevitability in Software Transactional Memory. In *Proceedings of the 37th International Conference on Parallel Processing*, Portland, OR, September 2008.

[113] THE TRANSACTION PROCESSING COUNCIL. Tpc- c benchmark v5, 2015.

[114] AA Tulapurkar, Y Suzuki, A Fukushima, H Kubota, H Maehara, K Tsunekawa, DD Djayaprawira, N Watanabe, and S Yuasa. Spin-torque diode effect in magnetic tunnel junctions. *Nature*, 438(7066):339, 2005.

[115] Haris Volos, Andres Jaan Tack, Neelam Goyal, Michael Swift, and Adam Welc. xCalls: Safe I/O in Memory Transactions. In *Proceedings of the EuroSys2009 Conference*, Nuremberg, Germany, March 2009.

[116] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *ACM SIGARCH Computer Architecture News*, March 2011.

[117] Chao Wang, Yujie Liu, and Michael Spear. Transaction-Friendly Condition Variables. In *Proceedings of the 26th SPAA*, Prague, Czech Republic, June 2014.

[118] Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. Irrevocable Transactions and their Applications. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[119] Michael Widenius and David Axmark. *MySQL reference manual: documentation from the source.* " O'Reilly Media, Inc.", 2002.

[120] Wang Xin, Weihua Zhang, Zhaoguo Wang, Ziyun Wei, Haibo Chen, and Wenyun Zhao. Eunomia: Scaling concurrent search trees under contention using htm. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Austin, Texas, United States, February 2017.

[121] Richard Yoo, Christopher Hughes, Konrad Lai, and Ravi Rajwar. Performance Evaluation of Intel Transactional Synchronization Extensions for High Performance Computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver, CO, November 2013.

[122] Richard Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin Lee. Kicking the Tires of Software Transactional Memory: Why the Going Gets Tough. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[123] Pantea Zardoshti, Tingzhe Zhou, Pavithra Balaji, Michael L. Scott, and Michael Spear. Simplifying Transactional Memory Support in C++. *ACM Transactions on Architecture and Code Optimization*, 1, 2019.

[124] Tingzhe Zhou, Victor Luchangco, and Michael Spear. Brief announcement: Extending transactional memory with atomic deferral. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, Washington, DC, USA, July 2017.

[125] Tingzhe Zhou, Victor Luchangco, and Michael Spear. Extending transactional memory with atomic deferral. In *The 21st International Conference on Principles of Distributed Systems*, Lisboa, Portugal, December 2017.

[126] Tingzhe Zhou, Victor Luchangco, and Michael Spear. Hand-over-hand transactions with precise memory reclamation. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, Washington, DC, USA, July 2017.

[127] Tingzhe Zhou and Michael Spear. The Mimir Approach to Transactional Output. In *Proceedings of the 11th TRANSACT*, Barcelona, Spain, March 2016.

[128] Tingzhe Zhou, PanteA Zardoshti, and Michael Spear. Practical Experience with Transactional Lock Elision. In *Proceedings of the 12th ACM SIGPLAN Workshop on Transactional Computing*, Austin, TX, February 2017.

[129] Tingzhe Zhou, PanteA Zardoshti, and Michael Spear. Practical Experience with Transactional Lock Elision. In *Proceedings of the 46th International Conference on Parallel Processing*, Bristol, UK, August 2017.

[130] Tingzhe Zhou, Pantea Zardoshti, and Michael Spear. Brief announcement: Optimizing persistent transactions. In *Proceedings of the 31st ACM Symposium on Parallelism in Algorithms and Architectures*, Phoenix, AZ, USA, June 22–24 2019.

[131] Craig Zilles and Lee Baugh. Extending Hardware Transactional Memory to Support Non-Busy Waiting and Non-Transactional Actions. In *Proceedings of the 1st TRANSACT*, Ottawa, ON, Canada, June 2006.

[132] Ferad Zyulkyarov, Vladimir Gajinov, Osman Unsal, Adrian Cristal, Eduard Ayguade, Tim Harris, and Mateo Valero. Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server. In *Proceedings of the 14th PPoPP*, Raleigh, NC, February 2009.

Biography

Tingzhe Zhou received his Bachelor of Engineering in Computer Science and Technology from Wuhan University of Science and Technology in 2012. He was also awarded a Principal's Medal in the same year, which was established to recognize the top ten students who have made a significant contribution to the University. Then he attended to Huazhong University of Science and Technology as a research assistant in Cluster and Grid Computing Lab majoring in Distributed Systems. In 2014, Tingzhe joined the Computer Science and Engineering Department at Lehigh University as a Ph.D. student. He was received his Master of Science Degree in May 2018. He will award the degree of Doctor of Philosophy in Computer Science from Lehigh University in May 2019. He has published in many venues including SPAA, ICPP, PACT, OPODIS, CCS, ACM TACO.