The Dissertation Committee for Christopher J. Rossbach
certifies that this is the approved version of the following dissertation:

Hardware Transactional Memory: A Systems Perspective

Committee:

Emmett Witchel, Supervisor

Michael Dahlin

Doug Burger

Yale Patt

Mark Hill

Hardware Transactional Memory: A Systems Perspective

by

Christopher J. Rossbach, B.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2009

To Suzannne, who will undoubtedly find this a delightful read...

Acknowledgments

My gratitude to the Operating Systems and Architecture group at UT cannot be overstated: research of this kind can only be conducted when there is a synergy of great minds working together. In particular, thanks to Hany Ramadan, Don Porter and Owen Hofmann for bringing their great minds to that synergy. My advisor Emmett Witchel deserves acknowledgement for a great many more things than will fit on this page. In light of that, and in keeping with the spirit of "less-is-more", I'll leave it at this: thank you, Emmett.

Christopher J. Rossbach

The University of Texas at Austin
August 2009

Hardware Transactional Memory: A Systems Perspective

Publication No. ____________________

Christopher J. Rossbach, Ph.D.
The University of Texas at Austin, 2009

Supervisor: Emmett Witchel

The increasing ubiquity of chip multiprocessor machines has made the need for accessible approaches to parallel programming all the more urgent. The current state of the art, based on threads and locks, requires the programmer to use mutual exclusion to protect shared resources, enforce invariants, and maintain consistency constraints. Despite decades of research effort, this approach remains fraught with difficulty. Lock-based programming is complex and error-prone, largely due to well-known problems such as deadlock, priority inversion, and poor composability. Tradeoffs between performance and complexity for locks remain unnattractive. Coarse-grain locking is simple but introduces artificial sharing, needless serialization, and yields poor performance. Fine-grain locking can address these issues, but at a significant cost in complexity and maintainability.

Transactional memory has emerged as a technology with the potential to address this need for better parallel programming tools. Transactions provide the abstraction of isolated, atomic execution of critical sections. The programmer specifies regions of code which access shared data, and the system is responsiblefor executing that code in a way that is isolated and atomic. The programmer need not reason about locks and threads. Transactional memory removes many of the pitfalls of locking: transactions are livelock- and deadlock-free and may be composed freely. Hardware transactional memory, which is the focus of this thesis, provides an efficient implementation of the TM abstraction.

This thesis explores several key aspects of supporting hardware transactional memory (HTM): operating systems support and integration, architectural, design, and implementation considerations, and programmer-transparent techniques to improve HTM performance in the presence of contention. Using and supporting HTM in an OS requires innovation in both the OS and the architecture, but enables practical approaches and solutions to some long-standing OS problems. Innovations in transactional cache coherence protocols enable HTM support in the presence of multi-level cache hierarchies, rich HTM semantics such as suspend/resume and multiple transactions per thread context, and can provide the building blocks for support of flexible contention management policies without the need to trap to software handlers. We demonstrate a programmer-transparent hardware technique for using dependences between transactions to commit conflicting transactions, and suggest techniques to allow conflicting transactions to avoid performance-sapping restarts without using heuristics such as backoff. Both mechanisms yield better performance for workloads that have significant write-sharing.

Finally, in the context of the MetaTM HTM model, this thesis contributes a

high-fidelity cross-design comparison of representative proposals from the literature: the result is a comprehensive exploration of the HTM design space that compares the behavior of models of MetaTM [70, 75], LogTM [58, 94], and Sun's Rock [22].

Contents

List of Tables

List of Figures

Chapter 1

Introduction

For the past few decades, the value-proposition of the computer hardware industry has depended on Moore's Law [57] feature size and clock frequency scaling in subsequent generations of processor chips has historically guaranteed that subsequent generations of software would run faster, with minimal (if any) additional programmer effort. The industry has reached a point where traditional rates of performance improvement due to scaling clock speed and techniques to exploit instruction-level-parallelism can no longer be expected. Consequently, current trends in architecture reflect a change of focus: hardware manufacturers have shifted their efforts from scaling clock speed and mechanisms to exploit instruction-level-parallelism, toward scaling the number of cores on a chip. Chip multiprocessors are increasingly ubiquitous. A side effect of this change is that improving performance on new generations of hardware requires programmers to take advantage of the parallelism made available by multiple processing contexts. Programmers will be required to develop parallel code, and assume far more responsibility for performance than has been the norm during the previous era of dramatic scaling due to ILP. Accessible approaches to parallel programming are urgently needed.

The current state of the art in achieving concurrency using parallel programming relies heavily on threads and locks. Multiple sequential flows of control (threads) execute at the same time coordinating access to shared resources using locks to guarantee mutually exclusive access to critical sections. Despite decades of research, parallel programming using threads and locks remains quite difficult, even for experienced programmers. Locks complicate the lives of programmers in a number of well-known ways, for example, deadlocks, convoys, and priority inversion. Locks compose poorly and require complex ordering disciplines when multiple locks must be coordinated. The performance-complexity trade-off associated with locks remains a Hobson's choice: coarse-grain locking is simple to reason about but sacrifices concurrent performance. Fine-grain locking may enable high performance, but it makes code more complex and harder to maintain because it is dependent on invariants that can be difficult to express or enforce. Transactional memory has been the focus of much recent research attention because it is free of many of pitfalls of locks, and because optimistic implementation techniques can allow transactional memory to provide the performance of fine-grain locking with the code complexity of coarse-grain locking.

Transactional memory can greatly simplify parallel programming [76]. A programmer demarcates critical sections that may access shared data as *transactions*, which are sequences of memory operations that either execute completely (commit) or have no effect (abort). The system is responsible for ensuring that transactions execute *atomically* (either completely, or not at all), and in *isolation*, meaning that a transaction cannot see the effects of other active transactions, and it's own operations are not visible in the system until it commits. While transactions provide the abstraction of completely serial execution of critical sections, the system actually executes them optimistically, allowing multiple transactions to proceed concurrently, as long as atomicity and isolation are not violated. The programmer benefits because the system provides atomicity: reasoning about partial failures in critical sections is no longer necessary. Because transactions can be composed, and do not suffer from deadlock, programmers can freely compose thread-safe libraries based on transactions.

Hardware transactional memory (HTM) provides an efficient hardware implementation of the TM abstraction. Implementation techniques vary [**?**,2,5,7,9,10, 12,13,15,18,19,22,36,39,45,46,48,58,68,70,71,85,94,96], but in general approaches rely on simple extensions to to the ISA, as well as modifications to upper-level caches, and coherence protocols to provide the basic mechanisms required in a TM system: version management, and conflict detection. At a high level, HTM systems operate by speculating that critical section will complete without conflicting memory operations–the HTM will fail and retry speculative executions if cache coherence traffic indicate that read-write or write-write sharing occurs at run-time.

This thesis takes a systems-level perspective on the design, use, and management of hardware transactional memory. It explores the architectural and software support required to allow an operating system to use HTM for synchronization, as well as requirements for an OS to support user programs that use HTM. The thesis proposes mechanisms that can improve performance of HTMs under contention, contributing both programmer-transparent techniques, as well as techniques that allow software and hardware to collaborate to avoid contention before it occurs, and manage it well when it does occur. We consider the design space for HTM implementation as a whole, and contribute a comprehensive cross-design comparison for HTM proposals in the literature.

1.1 Operating Systems and HTM

Operating systems can benefit from using transactional memory, because TM provides a simpler programming model than locks. Operating systems rely on lock ordering disciplines to avoid deadlock: these disciplines become complex over time and are difficult for programmers to master. Transactions, by contrast, require no ordering disciplines. Because many applications spend a significant fraction of their runtime in the kernel (by making system calls, e.g., to read and write files), another benefit of TM in the OS is that increased concurrency due to optimism in OS synchronization can translate to increased performance for user programs without having to modify or recompile them.

Management and support of HTM in an operating system requires innovation both in the architecture and the operating system. Transactions cannot simply replace or eliminate locks in an operating system for two main reasons. The first is that many kernel critical sections perform I/O, actually changing the state of devices like the disk or network card. I/O is a problem for transactional memory because TM systems assume that if a conflict occurs, one transaction can be aborted, rolled back to its start, and re-executed. However, when the OS performs I/O it actually changes the state of a device (e.g., by writing data to the network). Most devices cannot revert to a previous state once a write operation completes, so a transaction that performs I/O cannot be rolled back and re-executed. The second reason is that some kernel critical sections are highly contended and currently locks are more efficient than transactions for highly contended critical sections. Under contention, the optimism of transactions is unwarranted and the rollbacks and backoff performed by the transactional memory system can significantly reduce performance. Operating systems will need to be able to use locks in some contexts, even in the presence of HTM support.

The *cxspinlock* (cooperative transactional spinlock), a contribution of this thesis, is a synchronization primitive that addresses the problem of I/O in transactions, allowing locks and transactions to work together to protect the same data while maintaining both of their advantages. Previous HTM proposals require every execution of a critical section to be protected by either a lock or a transaction, while cxspinlocks allow a critical section or a data structure accessed from different critical sections to sometimes be protected by a lock and sometimes by a transaction. Cxspinlocks dynamically and automatically choose between locks and transactions. Cxspinlocks attempt to execute critical sections as transactions by default, but when the processor detects an I/O attempt, the transaction is rolled back, and the cxspinlock will ensure that the thread re-executes the critical section exclusively, blocking other transactional and non-transactional threads. Additionally, cxspinlocks provide a convenient API for converting lock-based code to use transactions.

HTM enables a solution to the long-standing problem of priority inversion due to locks. Priority inversion occurs when a high priority thread waits for a lock held by a low priority thread. We demonstrate the modifications necessary in the TxLinux scheduler and the transactional memory hardware to nearly eliminate priority inversion. Moreover, the OS can improve its scheduling algorithms to help manage high contention by leveraging a thread's transaction history to calculate the thread's dynamic priority or deschedule conflicting threads.

1.2 Avoiding and managing contention

The TM research community has long held that conflicts are rare in transactional memory workloads: the suggestion is tenuous at best given the paucity of HTM implementations and real transactional memory workloads. There is evidence that contention may not be rare in TM workloads. For example, Shriraman et al. find that as many as 90% of transactions conflict in some STAMP [54] benchmarks and up to 70% conflict in STMBench7 [28]. Work with TxLinux [70] showed that even OS synchronization, typified by very small critical sections (hundreds of instructions), can show moderate contention rates (10-20%) and occasional pathologically high contention [75]. Assuming that contention is rare makes points in the design space that improve *uncontended* performance at the expense of contended performance artificially attractive. If the promise of TM is simpler programming, HTM implementations must perform well in the common case, but must degrade gracefully when sharing occurs. To this end, this thesis propose mechanisms that improve HTM performance under contention transparently, as well as mechanisms that allow programmers to avoid contention and manage it better when it actually does arise.

1.2.1 Dependence Aware HTM

Most transactional memory systems detect conflicts between two transactions and respond by forcing one of the transactions to restart or block. By restarting or blocking on conflict, TM implementations provide a level of concurrency that is equivalent to that of *two-phase locking* [27], becuase any data read or written by one transaction has an implicit lock on it that conflicts with any attempt to write the same data. Using conflict serializability as the system's safety property increases concurrency relative to using two-phase locking.

Dependence aware transactional memory (*DATM*) is a transactional memory implementation technique that ensures conflict serializability. DATM manages conflicts by commuting them to dependences that constrain commit order, and when possible, forwards speculative data values between uncommitted transactions. While previous HTM proposals cannot commit **any** conflicting transactions [1], even if the schedule yields a correct result, dependence-awareness allows conflicting transactions that are conflict-serializable to commit safely. This ability minimizes wasted work in the form of restarts, increases throughput, and translates to better utilization of parallel hardware than current TM systems. Dependence-awareness is *safe*—transactions have the same consistency guarantees, and still provide the same abstraction of atomic and isolated execution that conventional TM designs provide.

Dependence awareness provides is most beneficial for workloads that have

[1]Except potentially by stalling accesses, a mechanism called *stall-on-conflict* [58]

write-shared data (hotspots). Previously proposed mechanisms that improve TM performance for write-shared data (e.g. open nesting, open nesting [60, 64], early release [88], boosting [34], etc.) complicate the programming model and are touted as the purview of expert programmers. Dependence-awareness, by contrast, is transparent to the programmer. Transparency is particularly important because many common data structures, like shared counters and linked lists, have write-shared data that cause performance problems in conventional TM systems.

This thesis contributes a hardware design for Dependence Aware Transactional memory [71], providing a programmer-transparent mechanism to improve TM performance under high contention.

1.2.2 Notifying transactions and transaction annotation

If HTM must degrade gracefully under contention, they must not only avoid needless restarts, but must support mechanisms that help minimize performance lost to conflicts that cannot be resolved by any other way than restarting. Contention management and backoff policies are the *de facto* state of the art for handling conflicts between transactions. A contention manager is an abstraction responsible for deciding which transaction(s) must restart when a conflict occurs, and backoff policies govern when a transaction that has lost a conflict should retry. Both of these mechanisms have a first-order impact on performance, and both mechanisms are fundamentally heuristic. Since a decision must always be made about how to handle a conflict, contention management is a critical component of an HTM implementation: HTM must support contention management decisions that are both flexible and fast. The same does not necessarily hold for backoff. While it has been shown that backoff can reduce contention, and that exponential backoff in particular can avoid livelock [84], backoff can itself lead to performance pathologies [75]. HTM should support techniques that help software avoid contention by eliminating the need to use heuristic approaches such as backoff to handle retry after conflicts.

This thesis presents TagTM, a new HTM design that makes flexible contention management efficient in an HTM, and supports mechanisms to help software avoid conflicts. The techniques TagTM uses to improve performance under contention are applicable to any HTM design that relies on caches for version management and coherence for conflict detection. TagTM relies on *notifying transactions*, a mechanism for handling repeated transactional conflicts more effectively than backoff. A novel technique called *transaction annotation* helps TagTM provide user-defined contention management policies by allowing software to tag transactions with metadata that can be used by a hardware contention manager. TagTM uses a new transaction-aware coherence protocol called **XMESI** that extends transactional coherence to support powerful mechanisms such as transaction annotation and notifying transactions. To evaluate TagTM, we implement and evaluate several representative designs from the literature, including MetaTM [70, 75], LogTM [58], and Sun's Rock [22], and provide a comprehensive cross-design comparison.

1.3 HTM design space

The HTM design space is densely populated with proposals [?, 5, 7, 9, 10, 12, 13, 18, 19, 22, 36, 39, 48, 58, 68, 70, 71, 85, 94, 96] that address the general structure of TM support, exploring different different microarchitectural implementations, mechanisms for version management and conflict detection, contention management policies, and virtualization techniques. Yet despite many studies which subject particular dimensions of the design space to rigorous scrutiny, it remains difficult to compare different designs directly. Published results often rely on specialized assumptions or environments, reflect different simulation and simulated platforms, and often use different benchmarks as well.

This thesis explores the implementation details of HTM that have an important impact on the performance and programmability of the HTM system. Implementation techniques based on coherence and upper level caches are an attractive focus because implementing HTM using coherence is incremental over existing coherence mechanisms, making them a likely candidate for actual silicon implementation. The goal of this section of the thesis is to consider outstanding implementation details associated with transactional coherence protocols as well as to address the problem of comparability for different HTM designs. Microarchitectural parameters, such as memory hierarchy depth, semantics of prefetch instructions, and limited memory bandwidth, as well as architectural choices, such as support for suspending transactions, strong isolation, and multiple active transactions per thread, can have a first order impact on performance. This thesis introduces a new transactional coherence protocol called XMESI that is robust to these architectural parameters, and extends the MetaTM HTM model [70] to support high fidelity models of many representative candidates from the literature. The resulting contribution is a cross-product design-level comparison over MetaTM, DATM [71], LogTM [58], Rock [22], and FlexTM [86, 87].

1.4 A note on the relation of this thesis to published work, and contributions of others

Two chapters of this thesis are significantly related to my previously published papers: “TxLinux: Using and Managing Hardware Transactional Memory in an Operating System” [75], and “Dependence Aware Transactional Memory for Increased Concurrency” [71]. Chapters 3 and 4 both draw from, and expand on the ideas expressed in those papers: I consider these mechanisms in much greater detail in this thesis than is practical in a conference format. The remaining chapters in this thesis represent work that is unpublished in other venues at the time of this writing.

Because research of this scale requires a synergy of minds, I have retained the use of the first-person plural perspective in this thesis. However, to avoid the risk of hiding the contributions of others behind this gesture, I will make the contributions of others explicit. Development of the first version of MetaTM, and the original conversion of Linux into TxLinux (see chapter 3) was the collective work of myself, Hany Ramadan, Owen Hofmann, and Don Porter. While work and innovation in integration of the Linux scheduler to use transactional memory was a solo effort, `cxspinlocks` are the result of the collective efforts of that same group. The dependence-aware model of transactions owes a debt to Hany Ramadan, who developed a software-based implementation and a proof of it's safety in [72]; however the hardware design, implementation, and evaluation of DATM described in chapter 4 are entirely my own work. TagTM (chapter 5) and the cross-product design comparison (chapter 6) are also the results of solo research effort.

The rest of this thesis is organized as follows. Chapter 3 considers the problem of operating system support and use of hardware transactional memory. Chapter 4 explores a hardware design for dependence-aware transactional memory and chapter 5 considers techiniques for handling contention. Chapter 5 explores microarchitectural details for HTM implementation, and chapter 6 presents a detailed cross-product design comparison for representative HTM proposals. Chapter 7 reviews related work, and chapter 8 concludes.

Chapter 2

Background

The problem of how to write concurrent programs remains an active area of research, with diverse approaches tailored to diverse platforms and environments. This thesis is concerned with concurrent programming in a shared-memory model, where multiple threads of control execute concurrently, sharing data and communicating through memory in a single address space. In this model, the need to synchronize accesses to shared resources arises, because different threads may read and write data over which program-specific consistency constraints must be maintained. For example, a program may insert an element into a linked list by setting the `next` pointer of the element before the insertion point to point to the new element, and subsequently set the `next` pointer of the inserted element to point the element after the insertion point. Reads and writes to different cells in memory must occur to complete this update (different `next` pointer locations), and in the presence of concurrency, failure to *synchronize* or *coordinate* these operations can cause a program to yield incorrect results. In the example above, if a second thread traverses the list by reading next pointers at a time that occurs logically between the first and second update, the second thread can follow a meaningless pointer or fail to see the remainder of the list after the insertion point. Synchronization primitives provide a tool for programmers to express the need for and enforce consistency in the presence of sharing.

Transactional memory is a synchronization primitive and a programming abstraction that addresses this need.

2.1 Locks

The state of the art in concurrent programming to-date has relied on *mutual exclusion* in general, and on locks in particular to synchronize multiple sequential flows of control. When using locks, a programmer defines *critical sections*, or regions of code over which consistency constraints must be maintained. Mutual exclusion ensures that only one individual thread of control can execute the critical section at a given time. Locks can be used to enforce mutual exclusion by requiring that a thread must hold the lock in order to access the critical section: if the lock is unavailable (indicating the presence of another thread in the critical section), that thread must wait. Variations on this basic approach to synchronization and coordination are myriad. For example, reader/writer locks [53] allow many readers or a single writer into a critical section, semaphores [24] enforce an upper bound on the number of threads allowed in a critical section, barriers [51] require that many threads all reach a known point in the code path before any individual thread can proceed, and so on. The ecosystem of primitives that fundamentally rely on locks and mutual exclusion is rich and diverse, including spinlocks, mutexes/semaphores, condition variables and monitors, read-copy-update (RCU) [50], sequence locks, and so on.

Because locks can guarantee mutually exclusive access to shared resources, they are an effective tool for enforcing consistency. However, programming using threads and locks remains quite difficult, even for experienced programmers. First and foremost, locks do not express the needs of the programmer (consistency) directly. Programmers need *consistent operations on shared data*: mutual exclusion is a technique, but is not the high level goal. Moreover, while consistency may be maintained by mutually exclusive access to shared resources, it is often too conservative because locks are pessimistic. All threads are forced to serialize even if the conditions under which inconsistency may arise are rare. For example, if access to a hash table is protected by a single lock, threads that require access to disjoint buckets in the table are forced to serialize despite the fact that no inconsistency can arise from allowing those threads to proceed concurrently. In general, the use of a single lock to protect complex data (called *coarse locking*) leads to unattractive performance tradeoffs. In the example of the hash table, needless serialization can be addressed by using per-bucket locks (an example of *fine-grain locking*). However, the increase in complexity and the number of locks gives rise to a yet another set of difficult problems. Threads may need to hold multiple locks, and failure to acquire them in a globally consistent order can result in deadlock (a cycle in the waits-for graph). Lock ordering disciplines, which address this problem, can be difficult to express and maintain. The problem is compounded when programs may use library code that may acquire locks whose presence is transparent to the programmer. Because the programmer cannot use lock ordering to avoid deadlock on locks that she has no access to, locks *compose poorly*, and the obvious solution of exposing such locks compromises modularity. Locks can cause performance pathologies in systems when they are highly contended and become bottlenecks. Locks can cause priority inversion (when a higher priority thread must wait for a lower priority thread holding a lock it needs) and convoys (when a single thread holds a lock that many threads need). Collectively, these issues are a daunting obstacle, rendering the development of parallel programs a specialty, the domain of "expert programmers."

2.2 Transactions

Like locks, Transactional Memory addresses synchronization needs, and relies on the same thread-based model of concurrent execution. However, while locks provide mutual exclusion, transactional memory provides the abstraction of atomic, isolated execution of critical sections. A programmer demarcates critical sections that may access shared data as *transactions*: all memory operations within the critical section either execute completely (commit) or have no effect (abort). The system is responsible for ensuring that transactions execute *atomically* (either completely, or not at all), and in *isolation*. A critical section is said to execute with isolation if updates made by other concurrent transactions are not visible to it, and no other transaction can see partial results from it's execution. Ultimately, transactions provide the abstraction of completely serial execution of critical sections, without requiring that serialization be the mechanism used to implement it. Locks and transactions are very different abstractions in this regard: the TM abstraction can be implemented with locks or other mechanisms. This decoupling of abstraction from implementation yields many attractive properties of TM. Most TM systems implement the abstraction *optimistically*, allowing multiple transactions to proceed concurrently, as long as atomicity and isolation are not violated. As a result, programs need not necessarily trade simplicity against scalability and coarsely synchronized programs can still achieve good performance.

Memory transactions are free of deadlock and livelock, and may be composed safely as a result: programmers can freely develop concurrent programs using thread-safe libraries based on transactions. Because the system provides atomicity, the programmer need no longer reason about partial failures. The fact that implementations rely on optimistic concurrency allows coarser-grain critical sections to scale much like fine-grain critical sections, *as long as read-write conflicts are rare.* We consider implementations that attempt to relax this caveat in chapters 4 and 5.

2.3 TM Implementation

In order to provide the abstraction of atomic isolated execution optimistically, TM systems must detect events that violate these properties, and must be able to take action that restores the system to a consistent state (one that is free of any effects of these violations). To this end, TM systems must implement some form of *version management* and *conflict detection*. TM systems require version management so that active transactions can buffer updates until commit, and so that memory can be returned to a known-consistent state in the event of conflicts or failures. These requirements fundamentally imply that the system must be able to keep track of both speculatively written versions of data and globally committed versions. Transactional memory systems must detect conflicts to ensure that executions that violate consistency can be avoided. Transactional memory systems detect conflicts by observing the sets of reads and writes made by individual transactions (called read-write or RW-sets). In general if a memory cell is present in the write set of on transaction and in the read *or* write set of another, those two transactions have a conflict, and the system has detected an execution that could lead to inconsistent results.

In the taxonomy of Moore et al. [58], both version management and conflict detection can be either eager or lazy. In eager version management, updates are made in place with the result that successful commits require minimal additional work from the system, but failed transactions have the additional overhead of requiring the system to undo those updates explicitly. Conversely, lazy version management buffers updates, and publishes them on commit, making successful transactions potentially more expensive, and failures less so. In eager conflict detection, the system checks for intersections of RW-sets on each operation, while lazy conflict detection requires the system to compare RW-sets only at commit time. Performance tradeoffs exist for conflict detection as well, and are largely workload-dependent. Chapter 4 explores dependence awareness to manage conflicts, resulting in a system that manages versions and detects conflicts in ways that do not fit the lazy/eager taxonomy outlined in [58].

TM systems must also provide (either explicitly or implicitly) some form of *contention management*. When transactions conflict, the system can either abort some number of conflicting transactions and restore to a known-consistent state, or can wait in hopes that the conflict will be resolved in a way that does not require any transactions to abort.

2.4 Hardware Transactional Memory

The transactional memory abstraction can be implemented in hardware [?,5,9,12,13, 18,22,36,39,48,58,68,70,85,94], software [2,6,15,23,31,35,72,81] or as a hybrid [7,19, 45,46,55,71,87]. Because the focus of this thesis is hardware transactional memory (HTM), the following sections explore techniques for implementation of the aspects discussed above in hardware, and detailed discussion of software techniques is elided. A thorough discussion of software techniques is available in [42].

2.4.1 Version management in hardware

Version management approaches in HTM typically rely on either modifications to caches [9,29,36,70,71] and store buffers [22,29], or logging [12,58,94].

In the cache-based approach, speculative versions of data written in transactions are effectively buffered in upper level caches, and globally committed versions are maintained in lower level caches or main memory. On a commit, speculative versions are made non-speculative with coherence or permissions state changes [9, 71,85], flash clearing of the bits that indicate a version is speculative [36], or global broadcast of written data [29]. On an abort, any speculative updates can be discarded by invalidating the corresponding cache lines. In the log-based approach, speculative updates are made in-place (in cachable virtual memory) and old versions of memory cells are written by the hardware to a thread-private undo log. On commit, logs are discarded by resetting a log pointer, and on abort, the system must walk the log to restore speculatively written data to their previous values.

Log-based and cache-based designs work with different sets of tradeoffs. The log-based approach requires (potentially) two writes for every speculative write introducing a higher fixed overhead on transactional execution, but has the benefit of avoiding artificial limits on transaction sizes that are introduced by cache geometry. A cache-based approach requires relatively less hardware and may have lower common-case latency because log operations are not required. However, if a speculative line is evicted from the cache that is buffering updates for the HTM, the system must either abort the transaction, or provide another versioning mechanism to back up these evicted data (this is the *overflow* or *virtualization* problem). Providing the abstraction of unbounded transactions in a cache-based design [5,9,18,19,29,68,91] requires either significant hardware and software complexity, or relies on inevitability mechanisms that can ultimately serialize transactions. Failure of an HTM to provide unbounded transactions admits simpler designs (called "best-effort" HTMs) but yields a programming model that requires the programmer to provide fall-back mechanisms to handle cases when the HTM exhausts hardware resources [22,39] and in-so-doing potentially compromises the accessible programming model TM claims as it's *raison d'être.*

2.4.2 Conflict detection in hardware

A transactional *conflict* occurs when the write-set of one transaction intersects with the union of the read-set and write-set of another transaction. The read(write)-set is defined as the set of addresses read(written) by a transaction. Such a conflict compromises the isolation of the transaction, so only one transaction may proceed.

This safety property is called *conflict serializability*, and it is the most efficient method for a transactional system to provide provable isolation [27].

Conflict detection in HTM is typically implemented by some variation on the theme of observing coherence traffic with interconnect-side controllers: by comparing observed reads and writes against local read-write sets, the system can accurately know when a violation of the system's safety property has occurred. Read-write sets can be represented at local nodes either as signatures [16, 55, 82, 86, 95] (hardware bloom filters [8]) or in permissions caches [10] or by presence of a cached line with transactional bits [29, 36, 58] or in a transactional coherence state [70, 71, 87]. Using signatures for conflict detection has the advantages that conflicts can be detected at arbitrary granularity (as opposed to cache-line or data-word granularity), and that the size of read-write sets in not bound to cache geometry. The most salient disadvantages of signatures stem from the fact that they can saturate and generate false positives as a result. The potential for read-write set representation to saturate complicates HTM design decisions. Signatures must be sized (in bits) to balance hardware cost against the cost in performance of falsely detected conflicts, and that balance is typically workload-dependent. Like cache-based version management, cache-based representation of read-write sets must contend with cache evictions. Without a fall-back mechanism, eviction of a transactional line causes it to disappear from the read-write set [1] Additionally, because cache coherence protocols function at cache line granularity, use of caches to represent read-write sets makes the cache line size the only natural granularity of memory blocks on which to detect conflicts. Supporting byte or word granularity requires more bits of meta-data in the cache; handling sub-line granularity conflicts in a system that fundamentally manages versions at line granularity involves non-obvious corner cases and non-trivial additional hardware.

[1] The "Sticky M@P" mechanism in LogTM [58], for example, can help address this problem for a directory-based implementation.

Conflicts can be transactional or *asymmetric*. An asymmetric conflict is one in which a non-transactional memory reference conflicts with a transaction's read-write set. Support for detection of asymmetric conflicts is known as *strong isolation* [42] or *strong atomicity* [11]. Any of the hardware-based representations of read-write sets discussed in this section naturally support detection of asymmetric conflicts, with the exception of TCC [29], which posits that all memory traffic is transactional, which eliminates the possibility of asymmetry.

2.4.3 Contention management in hardware

The hardware/software logic that determines which of a set of conflicting transactions may proceed is called the *contention manager*. Due to performance constraints, some level of contention management may happen in hardware, but most designs accommodate implementation of higher level policies in software. The losing thread(s) in a conflict will discard all of its(their) buffered changes and restart execution at the **xbegin** instruction. Approach to contention management may be complicated by *asymmetric conflicts* (see above), and *complex conflicts*, where an operation causes a conflict that involves more than two transactions (e.g. a write to a location that has been read by many readers).

When transactions conflict the HTM system must handle the conflict by ensuring that the results of the transactions involved are serializable. Traditional approaches involve either aborting some number of transactions, or stalling in hopes that the conflict will resolve (for example, by using a *stall-on-conflict* [58] policy). Chapter 4 examines other alternatives, but with the exception of DATM [71,72], TM designs rely on either aborting or stalling. When aborting some number of transactions is the strategy for handling conflicts, a *contention manager* is responsible for implementing a policy that promotes good performance. Research has clearly demonstrated the need for flexible policy in contention management [75,83], implementation in hardware requires tradeoffs. Simple policies (for example, requester-stalls [58] and requester-wins [22]) are simple to implement in hardware but can have pathologically poor performance [13]. More complex policies that rely on transaction meta data must either trap to software to run contention management handler functions [12, 48, 85, 87] or imply implementation of specialized and potentially complex algorithms directly in hardware [70]. Both approaches have drawbacks. The software approach gives rise to subtle race conditions between handler functions and concurrent transactions on remote nodes. Hardware implementation violates the time-honored principle of separating policy from mechanism, and does so at the expense of additional hardware complexity in sensitive structures such as L1 caches and coherence controllers. Chapter 5 proposes an approach to this dilemma that provides the flexibility of software preserving hardware simplicity.

2.4.4 Virtualization

Providing the abstraction of unbounded transactions is paramount to the TM vision of accessible parallel programming. Dealing with transactions that overflow the hardware state is called *virtualizing* transactions. While some designs directly accommodate large transactions [12,58,94] virtualization also also comprises the problems of handling OS events such as interrupts, context switches, and re-mapping of memory pages that may be transparent to the user programming. There are many techniques for virtualization in the recent literature, including using direct hardware support [5,68], OS page-based data structures [18,19], backup software transactional memory system [21, 41, 80], or relying on inevitability mechanisms such as allowing only one overflowed transaction [9, 29] or ordering techniques to allow concurrent software and hardware transactions [39].

Chapter 3

TxLinux and MetaTM

This chapter presents MetaTM [69, 70] and TxLinux [75, 77] a hardware software co-design for operating system support and use of hardware transactional memory.

3.1 MetaTM

MetaTM is a parameterized hardware transactional memory model capable of emulating many HTM design points, and which provides architectural features necessary to support TxLinux. The architectural interface for MetaTM is listed in Table 3.1. Starting and committing transactions rely on **xbegin** and **xend** instructions, while **xrestart** provides an explicit mechanism for retrying a transaction. The **xcas** instruction provides a compare and swap instruction that resolves contended operations using the transactional subsystem. The **xtest** instruction tests a value against an argument; if the test succeeds, the reference is included in the current transaction, and conversely, if the test fails, the system behaves as if the location were not read in the context of a transaction. The **xcas** and **xtest** instruction are critical building blocks for synchronization primitives that allow cooperation between locks and transactions (see section 3.2.1). The **xquery_tqc** instruction takes an address as input and returns true if the address was recently involved in a transactional conflict (see section 5.4 for more).

To facilitate interrupt handling in TxLinux, MetaTM supports multiple active transactions for a single thread of control [48, 69]. A thread can suspend a current transaction using **xpush** and restore it using **xpop** (see Table 3.1). The ability to save and restore transactions in LIFO order allows interrupt handlers in TxLinux to use transactions [70]. An interrupt handler executes an **xpush** to suspend any current running transaction, leaving the handler free to use transactions

Primitive	**Definition**
xbegin(*)	Begin a transaction. If the transaction does not succeed, the return code indicates the reason for the abort (e.g. conflict, I/O). Subsequent chapters explore parameterizing this instruction with meta-data (see chapter 5)
xend	Commit a transaction.
xrestart	Restart a transaction
xgettxid	Get the current transaction identifier, which is 0 if there is no currently active transaction.
xpush	Save transaction state and suspend current transaction. Used on receiving an interrupt.
xpop	Restore **xpush**ed transaction state and continue. Used on an interrupt return.
xtest(addr)	If the value of the variable equals the argument, enter the variable into the transaction read-set (if a transaction exists) and return true. Otherwise, return false.
xcas(addr, cv, nv)	A compare and swap instruction that subjects non-transactional threads to contention manager policy. If the value at the address parameter matches the compared value parameter (cv), set it to the new value (nv). Return success or failure.
xquery_tqc(addr)	An instruction that takes an address as input, and returns true if the address is on a cache line that was recently involved in a conflict.

Table 3.1: Instruction set extensions in MetaTM.

itself.

MetaTM relies on a *transaction status word* or **TXSW** to communicate information to the current thread about its transactional state [10,68]. In MetaTM, the transaction status word is returned as a result of **xbegin**, and it's value can indicate whether this is the first execution of a transaction or the transaction has restarted. If the transaction has restarted, the status word indicates the reason for the restart, such as restart due to a conflict, manual restart from the **xrestart** instruction, or restart because some form of I/O or overflow of hardware resources occurred during the transaction. Threads that execute an **xrestart** may also set user-defined codes to communicate more detailed information about the reason for the restart when the transaction resumes.

MetaTM supports only flat nesting of transactions. If software begins a new transaction when one is currently active, the new transaction is said to *nest* within the outer transaction. Flat nesting means that the inner transaction is subsumed within the outer transaction (as opposed to closed nesting, in which an abort of the inner transaction does not necessarily cause the abort of the outer, and open nesting, in which the inner transaction may commit or abort independently of the outer). Because most transactions in TxLinux are short, averaging 51 instructions (449 cycles) [70], the benefit of closed-nested transactions [62,63] is small. Moreover, in the context of an OS, most of the benefit of nesting mechanisms can be achieved without complex nesting hardware. Cxspinlocks (Section 3.2.1) and the **xpush** and **xpop** instructions provide most of the functionality, e.g. handling I/O, that is supported in other systems by open-nested transactions or other forms of suspending transactional context [15, 59, 96]. MetaTM keeps a count of the current nest depth in the transaction status word (**TXSW**).

3.1.1 Contention management, Backoff policies, and stall-on-conflict

When a conflict occurs between two transactions, one transaction must pause or restart, potentially after having already invested considerable work since starting. Contention management is intended to reduce contention in order to improve performance by reducing work lost to restarts. The MetaTM model supports the contention management strategies proposed by Scherer and Scott [83], adapted to an HTM framework, as well as additional policies *SizeMatters* [70] and *Dependence-Aware* [71]. SizeMatters favors the transaction that has the larger number of unique bytes read or written in its transaction working set, reverting to timestamp on a tie, or after a thresholded number of restarts, making it free of livelock and deadlock. Dependence-Aware contention management (see section 4) restarts the transaction whose demise minimizes the impact on the global dependence graph, likewise reverting to timestamp after a threshold of restarts to ensure livelock and deadlock freedom [67]. The policies are summarized in Table 3.2.

When a conflict occurs between transactions, and one has been selected to restart, the decision for *when* the restart occurs can impact performance. In particular, if there is a high probability that an immediate restart will simply repeat the original conflict and cause another restart, it would be prudent to wait for the other transaction to complete. In an HTM system, where detection of conflicts requires coherence traffic, repeated wasted retries can affect system-wide performance by consuming interconnect bandwidth that does not contribute ultimately to forward progress. In the absence of an explicit notification mechanism, the decision for how long to wait before retrying is heuristic. The MetaTM model supports using different backoff strategies, enabling the tailoring of policy to different workloads. Previous work has focused on exponential backoff strategies [83, 84]. The following list summarizes the backoff policies supported by MetaTM. Chapter 5 proposes mechanisms that ameliorate the need for heuristic approaches to the timing of transaction retries.

Policy	**Definition**
Polite	Backoff up to an empirical threshold, 10 in our case. See section 3.1.1.
Karma	Abort transaction that has done the least work. Work is estimated with the number of operations to unique addresses within a transactional context. Karma updates a priority counter for each transactional reference, and does not reset the counter on restarts.
Eruption	Karma variant, with priority boosting. Conflict winner's priority is added to the loser, who has a higher priority for future conflicts.
Kindergarten	Transactions are willing to defer to each other once, but no more. If no transactions in a conflict are willing to defer, resorts to the timestamp policy.
Timestamp	Oldest transaction wins. Timestamp is not refreshed on restart [67].
Polka	Polite backoff strategy combined with Karma priority accumulation. The number of references to the transaction working set approximates priority, which is the same as the Karma policy. The backoff strategy does not have to be exponential, and the backoff seed (normally random) is the delta between the approximated priorities. With eager conflict detection, at least one of the operations involved in a conflict arbitration must be a write; consequently the policy defaults to a "writes-always-win" policy, unless both conflicting operations are writes.
SizeMatters	Largest transaction size (unique bytes read or written) wins. Size is reset on restart. After an empirical threshold number of restarts, it reverts to timestamp, to avoid livelock.
Dependence Aware	Transaction whose restart will minimize impact on the dependence graph of active transactions is selected (see chapter 4).

Table 3.2: Contention management policies implemented in MetaTM. Because hardware transactions do not block in our model (they can execute, restart, or stall, but cannot wait on a queue), certain features require adaptation.

- **Exponential – Choose a random wait time from a range that expands exponentially upon each subsequent retry.**
- **Linear** – Linear Backoff is implemented by choosing a random seed between 1 and 10. The seed is multiplied by the number of times the conflicting transaction has backed off to determine the number of cycles that the conflicting transaction should wait before a restart.
- **Random** – Random backoff is implemented by choosing a number of cycles at random to wait before restarting. The maximum value is 1000.

- **None** – Retry as soon as possible.

An HTM system need not necessarily respond to detected conflicts by aborting: the system can attempt to stall the requesting transaction until the conflict resolves (called *stall-on-conflict*), or can attempt to choose and enforce a *post-facto* commit order for the conflicting transactions that preserves correctness in the presence of the conflict (called *dependence-awareness* [71]). Both mechanisms introduce the need for some deadlock detection or avoidance mechanisms [58,72], and both can reduce the rate of transaction restarts for workloads with contention. The degree to which reducing restarts translates to improved performance is workload-dependent. MetaTM supports both stall-on-conflict, and dependence awareness.

3.2 TxLinux

TxLinux is a transactional variant of Linux that uses transactional memory to replace various forms of polling synchronization such as spinlocks, seqlocks, and read-copy-update (RCU). Initial work with MetaTM and TxLinux involved a profile-guided conversion of the most highly contended locks in some subsystems to use transactions. The complete list of subsystems converted in the initial effort is shown in Table 3.4. The effort was highly labor intensive, primarily because Linux performs I/O with spinlocks held. I/O cannot be performed in transactions (due to the output commit problem [25]), and the nesting relationships among locks cannot always be determined statically, so while the first version of TxLinux supported transactions in 9 subsystems, it was clear that the approach of rote conversion of locks to bare transactions in an operating systems was flawed. Transactions cannot eliminate locks in an OS: some cooperation between these synchronization mechanisms is a fundamental requirement.

3.2.1 Cooperation between locks and transactions

In order to allow both transactions and conventional locks in the operating system, TxLinux supports a synchronization API that affords their integration: *cxspinlocks*, or cooperative transactional spinlocks. Cxspinlocks allow different executions of a single critical section to be synchronized with either locks or transactions. This freedom enables the increased concurrency enabled by the optimism of transactions when possible, but enforces the safety of locks when necessary. Locking may be used for I/O, for protection of data structures read by hardware (e.g., page tables), or for high-contention access paths to particular data structures (where the performance of transactions might suffer from excessive restarts). The cxspinlock API also provides a simple upgrade path to let the kernel use transactions in place of existing synchronization. Because cxspinlocks transition transparently between transactions and mutual exclusion according to the dynamic needs of the system, programmers need not invest the painstaking effort in differentiating critical sections that may perform I/O that was required in the initial conversion of TxLinux. The second version of TxLinux, using cxspinlocks, required less than 1/10th of the development effort than the original.

Cxspinlocks rely on the ability to retry transactions that perform I/O dynamically, using mutual exclusion rather than speculation on the retry attempt, some need for lock variables is re-introduced. However, cxspinlocks are necessary for the kernel only and they allow the user programming model to remain simple. Users do not need them because they cannot directly access I/O devices (in Linux and most operating systems, users perform I/O by calling the OS). Blocking direct user access to devices is a common OS design decision that allows the OS to safely multiplex devices among non-cooperative user programs. Sophisticated user programs that may have some need for coexistence between transactions and locks *can* use cxspinlocks, but it is not required.

Using conventional Linux spinlocks within transactions is possible and will maintain mutual exclusion. However, conventional spinlocks reduce the concurrency of transactions and lacks fairness. Conventional spinlocks prevent multiple transactional threads from executing a critical region concurrently. All transactional threads in a critical region must read the spinlock memory location to obtain the lock and must write it to obtain the lock and release it. Write sharing of the lock variable among transactional and non-transactional threads will cause transactional and/or asymmetric conflicts, thereby preventing concurrent execution, even if concurrent execution of the "real work" in the critical section is safe (due to asymmetric conflict [70], or strong isolation [42]). Moreover, conventional spinlocks do not help with the I/O problem. A transactional thread that acquires a spinlock can restart, therefore it cannot perform I/O.

The progress of transactional threads can be unfairly throttled by non-transactional threads using spinlocks. In MetaTM conflicts between transactional and non-transactional threads (asymmetric conflicts) are always resolved in favor of the non-transactional thread. To provide isolation, HTM systems guarantee either that non-transactional threads always win asymmetric conflicts (like MetaTM), or transactional threads always win asymmetric conflicts (like LogTM [58]). With either convention, traditional spinlocks will cause unfairness between transactional and non-transactional threads.

3.2.2 Using cxspinlocks in TxLinux

TxLinux replaces all spinlocks with cxspinlocks. Cxspinlocks allow a single critical region to be safely protected by either a lock or a transaction. A non-transactional thread can perform I/O inside a protected critical section without concern for undoing operations on a restart. Many transactional threads can simultaneously enter critical sections protecting the same shared data, improving performance. Simple

cx_optimistic

```
void
cx_optimistic(lock){
 status = xbegin;
 if(status==NEED_EXCL){
  xend;
   if(xgettxid)
    xrestart(NEED_EXCL);
   else
    cx_exclusive(lock);
   return; }
 while(!xtest(lock,1));
}
```

cx_exclusive

```
void
cx_exclusive(lock){
 while(1) {
  while(*lock != 1);
  if(xcas(lock, 1, 0))
    break;
 }
}
```

cx_end

```
void
cx_end(lock){
 if(xgettxid)
  xend;
 else
  *lock = 1;
}
```

Table 3.3: The cxspinlock API and implementation. The `cx_optimistic` function attempts to execute a critical section by starting a transaction, using **xtest** to spin until the lock is free. If the critical section attempts I/O, the hardware will retry the transaction, returning the `NEED_EXCL` flag from the **xbegin** instruction. This will result in a call to the `cx_exclusive` function, which waits until the lock is free, and acquires the lock using the **xcas** instruction to atomically compare and swap the lock variable, and which invokes the contention manager to arbitrate any conflicts on the lock. The `cx_end` function exits a critical section, either by ending the current transaction, or releasing the lock.

return codes in MetaTM allow the choice between locks and transactions to be made dynamically, simplifying programmer reasoning. Cxspinlocks ensure a set of behaviors that allow both transactional and non-transactional code to correctly use the same critical section while maintaining fairness and high concurrency:

- Because cxspinlocks test lock variables with the **xtest** instruction, multiple transactional threads may enter a single critical section without conflicting on the lock variable. A non-transactional thread will exclude both transactional and other non-transactional threads from entering the critical section because it will update the variable to indicate that the lock is held.
- Transactional threads poll the cxspinlock using the **xtest** instruction, which allows a thread to check the value of a lock variable without entering the lock variable into the transaction's read set. This enables the transaction to avoid restarting when the lock is released (another thread writes the lock variable). This is especially important for acquiring nested cxspinlocks where the thread

```
void
dnotify_parent(
    dentry_t *dentry,
    ulong evt) {
 dentry_t* p;
 spin_lock(&dentry->d_lock);
 p = dentry->d_parent;
 dget(p);
 spin_unlock(&dentry->d_lock);
 inode_dir_notify(p->d_inode,
    evt);
 spin_lock(&dcache_lock);
 if(!(--p->d_count)) {
  spin_lock(&p->d_lock);
  dentry_iput(p);
  d_free(p);
  spin_unlock(&p->d_lock);
 }
 spin_unlock(&dcache_lock);
}
```

```
void
dnotify_parent(
    dentry_t *dentry,
    ulong evt) {
 dentry_t* p;
 xbegin;
 p = dentry->d_parent;
 dget(p);
 inode_dir_notify(
    p->d_inode,
    evt);
 if(!(--p->d_count)){
   dentry_iput(p);
   d_free(p);
 }
 xend;
}
```

```
void
dnotify_parent(dentry_t *dentry,
               ulong evt) {
 dentry_t* p;
 cx_optimistic(&dentry->d_lock);
 p = dentry->d_parent;
 dget(p);
 cx_end(&dentry->d_lock);
 inode_dir_notify(p->d_inode,
                  evt);
 cx_optimistic(&dcache_lock);
 if(!(--p->d_count)){
   cx_optimistic(&p->d_lock);
   dentry_iput(p);
   d_free(p);
   cx_end(&p->d_lock);
 }
 cx_end(&dcache_lock);
}
```

Figure 3.1: Three adapted versions of the Linux file system `dparent_notify()` function, which handles update of a parent directory when a file is accessed, updated, or deleted. The leftmost version uses locks, the middle version uses bare transactions and corresponds to the code in TxLinux-SS, and the rightmost version uses cxspinlocks, corresponding to TxLixux-CX. Note that the `dentry_iput` function can do I/O.

will have done transactional work before the attempted acquire.

- Non-transactional threads acquire the cxspinlock using an instruction (**xcas**). Transactional conflicts involving **xcas** are arbitrated by the transactional contention manager. This preserves fairness between locks and transactions because the contention manager can implement many kinds of policies favoring transactional threads, non-transactional threads, readers, writers, etc. Non-transactional threads attempting to enter a critical section (with cx_exclusive) are subject to the same policy decisions as transactional ones, giving the contention manager the power to force a non-transactional thread to wait for a transactional one.

Figure 3.3 shows the cxspinlock API and implementation. Cxspinlocks are acquired using two functions: `cx_exclusive` and `cx_optimistic`. Both functions take a lock address as an argument. `cx_optimistic` is a drop-in replacement for spinlocks and is safe for almost all locking done in the Linux kernel (the exceptions are a few low-level page table locks and locks whose ownership is passed between threads, such as that protecting the run queue). `cx_optimistic` optimistically attempts to protect a critical section using transactions. If a code path within the critical section protected by `cx_optimistic` requires mutual exclusion, then the transaction restarts and acquires the lock exclusively. The code in figure 3.1, which can fail due to I/O with bare transactions, functions with cxspinlocks, taking advantage of optimism with transactions when the `dentry_iput` function does no I/O, and retrying with with exclusive access when it does.

Control paths that will always require mutual exclusion (e.g., those that always perform I/O) can be optimized with `cx_exclusive`. Other paths that access the same data structure may execute transactionally using `cx_optimistic`. Allowing different critical regions to synchronize with a mix of `cx_optimistic` and `cx_exclusive` assures the maximum concurrency while maintaining safety.

TxLinux Scheduling

MetaTM allows the OS to communicate its scheduling priorities to the hardware conflict manger, so the hardware does not subvert OS scheduling priorities or policy. Locks can invert OS scheduling priority, resulting in a higher-priority thread waiting for a lower-priority thread. Some OSes, (e.g. Solaris [49, 78]), support priority inheritance to address this issue. Priority inheritance guarantees an upper bound on the impact of inversion, but is complicated, and cannot completely eliminate the problem. In contrast, the contention manager of an HTM system can nearly eradicate priority inversion. If the contention manager resolves conflicts in favor of the thread with higher OS scheduling priority, then transactions will not experience priority inversion.

To eliminate priority and policy inversion, MetaTM provides an interface for the OS to communicate scheduling priority and policy to the hardware contention manager [58,69] MetaTM implements a contention management policy called *os_prio*, which prefers the transaction with the greatest scheduling value to the OS, defaulting to other policies when a tie occurs in the priority value. The policy is able to eliminate virtually all inversion in TxLinux.

The presence of HTM provides opportunity for the OS scheduler to use process transaction state to mitigate the effects of high contention. TxLinux supports a modified scheduler that takes into account the existence of any currently active transactions, number of recent restarts, cycles spent backing off, and the size of the transaction read and write set, when making scheduling decisions, relying on the MetaTM transaction status word [68] to determine the status of the current transaction (none, active, stalled, overflowed). Using this information, the scheduler dynamically adjusts priority or deschedules processes likely to cause repeated restarts, improving throughput for workloads with high contention.

3.3 Evaluation

This section presents detailed measurements of TxLinux. The experiments show that the performance of transactions is generally good for 16 and 32 CPUs (in an SMP organization), though a performance pathology is introduced in one case. Even at 32 cores, the kernel spends less than 12% of its time synchronizing, so the opportunity to improve performance with synchronization primitives is limited at this scale. Using cxspinlocks to add transactions to the kernel removes the primary reasons to eschew transactions in the kernel—the engineering effort to add them and their incompatibility with I/O.

Priority inversion is a common occurrence in the Linux kernel the selected benchmarks, and TxLinux's ability to nearly eliminate it is an encouraging result for transactional programming. Allowing the scheduler to use transaction state information has little ability to affect performance for the workloads studied, although scheduler effort can be profitably directed toward avoiding transactional performance pathologies.

3.3.1 Experimental setup

TxLinux is based on Linux 2.6.16, and MetaTM is implemented as a hardware module in the Simics [47] 3.0.31 machine simulator. The architecture is x86, with 16 and 32 processors. The model assumes 1 instruction per cycle and in-order cores. Simics only allows a constant IPC, and 1 is a reasonable choice for a moderate superscalar implementation. Level 1 caches are both 16 KB with 4-way associativity, 64-byte cache lines, 1-cycle cache hit and a 16-cycle cache miss penalty. The L1 data caches contain both transactional and non-transactional data. Second level caches are 4 MB, 8-way associative, with 64-byte cache lines and a 200 cycle miss penalty to main memory. Cache coherence is maintained with a MESI snooping protocol, and the main memory is a single shared 1GB. This configuration is typical for an SMP, and reasonably approximates a CMP.

The disk device models PCI bandwidth limitations, DMA data transfer, and has a fixed 5.5ms access latency. Simics models the timing for a tigon3 gigabit network interface card that supports DMA data transfer, with an Ethernet link that has a fixed 0.1ms latency. All of the runs are scripted, with no user interaction.

MetaTM uses word-granularity conflict detection, exponential backoff on conflict, and the *SizeMatters* contention management policy [70]. Simics uses execution-based simulation, which allows the choices made by the OS and hardware (e.g., scheduling decisions and contention management) to feed back into the simulation and change thread orderings and application behaviors. This provides more realistic modeling.

Multi-threaded workloads tend to have variable performance, in the sense that a small change to the thread schedule can introduce noticeable jitter into execution time. To compensate for this variability, cache miss timings are pseudorandomly perturbed to allow data to be sampled from the space of reasonable thread interleavings using the statistical approach of Alameldeen and Wood [4] to produce confidence intervals from the perturbed runs.

The TxLinux workloads are described in Table 4.7, and figure 3.2 characterizes them in terms of user, system, I/O wait, and idle time. The benchmarks are large applications that exercise the kernel in realistic scenarios. Some of them fix the amount of work, usually at 32 threads, and some scale the amount of work with the processor count. The benchmarks do not execute any transactions at user-level: all transactions occur in the kernel. Since the kernel is using HTM, experiments measure the behavior of the *kernel* being exercised by the workloads. The benchmark `bonnie++` is run with a zero latency disk because its performance with disk latency is highly dependent on block layout. Removing the disk delay allows our analysis to focus on the CPU portion of the workload, independent from the file system layout.

TxLinux was converted to use transactions using two methods. The first conversion (called the subsystem (SS) kernel, or TxLinux-SS) was done by hand, converting the spinlocks in subsystems shown in Table 3.4 to use transactions. The second (called the cxspinlocks (CX) kernel, or TxLixux-CX) converted nearly all spinlocks to use `cx_optimistic`. For both conversions, all sequence locks are converted to use transactions, and some reader/writer spinlocks are converted.

3.3.2 Synchronization performance

The time wasted due to synchronization as a percentage of kernel execution time was measured for Linux and TxLinux-SS. In Linux, synchronization time is wasted spinning on locks. In TxLinux time is wasted spinning on locks as well as restarting transactions (including time spent backing off before restart). Figure 3.3 shows that both Linux and TxLinux spend from 1–14% of their execution time synchro-

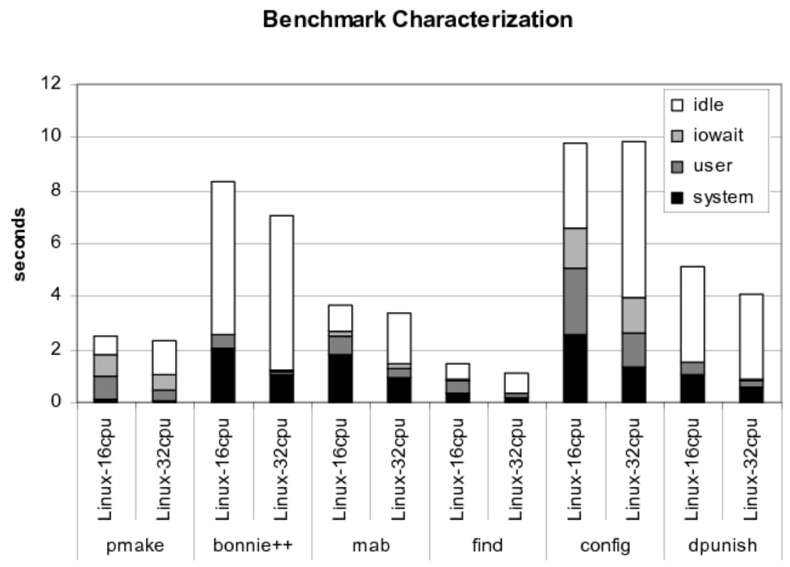


Figure 3.2: User, system, I/O wait, and idle time for all benchmarks for 16 and 32 CPUs, characterized using unmodified Linux.

nizing. For a 16 CPU configuration, TxLinux-SS wastes an average of 57% less time synchronizing than Linux does, and for 32 CPUs it wastes 1% more. Most of this time savings is attributable to removing the cache misses for the lock variable itself. These experiments did not measure time spent spinning on seqlocks, which biases the results in favor of Linux.

The data shows that as the number of CPUs increase, time wasted synchronizing also increases. While HTM generally reduces the time wasted to synchronization, it more than doubles the time lost for *bonnie++*. This loss of performance is due primarily (90%) to transactions that restart, back-off, but continue failing. Since *bonnie++* does substantial creation and deletion of small files in a single directory, the resulting contention in file system code paths results in pathological restart behavior in the function `dput`, which decrements the link count of the directory and manipulates a few lists in which the directory entry appears. The fast-changing

slab allocator	Kernel memory allocator with extensive use of fine-grained locking.
dentry cache	Locks protecting the directory entry cache, accessed on pathname lookup and file create/delete.
RCU	Transactions used in place of spinlocks in the Read-Copy-Update implementation
struct address_space	Protects private, shared, and nonlinear mappings within an address space (i_mmap_lock).
zoned page frame allocator	Physical memory zone descriptor and active/ inactive lists synchronized with transactions. Includes ZONE_HIGHMEM locks.
timekeeping architecture	Sequence lock protecting the xtime variable
memory	Lock protecting a list that contains all process descriptors list process memory descriptors (mmlist_lock)
VFS file objects	Protects accesses to lists of open files. (files_lock)
noncontiguous memory areas	protects a doubly linked list of physically non-contiguous memory areas. (vmlist_lock)

Table 3.4: Subsystems from the Linux 2.6.16 kernel altered to use transactions instead of locks (TxLinux-SS). Subsystem names correspond directly to index entries in Bovet and Cesati [14].

link-count effectively starves a few transactions. Using back-off before restart as a technique to handle such high contention may be insufficient for complex systems: the transaction system may need to queue transactions that consistently do not complete. The remaining 10% of the performance loss is attributable to large transactions, which cause overflow of the transactional memory state from the L1 cache and incur virtualization costs for conflict detection and version management of the overflowed data. There are many proposals to virtualize transactions that grow too large for hardware resources, and this data indicates the importance of such schemes. However, both of these issues in *bonnie++* could be addressed in TxLinux by using `cx_exclusive` to protect the critical region in `dput` that creates the transaction that has difficulty completing.

The Simics hardware module implementing the transactional subsystem is

bonnie++	Simulates file system bottleneck activity on Squid and INN servers stressing create/stat/unlink. 32 instances of: `bonnie++ -d /var/local -n 1` Run with 0ms of disk delay.
configure	Run several parallel instances of the configure script for a large software package, one for each processor.
find	Run 32 instances of the `find` command, each in a different directory, searching files from the Linux 2.6.16 kernel for a text string that is not found. Each directory is 4.6–5.0MB and contains 333–751 files and 144–254 directories.
MAB	File system benchmark simulating a software development workload. [65] Runs one instance per processor of the Modified Andrew Benchmark, without the compile phase.
pmake	Runs make -j 2 * number_of_procs to compile 27 source files totaling 6,031 lines of code from the libFLAC 1.1.2 source tree in parallel.
dpunish	A locally developed micro-benchmark to stress synchronization in VFS directory entry cache. Parallel lookups and renames across multiple, memory-based file systems.

Table 3.5: Benchmarks used to evaluate TxLinux.

also used to measure the number of times a spinlock was acquired, the number of cycles spent acquiring it, and the number of times a process had to spin before acquiring a lock. Spinlocks are "test and test&set" locks, so iterations of the inner (test) and outer (test&set) loops are counted separately.

Table 3.6 presents details on the locking behavior of Linux and TxLinux, showing that TxLinux reduces lock contention more than it eliminates calls to locking routines. It eliminates 37% of calls to lock routines, 34% of the test loops and 50% of the test&set loops. Reducing the number of test&set operations is important because these operations use the coherence hardware, reducing system throughput. TxLinux lowers lock contention by converting some heavily contended locks to use cxspinlocks that allow multiple transactional threads into a critical region concurrently. Another interesting trend in Linux is that from 16 to 32 CPUs the number of

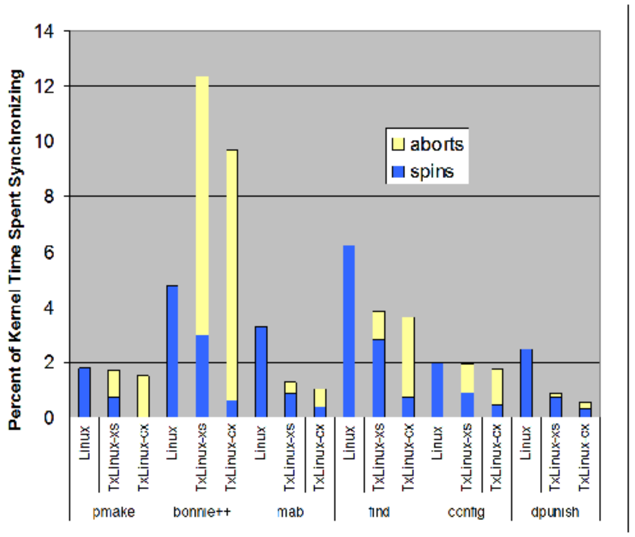


Figure 3.3: Time lost due to restarted transactions and acquiring spin locks in 16 and 32 CPU experiments. For each benchmark, the first bar represents Linux and the second represents the subsystem kernel TxLinux-SS. Time for TxLinux-SS is broken down into spinlock acquires and restarted transactions, whereas synchronization time for Linux is only for spinlock acquires.

lock acquires does not increase substantially, but the amount of spinning increases about 3×. This suggests that while the amount of time spent in synchronization for 32 CPU configurations is tolerable, lock-based synchronization overhead will be an impediment to large system scalability.

3.3.3 Concurrency in TxLinux

In order to measure the degree of concurrency provided by transactions compared to locking, each transactional thread upon entering a critical section records the number of other transactional threads in that critical section. Figure 3.4 shows a histogram of the maximum concurrency for the critical sections used in many of the benchmarks on 32 CPUs with the cxspinlock kernel. 67% of the 284 critical regions have more than a single thread executing at once, indicating that even Linux's highly tuned critical regions can benefit from being executed in parallel. The critical region

		Linux			TxLinux		
		Acq	TS	T	Acq	TS	T
bonnie++	16	12,478	132	340,523	28%	20%	68%
config	16	16,087	62	49,432	31%	56%	33%
dpunish	16	9,626	35	18,406	51%	66%	32%
	32	10,514	102	153,699	49%	39%	6%
find	16	2,912	72	34,553	39%	42%	14%
	32	2,758	183	111,629	40%	52%	21%
mab	16	15,451	101	45,167	51%	81%	55%
	32	15,871	146	96,370	50%	71%	39%
pmake	16	764	9	8,981	30%	38%	24%
	32	1,004	24	35,341	25%	48%	18%

Table 3.6: Spinlock performance for unmodified Linux vs. the subsystem kernel TxLinux-SS. Acq represents the number of times the spinlock (a test and test&set lock) is acquired. T (test) represents the number of times a processor spins on a cached lock value, while TS (test&set) represents the outer loop where the lock code performs a cache coherent locked decrement. Linux measurements are in the thousands. TxLinux-SS measurements are the percent reduction from Linux. For example, for 16 CPU pmake,Linux performs 9,000 locked decrements in the outer loops of spinlock acquisition, while TxLinux-SS performs about 5,500 resulting in a 38% reduction. 32 CPU data for *bonnie++* and *config* were not available.

that has 32 threads in it at once is the sequence lock that reads the kernel tick counter in the frequently executed function `do_gettimeofday`. In Linux, this critical region is guarded by a sequence lock, so it may also contain many concurrent threads. In TxLinux, however, it is not necessary to reason about the type of accesses to protected data, while seqlocks require a programmer to distinguish between read-locking and write-locking. With transactions and cxspinlocks, a single primitive adds concurrency for critical regions with many readers.

Linux 2.6.16 is optimized to minimize lock contention. Moreover, most transactions executed by TxLinux are critical sections converted from locks. As a result, the average concurrency in critical regions is low, and the amount of time spent in critical regions is small compared to the total kernel execution time. If average transaction sizes grow to reflect TM's ability to achieve high concurrency with coarser-grained critical sections, the average and maximum concurrency will increase.

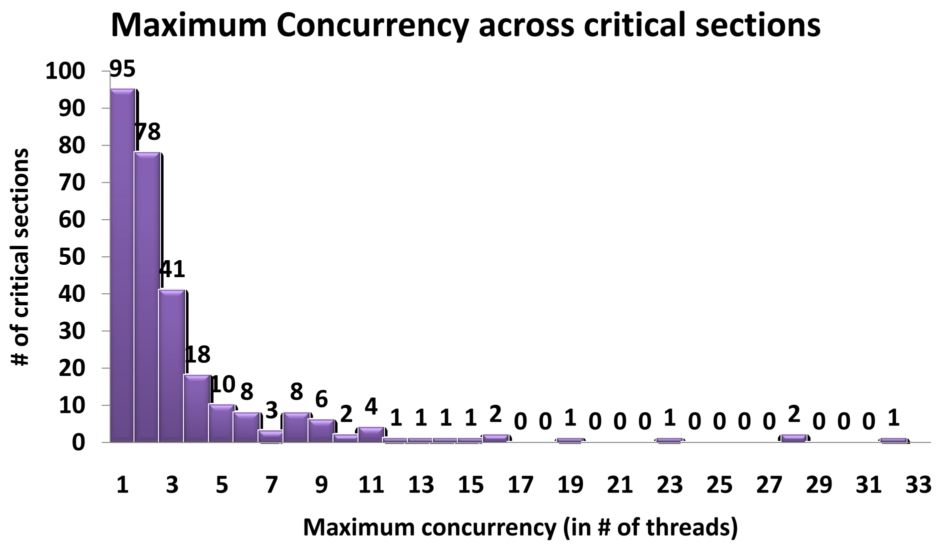


Figure 3.4: Distribution of maximum concurrency across TxLixux-CX critical sections for the config, find, mab and pmake benchmarks on 32 processors.

3.3.4 Cxspinlock performance and use

One of the main advantages of traditional spinlocks is their low overhead for locking and unlocking. When acquiring an uncontended lock, the body of the `spin_lock` function executes only 3 instructions, including 2 memory references. When acquiring a spinlock that is already locked, only 9 instructions are executed in addition to the time spent waiting. Unlocking a spinlock is usually inlined, requiring just one instruction.

Acquiring a cxspinlock involves more complicated logic than a normal spinlock, introducing some overhead in the number of instructions executed. Calling `cx_optimistic` to begin a transaction for an uncontended critical section requires 21 instructions and 9 memory references. Using `cx_exclusive` to enter an uncontended critical section from a non-transactional thread requires 21 instructions and

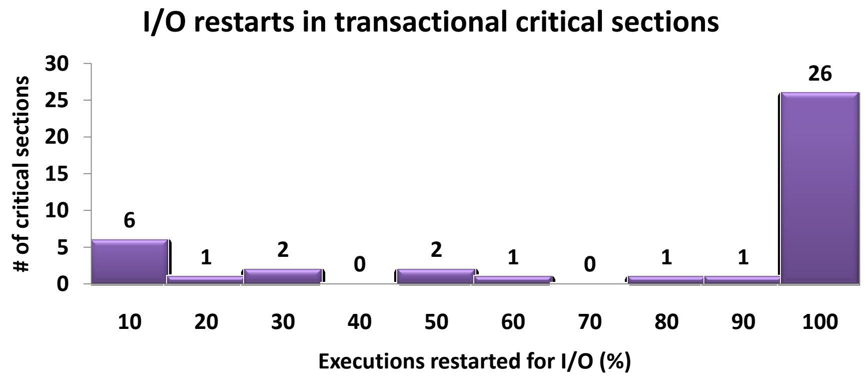


Figure 3.5: Distribution across TxLixux-CX critical sections of the percentage of executions that require restarts for I/O, measured with the config, find, mab and pmake benchmarks with 16 and 32 processors.

8 memory references. In both cases, all references except one are to stack variables. The x86 optimizes accesses to stack variables, and stack addresses are highly likely to be cache-resident, so these references contribute minimal additional latency.

In practice, the performance of cxspinlocks is very near that of traditional spinlocks. Averaging across all benchmarks, the introduction of cxspinlocks results in kernel time slowdowns of 3.1% and 2.8% for 16 and 32 CPUs respectively. By contrast, the subsystem conversion of Linux to TxLinux does not use cxspinlocks: for 16 CPUs, the subsystem kernel has a 2.0% slowdown on average (excluding *bonnie++*, whose pathologies were discussing in section 3.3.2 this becomes a 0.9% speedup), and on 32 CPUs it garners a 2.0% speedup. In all cases the change in performance is within the confidence interval of the measurement.

To justify the increased complexity of cxspinlocks, there must exist critical regions in the Linux kernel that require exclusion along some but not all code paths. Figure 3.5 shows how often I/O is performed in critical regions protected by `cx_optimistic`, restricted to those critical regions that contain I/O along at least one code path. Several critical regions perform I/O along a small percentage of dynamic code paths, and so may benefit from `cx_optimistic`. The majority, however, perform I/O all or nearly all of the time. These critical regions should be optimized by replacing `cx_optimistic` with `cx_exclusive`. Even in these cases cxspinlocks enable additional concurrency, as there are locks shared between critical regions that always perform I/O and critical regions that never perform I/O (e.g. the coarse lock protecting the ide subsystem is sometimes used to protect device access and is sometimes used to protect simple data structures). Critical regions that do not perform I/O may execute concurrently, even when they share data with critical regions that will always require mutual exclusion.

Table 3.7 shows the amount of time wasted when restarting transactions for I/O. In the current implementation of cxspinlocks, an I/O operation can cause a number of transaction restarts equal to the nesting depth when the I/O operation was executed. However, the average nesting depth when executing I/O (shown in the Table) operations is low, with no I/O nested at more than 3 levels. The config and MAB workloads perform a lot of I/O, and hence lose the most time to I/O restarts. The time wasted restarting for I/O in TxLinux is mostly time spent idle in Linux, because the I/O restart happens right before suspending the last runnable process (all other processes are blocked on I/O). The runtime of Linux and TxLinux on these workloads is nearly identical.

3.3.5 Contention management using OS priority

Figure 3.6 shows how frequently transactional priority inversion occurs in TxLinux. In this case, priority inversion means that the default *SizeMatters* contention management policy [70] favors the process with the lower OS scheduling priority (results for timestamp are similar). Most benchmarks show that a significant percentage of transactional conflicts result in a priority inversion, with the average 9.5% across all

		I/O		Origin (SS)		Origin (CX)	
		Nest	Waste	sys	intr	sys	intr
config	16	1.42	32.3%	46.3%	53.7%	49.6%	50.4%
	32	1.36	36.3%	45.9%	54.0%	49.8%	50.2%
find	16	1.51	0.3%	74.8%	25.2%	68.6%	31.4%
	32	1.39	2.8%	79.5%	20.5%	67.8%	32.2%
mab	16	1.36	13.7%	73.4%	26.6%	63.6%	36.4%
	32	1.30	31.2%	73.2%	26.8%	63.8%	36.2%
pmake	16	1.51	0.3%	51.5%	48.4%	21.3%	78.7%
	32	1.50	0.3%	48.6%	51.2%	15.1%	84.9%

Table 3.7: Cxspinlock usage in TxLinux. Nest is the average nesting depth when I/O operations are executed in transactions. Waste is the total time wasted due to restarting for I/O as a percentage of kernel execution time. Sys/intr shows the percentage of all transactions that originated in system calls and interrupts, respectively. Data is given for both the subsystem and cxspinlocks kernel.

kernel and CPU configurations we tested. While priority inversion tends to decrease with larger numbers of processors, the trend is not strict. The `pmake` and `bonnie++` benchmarks show an increase with higher processor count for the TxLinux-default (the unmodified Linux scheduler) and TxLinux-sched (the transaction-aware modified scheduler) kernels respectively. The number and distribution of transactional conflicts is chaotic, so changing the number of processors can change the conflict behavior. Policy inversion, where a non-real-time thread can be favored in a conflict

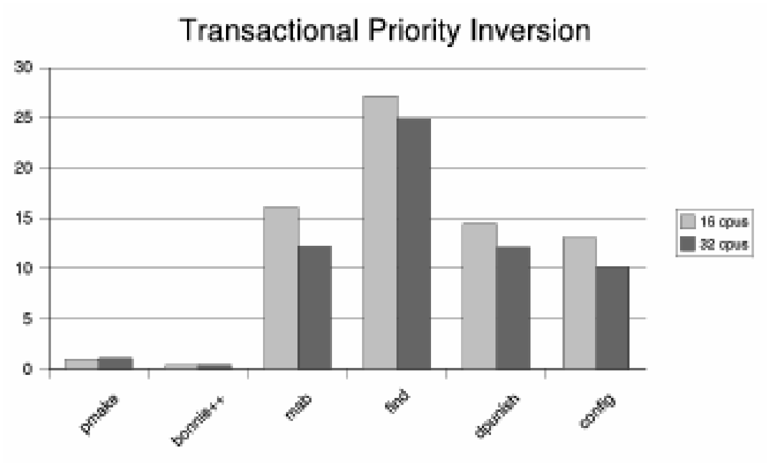


Figure 3.6: Percentage of transaction restarts decided in favor of a transaction started by the processor with lower process priority, resulting in "transactional" priority inversion. Results shown are for all benchmarks, for 16 and 32 processors, TxLinux-SS .

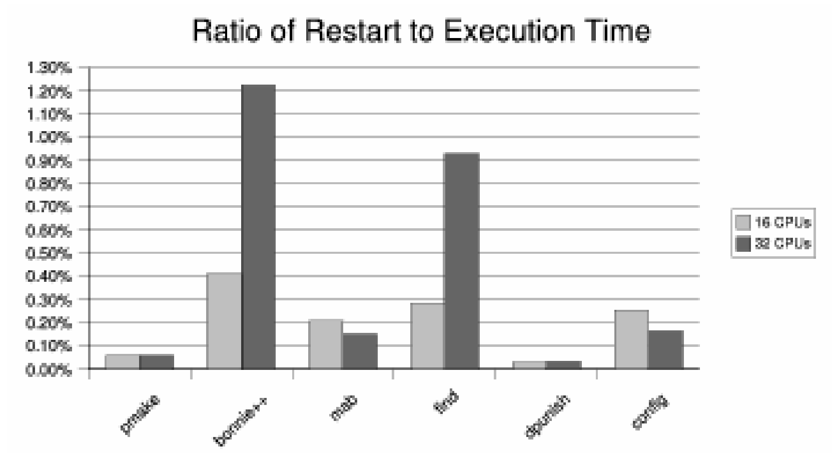


Figure 3.7: Restart cycles as a percentage of total execution time for TxLinux-default (SS) with 16 and 32 cpus. The percentage of restart cycles gives a theoretical upper bound on the performance benefit achievable by a scheduling policy that attempts to minimize restart waste.

over a real-time thread, is much rarer: it occurs only in `mab` and `dpunish` benchmarks at rates of 0.01% and 0.02% respectively. The os_prio contention management policy eliminates both priority inversion and policy inversion entirely in our benchmarks, at a cost in performance that is under 2.5% for TxLinux-default and under 1% for TxLinux-sched.

The frequency with which naïve contention management violates OS scheduling priority argues strongly for a mechanism that lets the OS participate in contention management, e.g., by communicating hints to the hardware.

3.3.6 Transaction-aware scheduling

The goal of transaction-aware scheduling (TxLinux-sched) is to take advantage of the availability of transaction state information from the hardware to increase performance, primarily by making scheduling decisions that attempt to decrease lost work due to restarts. Figure 3.7 shows cycles spent restarting contending transactions as a percentage of total execution time for all benchmarks using TxLinux-default (unmodified scheduler) and TxLinux-sched kernel configurations. For most benchmarks, the opportunity to improve performance by eliminating restarts is lim-

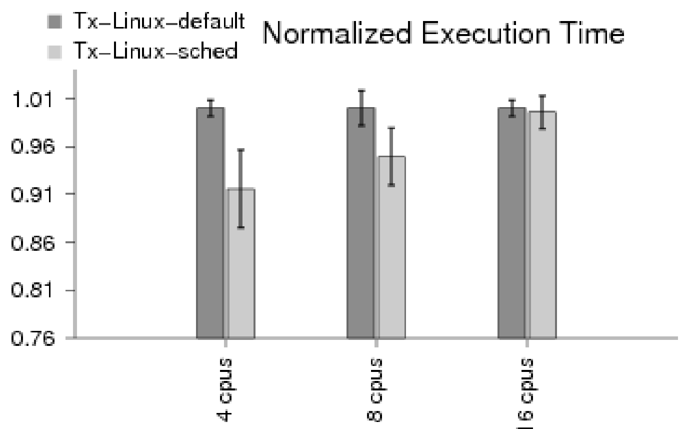


Figure 3.8: Relative execution time for the pipeline micro-benchmark for TxLinux-sched , TxLinux-default with 4, 8, and 16 cpus.

ited: on average, if savvy scheduling were to eliminate all wasted restart cycles, the overall performance gain for 16 and 32 cpus would be <1% (averaged across all benchmarks), a statistically insignificant margin, given the confidence intervals shown in simulation. Empirically, TxLinux-sched execution time is within 1.5% of TxLinux-default for all benchmarks, providing neither a consistent benefit, nor a consistent detriment to performance.

The TxLinux-sched policy attempts to deschedule threads that are under significant contention, as indicated by the restart and backoff profile for the thread. As a result, the ability of the policy to have a significant positive effect relies heavily on both the presence of significant contention and the availability of threads at a similar priority that are able to make progress when scheduled in place of descheduled threads. While a scheduling policy that reduces restarts may have minimal impact where contention is low on average, as it is in our benchmarks, it can have a more significant impact in situations where contention is high, reacting to contention to ameliorate extreme conditions in ways that are not possible with traditional locks.

A micro-benchmark called `pipeline` was used to test this hypothesis. The pipeline benchmark, simulates a multi-threaded application that has significantly longer transactions and higher contention than the critical regions in TxLinux. It consists of multiple threads (4× the number of processors) each working through a set of 8 phases: the memory references made by the threads are mostly distinct to the phase. If all threads are working in the same phase, contention is very high, and it is unlikely that more than one thread at a time can make progress, while execution can generally be overlapped safely for threads in different phases. Figure 3.8 shows normalized execution time for this micro-benchmark, for the TxLinux-default and TxLinux-sched configurations. The TxLinux-sched scheduler is able to improve performance by 8% and 6% for 4 and 8 cpus respectively, while the benefit under 16 cpus is too close to the confidence intervals to be significant. The total number of restarts and total restart cycles wasted are reduced by 20.3% and 21.5% respectively on average, showing that transaction aware scheduling can potentially help manage contention related pathologies, while having no negative performance impact under low contention.

3.4 Conclusion

TxLinux is the first operating system to use HTM as a synchronization primitive, and represents innovation for HTM-aware scheduling and cooperation between locks and transactions. TxLinux demonstrates that HTM can provide comparable performance to locks, and can simplify code while coexisting with other synchronization primitives in a modern OS. The cxspinlock primitive enables a solution to the long-standing problem of I/O in transactions, and the API eases conversion from locking primitives to transactions significantly. Introduction of transactions as a synchronization primitive in the OS reduces time wasted synchronizing on average, but can cause pathologies that do not occur with traditional locks under very high contention or when critical sections are sufficiently large for the overhead of HTM virtualization to become significant. HTM aware scheduling eliminates priority inversion for all the workloads we investigate, and enables better management of very high contention in ways that are not possible with traditional locks.

Chapter 4

Dependence Aware Transactional Memory

4.1 Introduction

Transactional memory provides the abstraction of atomic, isolated execution of critical regions. A transactional *conflict* occurs when one transaction writes data that is read or written by another transaction. When the ordering of all conflicting memory accesses is identical to a serial execution order of all transactions, the execution is called *conflict-serializable* [27].

Transactional memory systems are typically implemented using optimistic techniques: the system speculates that critical sections are safe to execute concurrently as transactions, and detects conflicts between concurrent transactions by observing the memory addresses read and written. When two transactions access the same memory cell and at least one of those accesses is a write, the system *assumes* that a non-serializable schedule has occurred. Because a non-serializable schedule may yield incorrect results the system must take some action to handle the conflict.

TM systems typically respond to conflicts between transactions by forcing one or more of the involved transactions to restart or block, which yields a level of concurrency that is equivalent to that of *two-phase locking* [27]. In two-phase locking ownership of resources needed for a task is acquired in a single phase (where the number of resources held is increasing) and released in a single phase. The technique is attractive because following the discipline yields a system of that is free of deadlock. Note that the term *two-phase locking* refers to the order in which ownership of objects is acquired and released, and *locks* need not be the mechanism through which the notion of ownership is expressed. TM systems express ownership of objects by including them in read-write sets, acquiring addresses by reading and writing locations which are added to a read-write set. Any data read or written by one transaction has an implicit lock on it that conflicts with any attempt to write the same data. Addresses never leave a read-write set until abort or commit, at which point they are released atomically. Hence, transactional execution of a critical section has separate growing and shrinking phases for ownership of resources, which is precisely the conditions required for two-phase locking. Hence, despite the fact that TM systems need not use locks explicitly, Even TM implementations both eager and lazy systems [?, 58] only provide concurrency equivalent to two-phase locking.

Two-phase locking is conservative, and does not admit all conflict-serializable schedules. The goal of DATM is to use conflict serializability as the system's safety property, increasing concurrency relative to using two-phase locking.

Dependence-awareness is TM implementation technique that ensures conflict serializability. Dependence-aware transactional memory (DATM) manages conflicts by making transactions aware of data dependences and enforcing consistent commit orderings, and in some cases, by forwarding data values between uncommitted transactions. Dependence-awareness can allow transactions that encounter conflicting transactions that encounter transactional conflicts to commit safely if they are conflict-serializable. Because the system minimizes needless aborts, it increases concurrency and allows software to make better use of parallel hardware than current TM systems. Dependence-awareness is *safe*—transactions remain atomic and isolated in the same way as current TM systems.

Dependence-awareness is most profitable for workloads that have significant write-sharing. Write-sharing is problematic for TM in general because it can introduce the overhead of multiple restarts on critical sections that ultimately need to be serialized. Other approaches to this problem [32, 34, 40, 60, 64, 88, 89, 96]. involve mechanisms that complicate the programming model and require the attention of skilled programmers to be safe and effective. Dependence-awareness, by contrast, is completely transparent to the programmer. Handling write-sharing transparently is paramount: many common data structures, like shared counters and linked lists write-share data that cause performance problems in conventional TM systems, suggesting that relegating the task of getting good write sharing performance to "expert programmers" is untenable. Because dependence-awareness admits concurrency where current designs cannot, it provides good system performance without burdening programmers with exotic new programming issues.

4.2 Increasing concurrency with DATM

The dependence-aware model creates and tracks dependences between transactions that access the same datum, possibly allowing data to be forwarded speculatively from one transaction to another. Dependences let DATM commit transactions that a conventional TM would restart or block, making better use of concurrent resources.

4.2.1 Shared counter example

Data dependences are represented using standard notation; for example, W→R means a memory cell was written by one transaction and then the same cell was

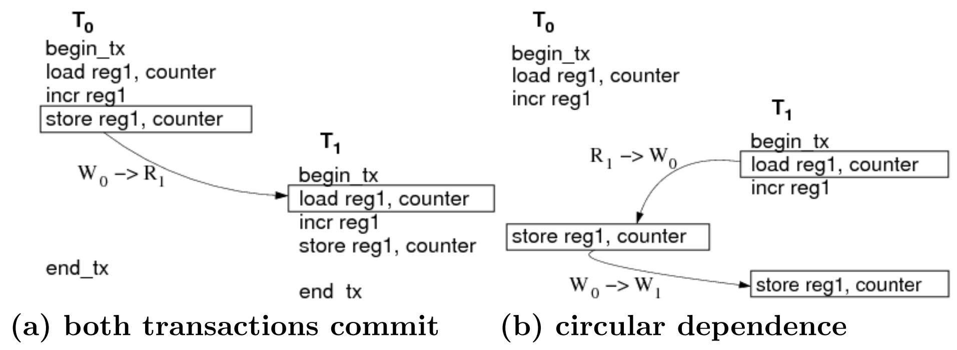
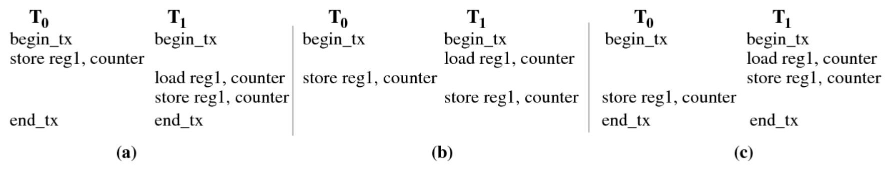


Figure 4.1: Two transactions increment the same counter, illustrating (a) a successful commit using dependences with data forwarding, and (b) an abort due to circular dependences.



T_0	T_1	T_0	T_1	T_0	T_1
begin_tx	begin_tx	begin_tx	begin_tx	begin_tx	begin_tx
store reg1, counter			load reg1, counter		load reg1, counter
	load reg1, counter	store reg1, counter			store reg1, counter
	store reg1, counter		store reg1, counter	store reg1, counter	
end_tx	end_tx			end_tx	end_tx
(a)		(b)		(c)	

Figure 4.2: Three execution interleavings of two simple transactions. Time flows down. All memory references are to the same shared counter. DATM can accept interleavings (a) and (c), indicated by the presence of the end_tx instruction.

subsequently read by a different transaction. Dependences are subscripted with transaction numbers to indicate which transactions are involved. While the generic term “memory cell” indicates that the granularity of the datum is not intrinsic to the model, in this chapter a “memory cell” is a cache line unless otherwise stated.

Consider the shared counter shown in Figure 4.1(a). Assume that two different threads on two different processors (P_0 and P_1) execute this code in two different transactions (T_0 and T_1). The executions overlap in time as shown in the figure, with time flowing down. If the counter value starts at 0, the figure shows T_0 forwarding its counter value (1) to T_1. DATM establishes a $\mathrm{W}_0 \rightarrow \mathrm{R}_1$ dependence for the counter, and ensures that T_1 commits after T_0. The transactions are allowed to proceed concurrently even though they both write the same memory location. The counter's final value is two, which corresponds to the serialization order T_0, T_1.

The interleaving in Figure 4.1(a) is conflict-serializable, but would not be allowed by the two-phase locking style of conflict detection done by current TM systems. In most current TM systems, after T_0 reads and writes the counter, any subsequent access to the counter by T_1 is considered a conflict, either forcing T_1 to block or one transaction to abort.

The interleaving in Figure 4.1(b) is not conflict-serializable, so both transactions cannot successfully commit. Here, T_0 writes the counter after it is read by T_1, creating a $\mathrm{R}_1 \rightarrow \mathrm{W}_0$ dependence, which constrains T_0 to commit after T_1. However, when T_1 writes the counter, it creates a $\mathrm{W}_0 \rightarrow \mathrm{W}_1$ dependence, which constrains T_1 to commit after T_0. The dependence graph contains a cycle, and if both transactions were to commit, the counter would have the wrong value. DATM handles this potential conflict by detecting the cycle—T_0 is dependent on T_1 and T_1 is dependent on T_0. It aborts one of the transactions to break the cycle.

4.2.2 Accepting more interleavings

Figure 4.2 shows three different interleavings (called schedules in the database literature) for the memory references of transactions that increment a shared counter. Interleavings (a) and (c) are conflict serializable. In (a), T_0 can be serialized before T_1, and in (c), T_1 can be serialized before T_0. Interleaving (b) is not conflict serializable. DATM accepts interleavings (a) and (c), while conventional TM implementations do not.

Of course, accepting more interleavings does not by itself imply that DATM will outperform conventional approaches, since many other factors impact actual performance. However, by accepting more interleavings DATM increases the likelihood that parallel resources are utilized when transactions execute concurrently—instead of conflicting, concurrent transactions can coordinate and both commit.

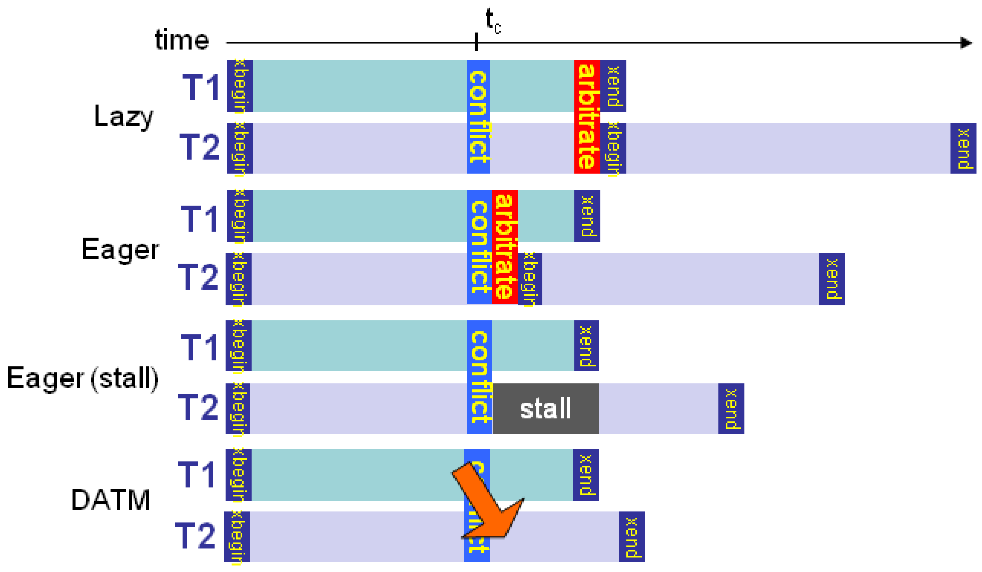


Figure 4.3: Two transactions that conflict while incrementing a shared counter. The bottom pair shows the dependence-aware implementation, while others are conventional HTM techniques. Assume that transaction $T1$ always commits first.

4.2.3 Comparison with other conflict resolution strategies

Figure 4.3 compares how DATM and existing systems execute a pair of transactions that conflict on a single shared datum. DATM creates a dependence from $T1$ to $T2$. Neither $T1$ nor $T2$ is forced to block or restart. DATM commits $T2$ earlier than the other conflict resolution strategies because it can accept memory access interleavings that require the other systems to block or restart.

Figure 4.3 shows eager conflict detection (done at the time of the memory reference) [58] and lazy conflict detection (done at commit time) [?]. Eager conflict detection with restart causes $T2$ to restart on the conflict, and $T2$ conflicts again. Eager conflict detection with stall-on-conflict causes $T2$ to stall until $T1$ commits. Finally, with lazy conflict detection, $T2$ must restart when it tries to commit. Execution interleavings that cause stalls or restarts with current conflict resolution

Dependence	**Forward**	**Restart**
$W_0 \rightarrow W_1$	No	If in cycle
$R_0 \rightarrow W_1$	No	If in cycle
$W_0 \rightarrow R_1$	Yes	If in cycle, and T_1 must if either: **a)** T_0 does. **b)** T_0 overwrites forwarded data with new value.

Table 4.1: Summary of dependence types and their properties.

strategies are committed safely by DATM.

4.3 Dependence-aware model

This section presents the dependence-aware model, describing how the system maintains dependences and how those dependences affect transactions. The dependence aware model admits all conflict serializable schedules.

4.3.1 Dependence types

Table 4.1 shows a summary of dependence types and their properties. The notation W→R denotes a read after write (RAW) dependence—one transaction reads a cache line that was written by another transaction. Dependences are subscripted with transaction numbers, so that $W_0 \rightarrow R_1$ means a write from transaction T_0 was read by transaction T_1. All dependences restrict commit order. If there is a $X_A \rightarrow X_B$ dependence, then transaction A must commit before B.

The system tracks all dependences at the level of cache lines creating new dependences between transactions in response to memory accesses at runtime. The ordering of transactions depends on their dynamic behavior. The "Yes" in the **Forward** column for W→R dependences means the system forwards the data in the cache line when the dependence is created. The system records that the cache line has been forwarded.

For a $W_0 \rightarrow R_1$ dependence, we call T_0 the *source* transaction and T_1 the *destination*, or the dependent. The destination transaction must restart if the source restarts, because the destination has read data forwarded by the source. To maintain serializability, a dependent transaction can read a value from a source transaction only if that value will be the final value of the cache line for the source transaction. So the destination transaction must restart if the source transaction overwrites the data it forwarded. Table 4.1 lists the cases when restarts are necessary.

Dependences are created per cache line on first access to the cell. Subsequent accesses to the same object do not affect dependence structure For example, if T_0 writes a cache line that T_1 then writes, and then T_1 reads the cache line the resultant dependence is formed on the basis of the initial write and is $W_0 \rightarrow W_1$. When a transaction commits or aborts, all of its dependences disappear. The next section discusses how dependences between transactions form when they both access multiple memory cells.

4.3.2 Multiple dependences

Multiple dependences arise when two transactions conflict on more than one cache line. Each cache line on which two transactions conflict creates a separate dependence. To manage multiple dependences between two transactions, the model has the restrictive dependence rule: The relationship between transactions is governed by the most restrictive dependence in each direction. W→R is more restrictive than W→W and R→W dependences, and the latter two are not ordered relative to each other.

If a transaction is the source for a R→W dependence, and later it writes and forwards a different cache line to the same destination transaction (thereby creating a W→R dependence), the transactions are constrained by the more restrictive W→R dependence. Both dependences are still tracked in the model.

If more than two transactions concurrently access the same cache line, then the first two will create a dependence as described above. The third transaction will create its dependence with the most recent writer of the cache line. The latest writer provides the most up to date version of the cache line. Conceptually, the dependences among transactions form a transaction dependence graph with a directed link between two transactions if there is a dependence between them on any memory cell.

4.3.3 Cyclic dependences

All dependences restrict commit order: a transaction must wait at commit time for any transaction that it depends on to commit. If cycles arise in the transaction dependence graph, the cyclic chain of dependences may cause deadlock. Dependences arise from reads and writes of memory cells, so a cycle indicates that the transactions have interleaved in a way that is not conflict serializable.

While there are several ways to handle cycles, our model avoids them. If a memory access would cause a cycle in the dependence graph, the system restarts at least one transaction in the cycle. The system does not allow cycles to form.

Another way to avoid cycles is to allow dependences only from older transactions to younger transactions. *Timestamp-ordered dependences* go in a single direction only, so they cannot form cycles. However, timestamp-ordered dependences do restrict concurrency more than a policy that allows dependences between any two transactions.

Contention management is important for dependence-aware transactions, just as it is for conventional TM systems [70, 83]. When the system detects a cycle in the dependence graph, it must restart at least one transaction in the cycle to break it. The contention management task is to preserve as much concurrent work as possible, such as by restarting transactions that do not have dependents.

4.3.4 Disabling dependence tracking: no-dep mode

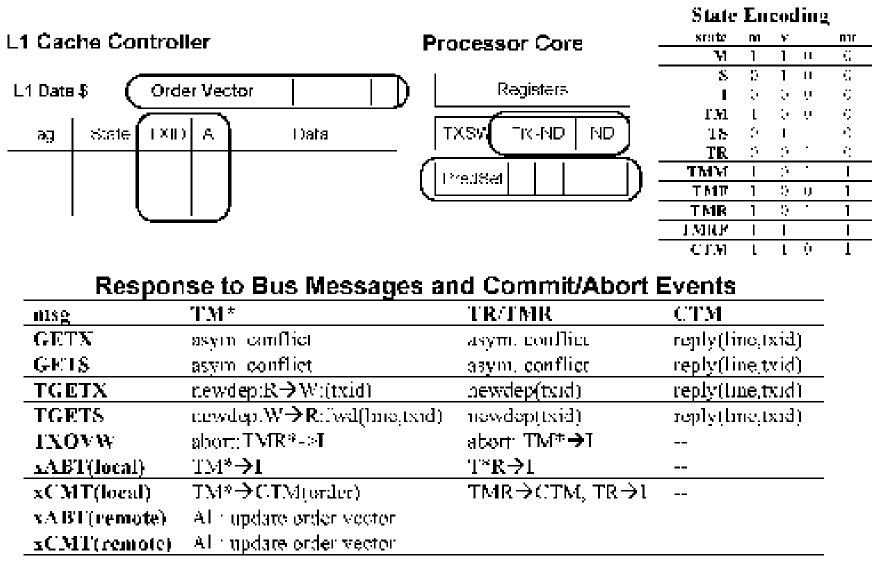
One attractive property of dependence-aware transactions is that they co-exist with other conflict resolution strategies for ensuring safety. Restarting a transaction in *no-dep mode* disables dependence tracking for a particular transaction. Sections 4.3.5 and 4.3.6 explain uses of the no-dep mode.

4.3.5 Exceptions and inconsistent data

Because the model forwards data between transactions, it is possible that a transaction can read invalid data, which in turn can lead to exceptions or infinite loops. Inconsistent state seen by destination transactions are eventually made consistent at the completion of the source transactions. The writes that bring the source transactions into a consistent state cause a restart of the destination due to overwrites of forwarded data. The restart of the destination eliminates infinite loops that are not part of the application's serial behavior.

A transaction that has read inconsistent data can throw an exception before subsequent execution of the source transaction causes the destination to restart. The hardware informs the OS through the transaction status register if the currently running transaction has read forwarded data. The OS exception handlers suppress these exceptions and, according to its policy, can restart a transaction in no-dep mode to avoid further spurious exceptions. Section 3.3 quantifies the small number of times transactions execute in no-dep mode for our prototype.

Program asserts must also be made dependence-aware. Assert failures in transactions that have read forwarded data can be restarted or the failure is delayed until the source transaction commits.



State Encoding

state	m	v		mr
M	1	1	0	0
S	0	1	0	0
I	0	0	0	0
TM	1	0	0	0
TS	0	1		0
TR	0	0	1	0
TMM	1	0	1	1
TMI	1	0	0	1
TMR	1	0	1	1
TMRF	1	1		1
CTM	1	1	0	1

Response to Bus Messages and Commit/Abort Events

msg	TM*	TR/TMR	CTM
GETX	asym. conflict	asym. conflict	reply(line,txid)
GETS	asym. conflict	asym. conflict	reply(line,txid)
TGETX	newdep:R→W:(txid)	newdep(txid)	reply(line,txid)
TGETS	newdep:W→R:fwd(line,txid)	newdep(txid)	reply(line,txid)
TXOVW	abort:TMR*->I	abort: TM*→I	--
xABT(local)	TM*→I	T*R→I	--
xCMT(local)	TM*→CTM(order)	TMR→CTM, TR→I	--
xABT(remote)	All: update order vector		
xCMT(remote)	All: update order vector		

Figure 4.4: DATM architecture overview. DATM-specific state and structures are highlighted with dark lines.

4.3.6 Cascading aborts

Cascaded aborts occur when one transaction's abort causes other transactions to abort. For example, a cascaded abort happens when a source transaction forwards a value to a destination transaction and the source aborts—the destination must abort as well. In DATM, cascaded aborts arise only from W→R dependences, where the source aborts or overwrites forwarded data. This data sharing pattern, with one transaction updating a variable multiple times while other transactions read it, is not conflict serializable. Any safe transactional system will serialize such transactions, either by stalling or aborting. Section 3.3 quantifies the small effect of cascaded aborts on the performance of our prototype.

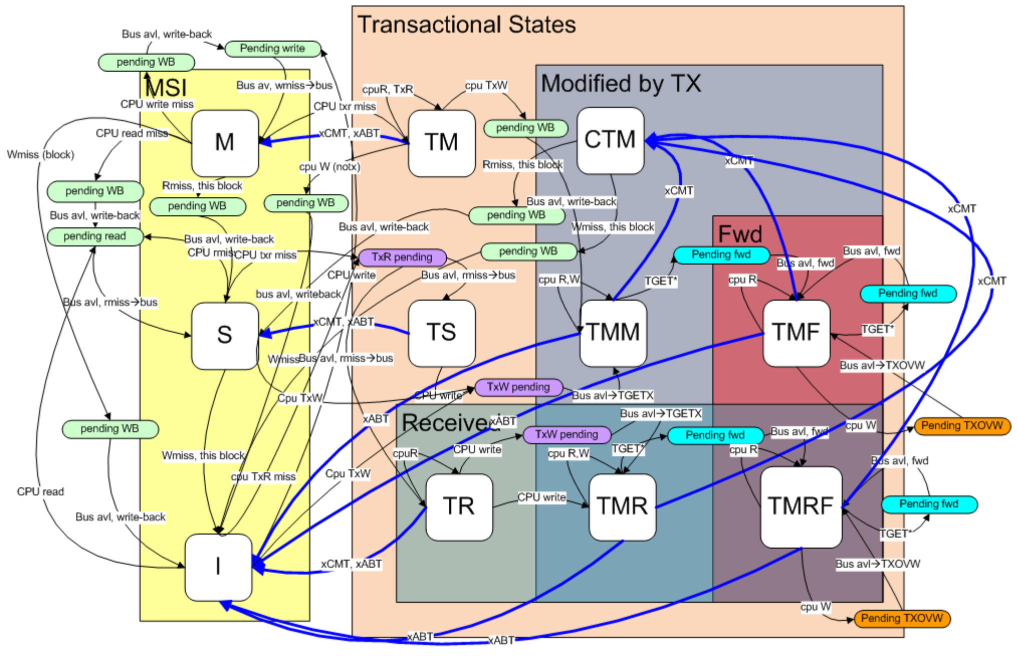


Figure 4.5: State diagram for the FRMSI cache coherence protocol. Standard transitions between MSI are omitted for clarity. Transitions out of I are omitted as they are the same as those out of S.

4.4 Hardware design

This section discusses the hardware design for dependence-aware transactional memory. Key elements in the design are shown in Figure 5.2. The design must implement commit ordering, version management, W→R data forwarding, restart when forwarded data is overwritten (called a *forward restart*), and cyclic dependence prevention. To understand how these pieces interrelate, we first describe solving them with a minimum of new hardware. We then refine the design to improve performance while still keeping hardware state and hardware complexity low (Section 4.4.4). The design described here relies on broadcast coherence.

DATM can be implemented with a novel cache coherence protocol called FRMSI (**F**orward **R**eceive **MSI**: pronounced like pharmacy), along with either an an ordered vector of transaction IDs maintained at each cache, or a timestamp table.

The cache coherence protocol supports version management, helps order write backs of committed state, and handles data forwarding and forward restarts. DATM relies on global ordering for transaction commits and write backs of data modified in committed transactions, as well as for prevention of deadlocks and cycle dependences. Support for such ordering decisions can be implemented either using timestamps generated at transaction begin for contention management [67], or using an ordered vector of transaction IDs. The FRMSI protocol relies on the augmentation of cache lines with a transaction identifier [48, 70], shown as **TXID** in Figure 5.2.

An important design principle in DATM is that while dependences enable concurrency not currently accessible in TM designs, dependences are not a requirement for transactions to proceed. If any hardware resources or structures in the DATM design reach a limit, dependences for that transaction are dynamically disabled by restarting in a force-no-dependence mode (Section 4.3.4) that resembles a current TM design. DATM is a *best effort* design, and contains no explicit overflow (sometimes called virtualization) strategy for when transactional state overflows hardware buffers—it can use any of the many current proposals [10, 18, 19, 68, 86].

4.4.1 Transaction status word

DATM adds two bits to the transaction status word, a register that holds the current state of the running transaction. One bit is a no-dependence bit (shown as **ND** in Figure 5.2), which indicates that the current transaction has not entered into any dependences. Transactions that have no dependences can be created and can commit without support from dependence-aware mechanisms: an explicit bit makes this check efficient. Decoupling dependence- and non-dependence-aware transactions ensures that regardless of the state of the dependence-aware hardware, transactions in the system can still commit, and forward progress can always be made.

DATM also adds a force-no-dependence bit (**Frc-ND**), which disallows the current transaction from entering transactional dependences. This state allows DATM to fall back into a traditional eager conflict-management HTM mode [58].

4.4.2 FRMSI coherence protocol

DATM is implemented with support from the FRMSI cache coherence protocol, which extends the MESI protocol, and has 11 stable states. The state diagram is shown in Figure 4.5. The E state is omitted for simplicity; it can be added as an optimization, but is not necessary. We could reduce the number of states if we use signatures [16] to track forwarded and received bytes. First we review the mechanics of the protocol and then show how the protocol achieves the goals of DATM.

Version management in DATM is complicated by data forwarding, which results in the ability of multiple caches to modify the same cache line. FRMSI contains states for forwarding and receiving data, allowing this kind of data sharing.

The TM and TS states are entered by lines in M or S that are read during a transaction. These lines can transition back to M or S when a transaction commits or aborts, because they are not modified during the transaction.

All TM* states (shaded) and TR (transaction received) states transition to invalid if the transaction aborts. All TM* states (note that TM* does not include TM) indicate a line is written during a transaction. These writes are buffered in the cache, but are discarded if the transaction is not successful, by a transition to I. A TR line must revert to I on abort because it contains speculative data received from an active transaction.

TMR and TR are states for cache lines that receive forwarded data, while TMF and TMRF are states for cache lines that have forwarded their data. These states are explained in detail in Section 4.4.2. Commit of lines that are modified in a transaction is the subject of the next Section.

Committing

All TM* states (shaded) transition to CTM (committing transactional modified) on a commit. The CTM state is much like the M state in that it indicates a line that has been modified with respect to main memory and requires writeback. However, a line in CTM state must obey the ordering restrictions associated with the transaction that wrote the line. To understand the need for a CTM state, consider that FRMSI allows lines to exist in TM* states in multiple caches, but cannot allow lines to exist in the M state in multiple caches. This could be addressed without an additional state if, on completing a transaction, all lines in TM* state were to atomically write back to memory, stalling the transaction commit until all write backs complete and transitioning those lines to state I. Such a solution is unattractive because waiting for write backs increases the latency of transactional commit, which must be fast for TM to provide good performance [70].

Instead, the CTM state allows write backs to take place after transaction commit while still preserving ordering with respect to other transactions in the system. Transaction commit causes all updates made by a transaction to linearize [37] to that commit point. Consider a line that is written by transaction A, and then forwarded to B which also updates it. A is constrained to commit before B. If commit includes all write backs, then after both transactions commit, B's line is in memory and in its cache in state M, which is correct. With delayed write back for transactionally modified state, the lines enter CTM, where A's line is constrained to never overwrite B's. Any access to the line gets B's version, which is the latest one. The CTM state uses the order vector to order accesses and write backs to committed data, as explained in Section 4.4.3.

Forwarding and receiving

One of the chief goals of FRMSI is to enable cache line forwarding among transactions. When a cache controller sees a transactional bus read (**TGETS**) for a line that it has in state TMM or TMR (the line has been locally modified in a transaction), then it responds with the line and the identifier of the transaction that wrote the line, and moves the state into TMF or TMRF. The receiving cache can be transitioning from I or S into TR, the transactional received state.

Forwarded lines (states TMF and TMRF) publicize writes, in effect using an update protocol by sending a **TXOVW** message on the bus to indicate that previously forwarded speculative values are now stale. Any cache that has the line in a received state must abort its transaction if it sees a write to the line, because speculative data it received has been overwritten. The transaction only aborts in TMRF if the overwrite is from an earlier transaction where ordering is defined in Section 4.4.3. We later describe additions to the protocol, which reduce the granularity of detecting the overwriting of forwarded data (Section 4.4.4).

Suspending transactions

DATM allows suspended transactions [70, 96], and it allows transactions with dependences to suspend and resume. Cache lines store the transaction ID to enable suspend and resume [70]. However, any attempt to create a dependence with a suspended transaction will fail and the operation will be handled as a transactional conflict, requiring the restart of one of the transactions involved.

Processor identifiers are insufficient for dependence management and cycle detection when transactions can suspend. For example, to support 3 inactive transactions per processor, the transaction identifiers have 2 bits more than the number of bits in the processor identifier.

FRMSI transient states

For a bus-based design that doesn't use a split transaction bus, FRMSI introduces 19 transient states. In general, transient states are required in this protocol when forwarding occurs (forward pending states), when overwrites of previously forwarded data require a TXOVW broadcast, and on transitions from non transactional (MSI) states to transactional (T*) states, since many of these transitions must be globally visible. However, the number of transient states does not grow polynomially with the number of stable states for a number of reasons:

- DATM is a best-effort mechanism: eviction of lines in any transactional state results in an abort of the transaction. As a consequence, any lines held in speculative modified states can transition straight to invalid (I), and do not require write-back pending states.
- Lines held in any received state (T*R*) are isolated: modifications made to them locally require no permissions upgrades.
- Transitions from transactional states to non-transactional states require no transient states since they reflect a local commit or abort. Note that the commit or abort event itself requires global visibility and therefore broadcast, but this is a per-transaction event, not a per-cache-line event.

Also note that use of hardware signatures to represent forward and recieve states could eliminate 5 states from this protocol.

4.4.3 DATM ordering requirements

DATM provides conflict serializability by ordering dependent transactions with respect to each other, and by linearizing their updates to transaction commits. We first present the ordering requirements of DATM and then discuss two implementation strategies: an order vector and a timestamp table.

These are DATM's ordering requirements.

1. Dependent transactions must commit in order.
2. Transactions that form dependences by receiving forwarded speculative data must become dependent on the most recent writer of that data.
3. Cyclic dependences must be detected in advance, and avoided by restarting one or more transactions.
4. Dependences are transitive: when transactions abort, dependence ordering must be preserved for transactions that remain active.
5. Caches with the same line in CTM state must write back the lines in the order dictated by their commit order, and subsequent requests for the line must be serviced from the last cache to commit.

The succeeding text has numbers in bold parenthesis to indicate how the design enforces the given requirement (e.g., **(1)** marks the explanation of how dependent transactions commit in order).

The order vector

A DATM implementation can support ordering by maintaining an **order vector** of transaction IDs in each cache. Each cache that contains a transaction with dependences maintains a copy of the order vector, and all copies have identical data. Each entry in the list has a transaction identifier, a valid bit and an active bit. The active entries in the vector topologically sort the dependence graph of the currently active transactions. The order vector provides the serialization order for active transactions and for writebacks of committed transactions whose results are still cache resident.

The vector reflects the superset of all dependences between transactions. Dependences are created when a cache snoops a memory access on the bus that is responded to by a cache rather than memory (using a mechanism analogous the *shared* response to **GETS** request for a line in **S** or **E** in MESI). New dependences are appended to the list, after all valid transaction identifiers. If transaction A forwards a cache line to transaction B, then A,B is appended to the list (with the rightmost position being the newest transaction). Each cache must see these dependences in the same order if the vector is to be identical at every cache. In our bus-based design, the bus ensures all dependences are seen in the same order: FRMSI would require extra messages to be extended to support a directory protocol.

Timestamp ordering

Timestamp-ordered dependences are implemented with a *timestamp table*. Each cache that contains a transaction with dependences (active, or with pending write backs) maintains a copy of the timestamp table, and all copies have identical data. Each entry in the table has a transaction ID, a timestamp, a valid bit and an active bit.

Using timestamps is similar to using the order vector, any difference are noted below in the discussion of the order vector. The main simplification of timestamp ordering is that dependences only go from older to newer transactions, so cyclic dependences cannot arise **(3)**.

Meeting ordering requirements

A transaction can only commit if it is the first (leftmost) active transaction in the order vector **(1)**. Being first ensures that this transaction does not depend on any others. When using timestamps, it can only commit if it is has the smallest timestamp in the timestamp table **(1)**.

If a transaction receives a cache line, its ID gets appended (on the right) to the order vector. The receiver uses the order vector to determine the last dependent transaction that provides the data for that line **(2)**. Suppose A has forwarded a line to B. If C reads that line, then C should receive the line from B, not A. If both A and B attempt to forward the line to C, the order vector is used to determine that B's data should be received and A's should be discarded. Using the received cache line from the transaction with the highest timestamp also creates a dependence with the latest writer **(2)**.

When a new dependence is added, the hardware checks if the transaction ID already appears in the vector of an active entry *to the left of the current transaction.* If it does, there is a cycle in the dependence graph and some transaction in the graph must be restarted **(3)**. The order vector thus provides a simple mechanism to detect cyclic dependences. Cyclic dependences cannot form when using timestamps **(3)**.

When a transaction aborts, it must publish this event to the coherence protocol by placing the **xABT** message on the bus along with its transaction ID. An abort message notifies other processors to turn off active and valid bits for that transaction ID in the order vector, while leaving its predecessors and successors in place **(4)**. Therefore, if a dependence chain of transactions A $\rightarrow$ B $\rightarrow$ C arises, B can abort without affecting A or C (provided the dependences are not forwarding dependences) C remains serialized behind A. The **xABT** message is not needed with timestamps, but an aborted transaction maintains its timestamp and hence its place in the serialization order **(4)**.

Commit must also be made visible to the coherence protocol, placing the **xCMT** message on the bus along with its transaction ID. The commit message notifies other processors to turn off the active flag but leave the valid bit. This way, the commit can retain its position in the order vector to order write backs for lines moving to the CTM state. This is explained in the next Section, and is handled identically when using timestamps.

Write backs

Position in the order vector is used both to order the commit of active transactions and to order writebacks for lines modified in committed transactions. Entries in the order vector for committed transactions with pending writebacks are valid but not active. Transactions remain valid in the order vector until all lines from the transaction have exited the CTM state. The ordering remains valid until all of the data updated during the transaction leaves the CTM state. A cache can detect when it has written back the last CTM line for a given transaction ID and at that point it sends a message to make the ID invalid in the order vector. Detection can be implemented using simple logic on the state bits, or if the CTM state were recoded as a single bit, as a wired OR.

Note that non-dependent transactions that include the TM and TS states can execute and commit while lines are in the CTM state. If the processor accesses any line in a CTM state, the line is written back and then the processor processes the operation as if the line were in M.

All cache lines in CTM are marked with their transaction identifier (**TXID**), and the serialization order is determined by looking up the TXID in the order vector or timestamp table **(5)**. The order vector or the timestamp table enforce a single, global order for write backs in addition to active transactions.

Write backs are ordered and can be squashed. Assume A forwards a line to B, B overwrites the line, and then both transactions commit. If A sees B write back the line that A forwarded, A can transition the line from CTM to I without writing back. B's version is serialized after A.

The situation is similar for bus reads. If another processor issues a bus read for the line that both A and B have in CTM, then both transactions can respond to the request, write back the value, and transition the line to state S. If B responds first and A observes B's response, then A can squash its own response and transition the line to I.

Timestamps are used to order write backs in the same way the order vector is used **(5)**.

Capacity of the order vector or timestamp table

Any transaction that needs a dependence can claim the newest index after the last valid index. Once claimed, the index serializes the transaction after any that has results that might be written back. Making the order vector large will minimize the probability that the vector will fill. The order vector can be large because it is not communicated and it is mostly consulted during a cache miss, when its access latency can be overlapped with data fetch.

No matter how long the order vector is, it can fill because with pathological line replacement, lines can remain in the CTM state indefinitely. A new transaction cannot get a dependence if the order vector is full or if it has the same identifier as a valid entry in the order vector. In these cases, transactions simply restart with the force-no-dependence flag and the computation continues.

When the order vector fills, the last entry must write back its CTM lines. Each cache can monitor if it has a transaction that occupies the last entry and initiate write backs for the lines that transaction has in the CTM state.

The timestamp table can also fill, which would also require a cache to write back lines in the CTM state in order to free an entry in the table.

4.4.4 Performance optimization

This section describes additions to the basic protocol to optimize performance. These changes allow the hardware to manage dependences at the word level while keeping writeback and most cache coherence operations at the cache-line level. It also prevents short transactions from convoying behind unrelated long-running transactions.

As a motivating example, consider a line that starts with its data equal to all zeroes and that is not present in any cache. Transaction A writes word 0 with value A and transaction B writes word 1 with value B. The write from transaction A causes the line to enter A's cache in the TMM state. B's write of word 1 results in a bus read for the cache line that is forwarded from A's cache. A's cache moves the line into TMF and B's has it in TMR (not TR because it wrote the word after receiving the line).

In the cache-line-based design, if A writes word 2, it generates a bus write that causes transaction B to abort, due to a circular dependence $W_A \rightarrow R_B$ and $W_B \rightarrow R_A$. However, there is no circular dependence at the word granularity because B does not read the word A writes. Eliminating these false cycles will improve DATM's performance.

Per-word accessed bits for received states

We augment the cache with per word access bits (labeled **A** in Figure 5.2). On receiving a line (TR, TMR or TMRF), the processor resets accessed bits, one per word in the line. Every time the processor reads or writes a word, it sets the access bit for the word. The access bits play an important role in interpreting bus writes to forwarded lines. Such writes either cause restarts, update the value of the word, or are ignored.

The rules for dealing with overwrites to forwarded lines enforce the obvious causality: previous transactions cannot overwrite data the current transaction has read or written, but a future transaction may. When the processor sees a bus write for a line that it has received, it compares the order vector entry of the transaction writing to the bus with the transaction identifier of the line. If the bus write index is earlier and the access bit is clear, the word is updated (a previous transaction is updating a word untouched by the current transaction). If it is later, the word is not updated (the word belongs to a transaction that is serialized after this one). If the access bit is set and the index is earlier, the transaction aborts (forward restart). Otherwise, the transaction ignores the message (a future transaction will change the same word this transaction changed).

Publishing writes to the bus for forwarded lines make these states act like an update protocol. All receivers have the same value as earlier transactions for words the receivers do not touch. This value agreement allows cache lines to be written back without lost updates. In the above example, if A writes word 2 and that write is not propagated to B, then the line that B commits will have a zero for word 2, not an A, which is a lost update.

Predecessor transaction set

The main problem with having a single ordered vector for all transactions in the system is that short transactions may have to wait for long transactions. For instance, if transaction A forwards data to B, and then transaction Y forwards data to Z, the order vector will read A,B,Y,Z. If Y and Z are very fast and A and B are slow, then throughput will suffer, as Y and Z must be at the head of the order vector in order to commit (condition **(1)** above).

We add a set of predecessor transactions to each processor (depicted as **Pred-Set** in Figure 5.2), to prevent transactions having to wait for unrelated transactions. A transaction can commit when its predecessor set is empty, it does not need to be at the head of the order vector. The transaction builds the predecessor set with the identifiers of any transaction from which it receives data. The set requires a maximum of only P entries, where P is the number of processors. An active transaction must restart if it wants to commit but has a suspended predecessor. The set can be smaller than P and, if it fills, the transaction restarts in force-no-dependence

	Configuration
Processor	Pentium-4-like x86 instruction set, 1 GHz, 1 IPC
L1	Each core has separate data and instruction caches. 32 KB capacities, with 8-way associativity, 64-byte cache lines, lru-replacement policy, 1-cycle cache hit.
L2	4 MB capacity, 8-way associative, with 64-byte cache lines, 16-cycle access time.
Memory	1GB capacity, 350 cycle access time.
MetaTM	Timestamp contention management, linear backoff policy, word granularity conflict detection.

Figure 4.6: Architectural parameters of simulated machines.

mode. Timestamp-ordered dependences benefit in the same way from the predecessor transaction set.

4.5 DATM Evaluation

This section provides details of the DATM simulation model, benchmarks and experimental results.

4.5.1 Prototype model

We implement a dependence-aware HTM model by modifying a publicly available HTM simulator (MetaTM [70]). The model is implemented as a module in the Simics 3.0.27 machine simulator [47]. The core architectural parameters are shown in Figure 4.6. We evaluate the model with a 16-way SMP configuration (except for TxLinux benchmarks, where an 8-way SMP is used). Each processor has a private L2, and the L1 data caches contain both transactional and non-transactional data.

We modify the MetaTM cache coherence protocol, which is based on a MESI snooping protocol, to support transactional dependences using FRMSI. The latency of forwarding data between processors is conservatively modeled as a write back and a read from memory. Bus arbitration and bandwidth constraints are not modeled. The L2 cache also requires transactional state so that transactional state is visible

Name	*Description*
bayes	From STAMP [55], learns the structure of a Bayesian network, "-v32 -r384 -n2 -p20 -s1"
config, pmake	These benchmarks report transactions created in TxLinux, a Linux-variant operating system with several subsystems converted to use transactions for synchronization, instead of spin-locks [70,75]. The workload involves several user-mode applications (configure, make) which are running on the transactional OS.
counter, counter-tt	A micro-benchmark where threads increment a single shared counter. counter-tt adds think-time, to simulate longer transactions.
genome	From STAMP, a gene-sequencing bioinformatics application, "-g 1024 -s16 -n 4000000"
kmeans	From STAMP, implements a K-means clustering algorithm, "-m40 -n40 -t0.05 -i random-n65536-d32-c16.txt"
labyrinth	From STAMP, models an engineering program which performs path-routing in a maze, "-i random-x48-y48-z3-n48.txt"
list	A micro-benchmark which manipulates a linked-list. On a traversal, a thread may search for a random node (60%), insert a node (20%) or delete a node (20%). The number of nodes is 8192, node traversals per thread is 512, and the number of threads is set to four times the processor count.
ssca2	From STAMP, a scientific application with different kernels operating on a multi-graph, " -s13 -i1.0 -u1.0 -l3 -p3"
vacation	From STAMP, models a multi-user database, "-t 20000 -n 10"

Figure 4.7: Workloads used in DATM evaluation. TxLinux and list are kernel-mode transactions, while the other benchmarks run in user-mode. All benchmarks use a number of threads equal to the number of processors, unless noted otherwise.

to the coherence protocol (how to support transactional variants of MESI in the presence of multi-level private hierarchies is an open research question, and previous proposals would also be forced to make similar tradeoffs [87].)

The workloads are described in Table 4.7. They include a transactional operating system, several STAMP [55] benchmarks, and two micro-benchmarks to focus on specific data structures.

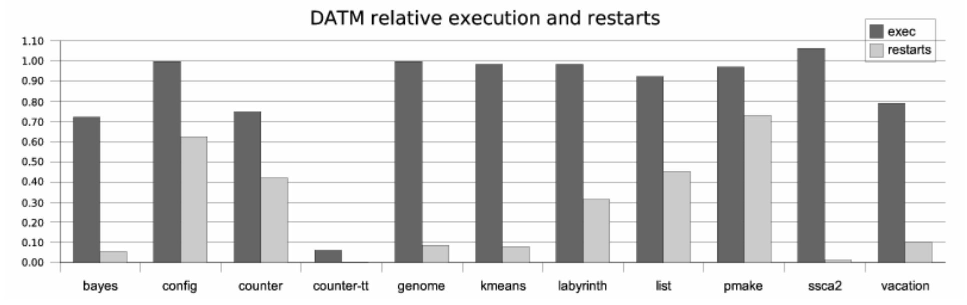


Figure 4.8: Relative execution time and restarts per transaction in DATM, normalized to MetaTM. Lower is better.

4.5.2 Experimental results

Figure 4.9 shows the basic runtime characteristics of the workloads on both MetaTM and DATM. Figure 4.8 shows graphically how DATM, normalized to MetaTM, reduces the execution time and number of restarts. The results show that, in almost all cases, DATM improves or does not harm the performance of realistic workloads. DATM increases concurrency by reducing the average number of restarts per transaction and average backoff cycles per transactions. Reducing restarts does not necessarily improve performance, but it does when the reduction is an indicator of increased concurrency.

In particular, the bayes and vacation STAMP workloads show a dramatic reduction in restarts, and a 39% improvement in execution time. The remaining STAMP benchmarks have little contention, so DATM does not change their performance. For instance, while DATM reduces ssca2 restarts by two orders of magnitude, it has only 0.1 restart per transaction on average under MetaTM.

Performance for the TxLinux workloads (pmake, config) is mostly flat because they spend a small amount of time executing transactions [70]. DATM reduces restarts substantially, but these restarts are not a performance problem. The counter micro-benchmark (especially with think-time) is able to dramatically benefit

benchmark	**exec**		**tx**	**avg rst/tx**		**pct rst**		**avg bkcyc/tx**	
bayes	0.008	0.006	762	13.9	0.8	9.38	5.45	27,284	760
config(8p)	3.543	3.537	4698136	0.1	0.1	1.91	1.86	1	0.5
counter	0.095	0.071	160000	9.4	4.0	59.59	76.83	519	119
counter-tt	3.067	0.189	16000	1056.3	0.1	99.99	0.39	264,061	0.1
genome	0.212	0.212	352376	0.1	0.1	0.21	0.15	1	0.1
kmeans	0.301	0.295	436986	1.1	0.1	11.11	6.33	58	1
labyrinth	0.066	0.065	128	88.5	27.8	36.77	29.74	140,086	41,521
list(8p)	0.386	0.355	78586	0.2	0.1	10.89	5.32	2	0.2
pmake(8p)	0.260	0.253	251844	0.2	0.1	3.93	3.80	5	3
ssca2	0.008	0.008	47304	0.1	0.001	0.15	0.10	28	0.1
vacation	0.030	0.025	20000	8.0	0.9	36.71	29.93	1,123	40

Figure 4.9: Basic transactional characteristics of benchmarks running on on DATM and MetaTM. In cases where two numbers are present, MetaTM is the leftmost number, while DATM is the rightmost number. The "exec" metric is execution time in seconds (user time for STAMP and micro-benchmarks, and kernel time for TxLinux benchmarks), and the "tx" metric is the total number of transactions. The "avg rst/tx" metric is the average number of restarts per transaction, and the "pct rst" metric is the percentage of transactions that restart at least once. The "avg bkcyc/tx" metric is the average number of cycles spent backing off before restart per transaction. All data is for 16 CPUs, except TxLinux benchmarks `config` and `pmake`, which were run using 8 CPUs.

	WR dep	**RW dep**	**fwd rst**	**casc abt**	**cplx conf**	**no-dep mode**	**incns read**	**b-cast write**
bayes	3.8%	8.5%	0.4%	0%	7.0%	1	3	1
config (8p)	0.3%	0.2%	19.7%	2.3%	0.3%	2	1	10,132
counter	90.0%	80.7%	0%	0%	90.8%	0	0	0
counter-tt	99.9%	0.3%	0%	100.0%	99.9%	0	0	0
genome	0.1%	0.1%	0%	14.2%	0.1%	0	1	104
kmeans	8.9%	6.0%	0%	3.1%	7.0%	0	0	40,723
labyrinth	32.0%	32.0%	0%	0.1%	39.0%	6	1	4
list	14.3%	3.8%	3.3%	0.2%	0%	3	0	86
pmake (8p)	0.5%	0.5%	12.9%	6.5%	0.6%	0	10	10,009
ssca2	0.1%	0.1%	0%	0%	0.1%	0	0	0
vacation	35.2%	7.9%	0.4%	3.5%	44.7%	6	34	143

Figure 4.10: Basic dependence-related statistics. The first two columns show the percentage of transactions that were involved in a dependence of that type (W$\rightarrow$W dependences formed between less than 0.2% of transactions for all workloads). The next three show the percentage of restarts that were due to forward restarts, cascading aborts, or complex conflicts. The next two columns provide an actual count of transactions that entered no-dep mode and experienced inconsistent reads. The last column shows the total broadcast writes for the workload.

from dependences, with up to an order of magnitude improvement in performance. DATM effectively forwards the counter values between uncommitted transactions.

Dependence-related statistics are shown in Figure 4.10. Benchmarks that spend significant time in transactions commonly form dependences. For vacation, 35% of transactions form a forwarding dependence (W→R), and for labyrinth 32% of transactions form R→W dependences. The formation of dependences increases concurrency, reduces restarts and often improves performance.

DATM greatly reduces restarts relative to MetaTM, and the remaining restarts are classified in Figure 4.10. Restarts due to transactions overwriting forwarded data (forward restarts are rare (less than 1%) in the STAMP programs. They are a high percentage of aborts in TxLinux, but that is mostly due to there being few aborts in TxLinux. Cascaded aborts are generally responsible for single-digit percentages of restarts, which is low considering that about 40% of conflicts in both vacation and labyrinth involve more than two transactions (complex conflicts). While counter-tt has 100% cascaded aborts, the abort rate is 0.39% (from Figure 4.9). All of these aborts involve more than two transactions.

Finally, the number of inconsistent reads and transitions into no-dep mode are very low. While these mechanisms are necessary for correct operation, they are rarely needed. Also, the number of broadcast writes is less than 1/1000-th of one percent of transactional writes for all benchmarks. While broadcasting writes is necessary to preserve the ability to write back entire cache lines, it does not create excessive interconnect traffic.

4.5.3 Hardware constraints

Figure 4.11 shows the performance impact of various hardware constraints: cache line granularity, the ordering vector, and timestamp-ordered dependences (shown for bayes and vacation). By comparison to word-granularity implementations, managing dependences at cache line granularity reduces performance. Word-granularity requires extra state bits in the cache, but does not significantly increase bus traffic due to broadcast writes.

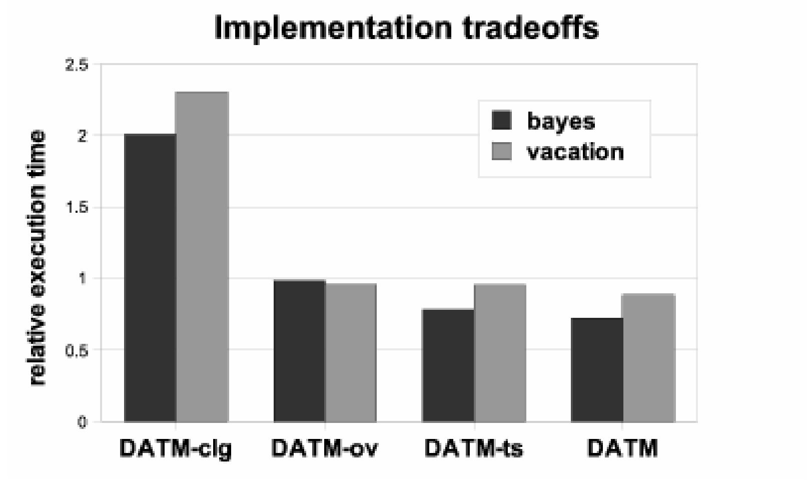


Figure 4.11: Relative execution time for various DATM hardware designs. DATM-clg uses cache-line granularity to manage dependences, DATM-ov constrains dependences according to the order vector, and DATM-ts constrains dependences with timestamp-ordered dependences. All relative execution times are normalized to MetaTM. Lower is better.

False cycles in the order vector reduce the performance of DATM. The average length of the order vector during transactions in bayes and vacation is approximately 6 (sampled at every dependence-causing memory operation) with maxima very close to the number of CPUs. Using timestamps to order dependences also reduces performance, but not as much as the order vector (e.g., bayes speedup goes from 39% to 14%).

4.5.4 Contention management

An attempt to create a dependence that would result in a cycle will cause DATM to invoke a contention management policy to resolve the conflict. DATM uses a novel dependence-aware contention management policy, which minimizes cascaded aborts by restarting the transaction with the fewest dependent transactions, resorting to timestamp when the number of dependents are equal. Figure 4.12 shows relative execution times for bayes and vacation using eruption, polka, and the dependence-aware

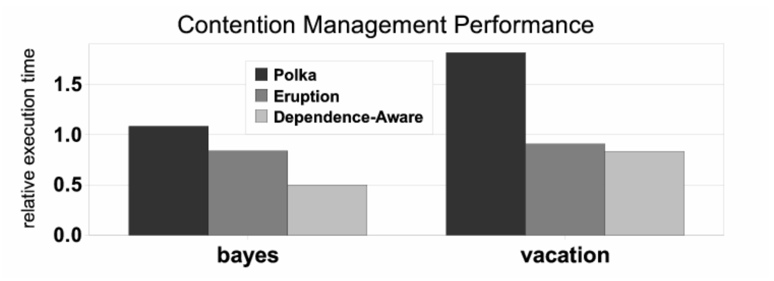


Figure 4.12: Impact of contention management policies in the presence of DATM. Performance is normalized to the MetaTM performance.

contention management policies. It outperforms non-dependence-aware contention managers (including timestamp, which is not shown).

4.5.5 Correctness of FRMSI

The case that the FRMSI protocol is correct relies on random stress testing, per-operation invariant-testing, rigorous use of asserts in the simulation modeling code. The techniques are described in much greater detail in section 5.7, where they are applied to another transactional protocol (**XMESI**), so this section provides only an overview.

FRMSI stress testing

An approach similar to that described by Wood et. al [93] was used to stress-test FRMSI. The CPU and interconnect-side cache controllers are exercised in simulation by selecting randomly from among a randomly chosen set of operations. All simulated CPUs run a program that continually, based on a psuedo-random number, starts, ends, pauses, and resumes transactions, along with generating random reads and writes. The program supports a mode in which transactions and pause/resume pairs are guaranteed to be well formed, and one in which they are not. All simulated modules which generate timing information (store buffers, all levels of cache, bus

local state	**legal remote states**	**legal higher-level states**
I	*	**I**
S	**I, S, TS**	**I, S**
M	**I**	**I, S, M**
TR	**I, T*, CTM**	**I, TR, S**
TS	**I, T*TM, S, CTM**	**I, TS, S**
CTM	**I, CTM, T***	**CTM**
TM	**I, T*TS, CTM**	**I, S, TS, TR, TM***
TMM	**I, T*, CTM**	**I, S, TS, TR, TM***
TMR	**I, T*, CTM**	**I, S, TS, TR, TM***
TMF	**I, T*, CTM**	**I, S, TMF**
TMRF	**I, T*, CTM**	**I, S, TMRF**

Table 4.2: Legal inclusion and sibling (local/remote) state pairs for FRMSI.

and main memory) randomly perturb stall times for operations across a wide range of cycle times to increase the stress on the simulated model.

FRMSI Invariant checking

While the stress test program runs, the simulator module implementing FRMSI caches runs in a mode that performs invariant checking after every memory operation or coherence event, as well as after instructions that affect the state of active transactions (`xbegin`, `xend`, `xpush`, `xpop`). The invariants checked include inclusion (traditional **MSI** inclusion as well as inclusion for **T*** states), along with invariants that describe legal sharing states across different tiers in the memory heirarchy (e.g. only one **M** or **TMM** copy, multiple **TM*** copies must all be **TMF**, **TMR**, or **TMRF**, and so on). Table 4.2 shows legal inclusion pairs and legal states for a block with same address cached in a sibling cache respectively.

Note that the inclusion properties do admit some combinations that are safe, but should not arise in practice. For example, it is safe for a copy of a cache line to be **TMR** in an L2 cache and in **TR** in an L1 cache because it must be in **TMF** or **TMRF** elsewhere. Either the remote copy is committed (making the local L2 the holder of the globally committed value), or aborts or overwrites (resulting in invalidation of both local copies). Both the **TR** and **TMR** copies require any local transactions to await the commit or abort of remote copies in the event that the local transaction occurs. However, while this combination is *safe*, it should not occur in practice because the L2 should only move to **TMR** in response to a local read, which would also move the the L1 to **TMR**. Note also, that unlike **XMESI**, **TMESI**, and other transaction aware protocols [58], FRMSI cannot run with MESI as a protocol for lower-level caches in a memory hierarchy.

In general, because FRMSI relies on the order vector (or timestamp-based ordering) to enforce a global order on versions for cached lines, any pair of speculative (**T***) states is legal. For example, a line may be cached in **TS** on CPU_0 and in **TMM** on CPU_1 as long as a $R_0 \rightarrow W_1$ dependence exists between transactions on those processors. The salient exception is the **TS**, **TM** pair which cannot occur because **TM** can only be entered from **M**. If the ***→TS** transition occurred logically before the ***→TM** transition, no **M** copy could exist, and writes on CPU_1 would transition straight to **TMM**. Conversely, if the ***→TM** transition occurred logically earlier, no **S** copy could exist, and reads on CPU_0 would move to **TR** rather than **TS**.

Comparison with an (assumed) correct MESI implementation

In the absence of transactional memory operations, FRMSI should behave exactly the same as MSI. By adding an **E** state to FRMSI, it was possible to perform the same cycle-wise comparison against teh Simics `g-cache` implemenation that is described later in section 5.7.3. The FRMSI implementation (with added **E** state) yields identical traces to `g-cache` up to 100,000,000 memory operations, after which the simulation terminates.

Use of asserts

The FRMSI transaction versioning C++ code relies on 51 asserts, and the cache model relies on 24 asserts during all simulations to check the correctness of state transitions and version management. Any simulation for which a failure occurs will cause the simulation to fail.

FRMSI stress-testings results

Bugs recently exposed in FRMSI by stress testing:

- A corner case for invalidations when a memory operation crossed a cache line boundary. When this case occurs, the cache needs fetch two lines, which it was correctly doing–however, the implementation of coherence traffic requires explicit "sends" to go to all the snooping caches, and the second send was using the wrong address, with a resulting invariant violation.

4.6 Conclusion

Dependence-aware transactions increase throughput by enabling concurrent execution of transactions that would otherwise conflict due to updating shared data structures. This paper presents the design, and a prototype implementation of the first dependence-aware hardware transactional memory system. Experimental results from our prototype confirms the potential performance benefits of dependence-aware transactional memory as compared to traditional HTM implementations. DATM eliminates the need for programmers to resort to esoteric programming patterns or to extend the TM programming model. This performance improvement is achieved through mechanisms that are completely transparent to the programmer.

Chapter 5

Avoiding and managing contention with TagTM

5.1 Introduction

The field of HTM research has become a diverse ecosystem of different designs and proposals [?, 2, 5, 7, 9, 10, 12, 13, 15, 18, 19, 22, 36, 39, 45, 46, 48, 58, 68, 70, 71, 85, 94, 96]. Despite this diversity, proposals to date have shared a unifying assumption that conflicts between transactions will be rare [1]. As a result, HTM designs favor tradeoffs that improve conflict-free performance at the expense of performance under contention. However, the ongoing lack of realistic TM workloads, combined with TM's envisaged role as a tool for the "average" programmer suggest this assumption may be tenuous.

This chapter argues that such an assumption endangers the success of TM because it admits designs that handle contention poorly. There is considerable evidence that contention is not necessarily rare. For example, Shriraman et al. find that as many as 90% of transactions conflict in some STAMP [54] benchmarks and up

[1]With notable exceptions [71, 84]

to 70% conflict in STMBench7 [28]. Work with TxLinux [70] showed that even OS synchronization, typified by very small critical sections (hundreds of instructions), can show moderate contention rates (10-20%) and occasional pathologically high contention [75]. In light of this, we that argue HTM implementations must perform well not only under low contention, but must degrade gracefully under contention. When conflicts do occur, HTM must manage them well by making contention management decisions quickly, and by supporting flexible policy so that decisions promote good performance. Additionally, HTMs should support techniques that help software avoid contention by eliminating the need to use heuristic approaches such as backoff to decide when conflicting transactions should retry. While it has been shown that backoff can reduce contention, and that exponential backoff in particular can avoid livelock [84], backoff can itself lead to performance pathologies [75] and is fundamentally heuristic.

This chapter presents TagTM, an HTM design that makes flexible contention management efficient in an HTM, and supports mechanisms to help software avoid conflicts before they occur. The techniques TagTM uses to improve performance under contention are applicable to any HTM design that relies on caches for version management and coherence for conflict detection. TagTM comprises two novel mechanisms to improve performance under contention: *notifying transactions*, and *transaction annotation*. *Notifying transactions* handle repeated transactional conflicts more effectively than backoff by making precise rather than heuristic decisions about when to retry. *Transaction annotation* helps TagTM provide user-defined contention management policies by allowing software to tag transactions with meta data that can be used by a simple hardware contention manager. Intelligent selection of meta data allows software and hardware to work in concert to implement flexible contention management policies without trapping to software or baking policy implementations into silicon. TagTM uses a transaction-aware coherence protocol called **XMESI** that extends transactional coherence to support powerful mechanisms such as transaction annotation and notifying transactions.

The rest of this chapter is organized as follows. Section 5.2 explores flexible hardware-based contention management, while section 5.3 explores TagTM's techniques for contention avoidance. Section 5.4 presents a coherence protocol and microarchitecture to support these features, which are evaluated alongside a cross-section of designs in section 5.5.

5.2 Transaction Annotation

A dynamic conflict occurs when two or more transactions access the same memory cell, and at least on of the accesses is a write. A conflict indicates that a non-serializable schedule has occurred: at least one of the transactions involved must abort and retry to preserve correctness. A contention manager (CM) is an abstraction that implements a policy or policies that decide which of the transactions must restart. The efficacy of CM policies is workload-dependent, and policies can have a first-order impact on performance [83]; as a result, support for flexible policy is a requirement for TM. HTMs can support contention management in software [48,86] or in hardware [22,58], and conventional wisdom has it that this flexibility must come at some complexity cost. Software implementations are arbitrarily flexible but require support from the hardware to trap to software handlers and communicate CM decisions back to the microarchitecture. Hardware implementations can be much simpler and lack flexibility, or must introduce even more complexity by implementing algorithms directly in silicon. Previously explored policies simple enough to be reasonable candidates for hardware implementation are likely to be inadequate. For example, the requester-wins [22,58] requires almost no additional hardware over the baseline HTM support, but is prone to livelock [13].

TagTM shows that flexibility and simplicity need not be at odds in this domain. TagTM provides the flexibility of software-based contention management, and does so without trapping to software, relying on only minimal hardware assists. As a result, TagTM reduces both hardware complexity and latency for contention management decisions.

5.2.1 Limitations of software conflict handlers

Contention management implementation for HTM is complicated by the tension between a need for user control over CM policy, and the fact that the mechanisms used to implement HTM abstraction are microarchitectural (e.g. caches, coherence protocols), and as such are typically not software-visible. Trapping to software handlers to implement CM introduces a number of design obstacles. It begs the question of where the handler should run: is the CM centralized or decentralized? If the CM is decentralized, which node runs the handler? Conflicts are detected at the coherence fabric by observing memory operations: if a contention manager is to have the freedom to decide in favor of either the requesting node or the detecting node, such conflicts must be resolved before the memory operation can complete. This requirement in turn implies that the results of software handlers run in interrupt context (and potentially involving user-kernel mode crossings) be available to coherence controllers, complicating coherence implementation, potentially introducing NACKs in the protocol, along with additional transient states.

The rationale behind software-based CM is that software knows its needs better than hardware, and can therefore implement better policy. In practice, this means that policies take into account dynamic state such as application data, and transaction meta data such as transaction age or the sizes of read-write sets [70, 84] to inform CM decisions. When transactions are supported using microarchitectural features such as caches and coherence, making such meta data visible to software requires additional support. In particular, making state available *on the node where* *a conflict handler runs* is a serious challenge: while it is easy to posit hardware assists that expose simple meta data through registers or ISA extensions, such hardware does not solve the problem of how such information is to become available at remote nodes. For example, the **Polka** CM policy [83] relies on restart histories and read-write set sizes. The node that detects the conflict in an HTM implementation can infer these data from microarchitectural state for the transaction running on that node, but cannot get this information for remote transactions without some interface for collecting this data and some mechanism to communicate with other nodes.

Additionally, some desirable policies may be very difficult to implement in software because they rely on access to application or OS data structures to arbitrate conflicts. This in turn requires synchronization, further complicated by the fact that conflict handlers run in interrupt context (which is a poor fit for many synchronization primitives). The os-prio [75] policy, which eliminates OS scheduler priority inversion by preferring the transaction whose thread has higher OS priority illustrates this problem well. Collecting priority values for conflicting transactions requires accessing oft-contended scheduler data structures. In the Linux kernel, collecting OS priority for all transactions involved in a conflict requires calling a function (`find_task_by_pid`) that requires the caller to hold a lock (`tasklist_lock`). The `tasklist_lock` cannot not be taken in interrupt context without risk of deadlock. Using transactions rather than locks to synchronize conflict handlers does not solve the issue and gives rise to the need to support nested conflict handling.

Finally, subtle race conditions arise in handler-based CM. A handler may execute concurrently with a transaction that is involved in the conflict being arbitrated. By the time a handler completes, conflicting transactions may have committed; more conflicting transactions can begin on remote processors. Finally, a CM handler needs some mechanism by which to communicate results. Communication through memory introduces the need for more synchronization. Interprocessor interrupts (IPIs) seem a likely candidate for communicating decisions between processor nodes, but it is common for spinlock-protected critical sections (e.g. in the Linux kernel) to mask interrupts, making IPIs a non-starter in some environments. It is worth observing that such race conditions are a challenge even for HTM designs that do not rely on cache-based version management. For example, TokenTM [12] attempts to decouple coherence from HTM implementation entirely by managing TM permissions using Tokens, which are written to a software-visible log in cachable virtual memory. While this does make transaction meta data effectively visible to software handlers, it does not solve the problem of how to coordinate decisions among local and remote transactions, and requires some form of synchronization to allow concurrent execution of local transactions and remote conflict handlers which may need to infer meta data from the logs.

5.2.2 TagTM: Implementing CM with annotation

We take the position that contention management decisions must be decentralized to avoid bottlenecks, and they must be flexible and under the control of software to avoid pathologies and livelock. However, for reasons detailed above, the policies themselves should not be run as software handlers. *Transaction annotation*, is a novel mechanism that allows software to *tag* transactions with data that can be used by a skeletal hardware-based contention manager. In transaction annotation, software controls policy by selecting tag values that encode the relative importance of individual transactions under some arbitration policy. Contention management decisions are made by the hardware based on the tag values selected, a decision that can be made in hardware at the node where conflicts are detected, and indeed by coherence controllers. TagTM implements transaction annotation with minor hardware assists (detailed in section 5.4), by including an additional register per node, extending cache-lines and augmenting coherence messages with payloads that include tag values. By allowing software to annotate transactions with tags to inform CM policies, TagTM provides the policy enable by software-based CM, but eliminates the complexity required to support handlers and allows CM decisions to made very quickly.

5.3 Avoiding conflicts with Notifying Transactions

TagTM uses a novel mechanism called *notifying transactions* to eliminate the need for contention-avoiding heuristics such as exponential backoff. Notifying transactions use the coherence fabric to communicate in much the same way TTAS (test-and-test-and-set) spinlocks [52] leverage coherence to reduce memory pressure caused by contended test-and-set locks. TTAS spinlocks effectively use the coherence fabric as a notification system: once a lock is held on a particular processor, any processors spinning on the lock variable can spin by repeatedly reading a shared copy of the lock's cache line without causing any additional traffic below the L1 cache. When a remote lock holder writes the lock variable to release it, it must request an exclusive copy: the subsequent GETX invalidates any shared copies of the line, and has the side-effect that waiters on the lock need not communicate with memory or execute coherence-visible CAS instructions repeatedly until exactly the moment the contended lock becomes available.

Currently, hardware transactions enjoy no analogous mechanism: if a transaction retries because of a conflict, no similar coherence traffic pattern exists to that allow aborted nodes to infer when a retry is likely to succeed or fail. This is a significant source of potential performance loss under contention. Traditionally, this problem is addressed with backoff policies, which can be effective [70, 84] but are fundamentally heuristic. A policy that is too aggressive puts pressure on the memory system, while a less aggressive policy can introduce needless serialization. Both cases result in lost performance. Moreover, because failed transactions must invalidate written cache lines, transaction retries incur cache misses and must interact with memory to rebuild read-write sets. The result is that contention in hardware transactional memory can have a much more severe impact in terms of pressure on memory bandwidth than highly contended locks.

Notifying transactions address this problem by leveraging modest extensions to transactional cache coherence protocols, which allows software to rely on support from the HTM hardware to make informed decisions. Notifying transactions retry after a conflict only when the retry is likely to be succeed, and need not initiate requests to memory while waiting to retry.

In TagTM, Notifying transactions are implemented with simple hardware support for remembering recent conflicts. For a transaction that wins a conflict, the hardware marks conflicting cache lines with a bit indicating a conflict has occurred. On commit or abort, coherence uses this bit to make coherence state transitions on those lines globally visible. For a transaction that loses a conflict, conflicting lines transition to a coherence state that indicates the lost conflict–this state behaves very much like a normal invalid state, with two exceptions. First, the ISA can query whether a given address is cached locally in that state, yielding a good indication of whether it was recently involved in a conflict. Second, lines in this state will transition to **I** in response to messages observed by the interconnect-side controller that match that line. Because a remote conflicter takes globally visible state transitions for conflicted lines on commit or abort, the resulting local transition acts as a notification that it is likely safe to retry a previously conflicted transaction.

Software examines a return code from the instruction the `xbegin` instruction (transaction begin) that indicates the reason for the abort , and provides the address of the conflicting memory location if a conflict caused the abort. Using an ISA extension (**xquery_tqc**) to query whether a conflicting cache line is still in a state that indicates a recent conflict, software can spin after a conflict until the conflicting line takes a state transition triggered by the commit or abort of the remote conflicting transaction, as shown in the code in figure 5.4.4. In this way, transactions can avoid heuristic approaches like backoff to determine when to retry, and can dramatically reduce pressure on memory when critical sections or data shared in transactions are highly contended. Reduced memory pressure can translate directly to improved performance in a bandwidth-constrained environment.

5.4 Design

The TagTM design relies on simple extension to microarchitecture, an additional ISA instruction, and a new cache coherence protocols called **XMESI**.

5.4.1 XMESI

The **XMESI** protocol augments cache lines and transactional messages with software-visible state called **TxTAG**, which is the fundamental building block TagTM relies on to make contention management decisions locally at cache controllers. Five stable states are added over the traditional MESI states: three support tracking of read-write sets and buffering of speculative writes, while two support inference that lines have been recently involved in transactional conflicts. It should be noted that at least two of the first three states are required in any HTM design that uses caches for version management and conflict detection. Figure 5.1 depicts the state transition diagram for stable states in **XMESI**; table 5.2 shows state encoding for, and table 5.1 details actions taken in **XMESI** to handle local commits, aborts, and conflicts detected locally and remotely.

The standard **M**, **E** (not shown in figure 5.1 but included in the protocol), **S**, and **I** states behave as they would in a normal MESI protocol. The **TS** state represents transactionally read cache lines, while **TMU** and **TMM** states represent lines previously in **M** subsequently read and modified in a transaction respectively.

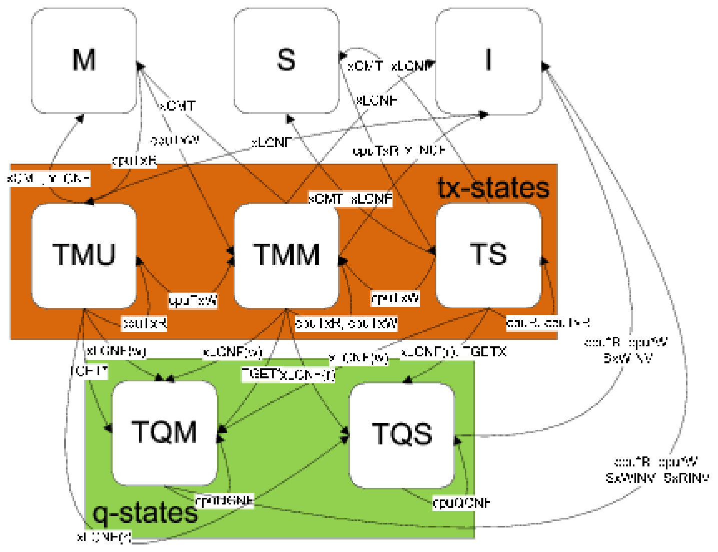


Figure 5.1: The XMESI protocol. State transitions for standard **MSI** states are elided for clarity. The **TMU** state indicates a line that was previously in **M**, and has been read but not written in a transaction. The value is dirty with respect to main memory, but is the current globally committed value. The **TMM** state indicates a line that has been written in a transaction, and is therefore both speculative and modified with respect to main memory. The **TS** states represent a non-dirty line (**E** or **S**) lines that have been read in a transaction. The **TQM** state is represents a line that was read or written in a local transaction, and conflicted with a remote writer, while **TQS** indicates a conflict with a remote reader. All edges labeled **cpuTxR, cpuTxW** indicate memory operations from the processor and are parameterized with a transaction identifier (TXID) and **TxTAG**. Edges labeled with **xCMT** indicate a transaction commit event. Edges labeled **xLCNF(r)** and **xLCNF(w)** indicate a remote conflict response to a locally initiated read or write respectively, and indicate that the local transaction has lost arbitration and must abort. Edges labeled **SxWINV** and **SxRINV** indicate transitions triggered by the commit or abort of a remote transaction that conflicted with the local one.

The **TMM** and **TS** states or their analogues are necessary in any cache-based HTM versioning implementation and the **TMU** state is an optimization analogous to the **TM** state described by Shriraman et al. [87]. Taken collectively, the **TS**, **TMU**, and **TMM** do not have semantics that differ significantly from previously proposed designs [71, 87].

The **TQS** and **TQM** states indicate lines cached that were referenced in a transaction that lost a conflict with a another remote transaction: **TQS** indicates that the remote transaction was a reader of the cached memory location, while

msg/act	*TMM, TMU*	*TS*	*TQM*	*TQS*
GETX	asym. conf:T*→I	asym. conf:T*→I	TQM→I	TQS→I
GETS	asym. conf:T*→I	–	–	–
TxGETX	Win:T*→T*xLCNF Lose:T*→TQM	Win:T*→T*+xLCNF Lose:T*→TQM	–	–
TxGETS	Win:T*→T*+xLCNF Lose:T*→TQS	–	–	–
SxRINV	X	X	TQM→I	–
SxWINV	X	X	TQM→I	TQS→I
xLCNF(R)	T*→TQS	T*→TQS		
xLCNF(W)	T*→TQM	T*→TQM		
xCMT(xC set)	T*→M/SxWINV	TS→S/SxRINV	X	X
xABT(xC set)	TMU: TMU→M/SxWINV TMM: TMM→I/SxWINV	TS→S/SxRINV	X	X

Table 5.1: XMESI state transition triggered by bus/interconnect-side memory traffic, local commit or aborts, and remotely detected conflicts. Remote commits and aborts are made visible with compulsory **SxWINV** and **SxRINV** messages for conflicted **TM*** and **TS** lines respectively.

TQM indicates that the remote transaction was a writer. The states are used to support an ISA extension called **xquery_tqc** (described in section 5.4.2, and shown in table 5.3) that returns a 1 or 0 indicating whether the given address was recently involved in a transactional conflict; an assessment that is well approximated by whether or not the address is cached locally in **TQ***. In all other respects, **TQ*** states behave like MESI invalid states: reads and writes to **TQ*** lines are cache misses. A simpler design could condense the **TQ*** states into a single state at some cost in precision in making retry decisions.

The **XMESI** protocol supports transactional versions of GETX and GETS (get exclusive and get shared) requests called **TxGETX** and **TxGETS** [87]; **XMESI** parameterizes these requests with both a transaction identifier **TXID**, and a small software supplied meta data value called **TxTAG** a value set as part of the **xbegin** instruction. Conflicts are always arbitrated at the node where they are detected, usually in reaction to snooped memory traffic (however, conflicts can occur between multiple local transactions as well). **XMESI** also supports reply messages that inform a requesting transaction that the request caused a conflict, and the contention

state	*m*	*v*		*c*
M	1	1	0	0
E	1	0	1	0
S	0	1	0	0
I	0	0	0	0
TMM	1	0	0	0
TMU	1	1	1	0
TS	0	1	1	0
TQM	1	0	1	1
TQS	0	0	1	1

Table 5.2: **XMESI** state encoding.

management policy invoked to handle the conflict decided that the requester would lose the conflict. To this end, the "lost conflict" messages **xLCNF(r)** and **xLCNF(w)** indicate that the winning transaction was a reader or a writer of the conflicting address respectively. Nodes receiving **LCNF** messages must abort the current transaction, and transition any locally cached copies of the conflicted line to **TQS** in response to **xLCNF(r)** and **TQM** in response to **xLCNF(w)**.

When contention management decides in favor of a transaction, **XMESI** marks the cache lines involved with a single bit (**xC**) that indicates the line was involved in a conflict. When the winning transaction subsequently commits or aborts, cache lines with this bit set require that hint invalidations messages, called **SxRINV** (for lines in bf TS) and **SxWINV** (for lines in **TMU or TMM**) be sent as part of the commit or abort. Remotely cached lines in **TQS** or **TQM** state respond to these messages by moving to the invalid state, with the side effect that **xquery_tqc** instructions executed for those lines will no longer return 1.

5.4.2 Micro-architecture and ISA extensions

Additions to microarchitecture and ISA necessary to TagTM are shown in figure 5.2 and table 5.3. ISA support includes the compulsory instructions to begin, end, retry, or check for the existence of a transaction, as well as a new instruction, **xquery_tqc**,

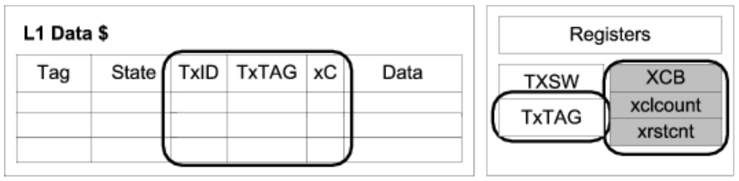


Figure 5.2: Microarchitectural features required to support TagTM. In addition to a transaction identifier **TXID**, L1 cache lines are augmented to include a **TxTAG** field and a single **xC** bit which, when set, indicates that the line being cached was involved in a transactional conflict that the local transaction won. The processor core is extended to include a **TxTAG** register as well. The additions shown in gray (**XCB**) is a transaction control block [48]: comprised of registers that make meta data about transactions visible to software. The **xclcount** indicates the number of lines in T* states (a heuristic for transaction size), the **xrstcnt** stores the number of times the current transaction has restarted.

which returns and indication of whether a given address was recently referenced in a local transaction that lost a conflict by checking whether the line is cached in **TQ***.

Processor nodes are extended to include a **TxTAG** register, set by the **xbegin** instruction, along with transaction status word **TXSW**. L1 data cache lines are extended to support a transaction identifier (**TXID**) [48, 70]); we additionally include the software-visible **TxTAG** field, and an **xC** bit. The **TXID** and **TxTAG** fields set by hardware to match the value in the **TXSW** and **TxTAG** processor registers when a cache line is referenced in a current transaction. The **xC** bit is set by hardware in reaction to conflicting memory traffic seen by the cache controller. If the line is involved in a transactional conflict, and the local transaction is selected as the winner, the **xC** bit is set.

Note that support for **TQ*** states in an SMP-like organization requires minor changes to lower level caches to ensure that coherence events for lines an upper level cache holds in **TQ*** are visible to that cache. This can be accomplished by allocating a single bit on cache lines in lower level caches that indicates that a line in **I** in a lower level cache was recently involved in a transactional conflict. This allows interconnect-side controllers to continue to observe and propagate coherence traffic for such lines, but note that lower level caches are free to treat the line as

Primitive	Definition
xbegin	Instruction to begin a transaction. Accepts a parameter **TxTAG** allowing software to label transactions with data that can be used by contention management algorithms. Returns a code indicating the reason for retry and updates a status register to hold the address of the conflicting memory location in the event of a conflict.
xend	Instruction to commit a transaction.
xrestart	Instruction to restart a transaction
xgettxid	Instruction to get the current transaction identifier, which is 0 if there is no currently active transaction.
xquery_tqc	Instruction to query whether an address was involved in a recent transactional conflict. Implemented as a query for whether the given line is cached locally in **TQM** or **TQS** state.

Table 5.3: ISA support for TagTM.

invalid in all respects, including reallocating the line to cache a different block of memory. When this occurs the lower level cache must (conservatively) invalidate copies in upper level caches.

5.4.3 Transaction Annotation

TagTM orchestrates the mechanisms described in the previous section to render contention management decisions locally at cache controllers at the time conflicts are detected. All memory traffic is labeled with both TXIDs and the **TxTAG** values (extending message payloads), and contention management is implemented by preferring the greater of the two **TxTAG** values, defaulting to preferring the greater TXID in the event of a tie.

This mechanism enables dynamic selection of policy in TagTM. Because software selects **TxTAG** values when using the `xbegin` instruction, these values can be chosen in such a way as to ensure that CM decisions are made according to the desired policy. For example, the **os_prio** policy [75] can be implemented by using the value of the Linux `effective_prio` function as the **TxTAG** value. The **polite** [83] policy can be implemented by parameterize `xbegin` with a restart counter. The transaction annotation is sufficient to support controller-based implementation of most policies described in the literature.

Contention management algorithms built using this mechanism are safe because the TXIDs and CPU identifier guarantee a total order. Software is responsible for ensuring that policies are livelock free, although this is not a significant burden in this context: any policy that avoids choosing a new **TxTAG** value on restart is guaranteed to be livelock-free.

5.4.4 Implementing notifying transactions using XMESI

TagTM supports notifying transactions with **XMESI** support as follows. The **xC** bit controls whether state transitions on a cache line must be globally visible on commit or abort, and additional coherence states **TQM** and **TQS** indicate whether a cached line was referenced in a transaction that lost a conflict with a remote transaction: **TQM** indicates that the remote transaction was a writer of the line involved, while **TQS** indicates the remote transaction was a reader. The **query_tqc** instruction allows software to interrogate the L1 data cache, and determine whether a given address is cached locally in **TQM** or **TQS**[2] When a local transaction loses a conflict, the cache line for the conflicting address transitions to the appropriate **TQ*** state. Wen the transaction restarts, the **xbegin** instruction will indicate that a conflict occurred, and the address of the contended memory

```
notifying_tx(int label) {
 txretry:
   if(CONFLICT(xbegin(label))){
     xend();
     int* v = conflict_addr(TXSW);
     while(xquery_tqc(v));
     goto txretry;
   }
   // work on data structure
   xend();
}
```

Figure 5.3: Pseudo-code using notifying transactions for a critical section. If a conflict is detected, software ends the transaction, collects the address of the conflicting line and spins until the **xquery_tqc** instruction returns false, which will occur when the cache line for that address is no longer cached in **TQM** or **TQS** state.

[2]A single **TQ** state could represent both **TQS** and **TQM**, with an associated loss of precision associated with predictions on when to retry.

location is available in the **TXSW** register. Software then uses this address and the `xquery_tqc` instruction to spin until the conflicting line is no longer cached locally in **TQ***. When the remote (winning) transaction commits or aborts, it's cached copies of conflicting lines will have the **xC** bit set, allowing it to force a globally visible transition for that line: for lines cached in **TMM** or **TMU** states, a **SxWINV** message is sent, while lines in **TS** result in a **SxRINV**. Differentiating between readers and writers when making this state transition enables receivers of the message to wait until actually read-write and write-write conflicts resolve. Using s single message in this case would preserve correctness, but make it possible for read-sharing between non-conflicting transactions to cause false positives for transactions waiting for a remote conflicter to commit or abort.

Note that the **TQ*** states behave like **I** state in all other respects. Reads and writes to the line (transactional or non-transactional) require a fresh copy of the line from memory, and the line is not invalidated by remote reads or writes. Other potentially unrelated events such as associativity evictions can cause a transition out of a **TQ*** state that might induce software to retry before the remote conflicter completes as well. Because the mechanism is fundamentally used as a hint for when retries may be profitable, correctness is not preserved for these scenarios. Because TagTM is a "best-effort" design, if an annotated line is evicted, the transaction will be forced to restart and annotations will be dropped.

It should also be noted that because software dictates policy for when to spin for remote conflicters, the hardware can still function as an HTM with traditional traditional retry-on-conflict semantics. The presence of these mechanisms is does not constitute a requirement for software to use them.

5.5 Evaluation

To evaluate TagTM, MetaTM [70, 75] was extended to implement **XMESI**, notifying transactions, and transaction annotation. TagTM runs in the Simics machine simulator [47] (all experiments were run with version 3.0.31). The `xquery_tqc` instruction is implemented as a new machine instruction (rather than a magic instruction) and the **XMESI** protocol in implemented as a separate Simics module. We simulated both CMP and SMP organizations using 16 and 32 cpus, and ran the TxLinux kernel benchmarks [70] as well as a subset of the STAMP benchmark suite [54]. Machine parameters and benchmark command lines are detailed in table 5.4. TagTM and MetaTM are both "best effort" designs, which means that the burden of handling transactions that fail because of associativity or capacity evictions of cache lines in **T*** states falls on software. TagTM and MetaTM both handle this case by extending cxspinlocks [75] to handle overflow of HTM resources in exactly the same way I/O is handled: by rolling back the transaction and arbitrating acquisition of a lock through the contention manager. The approach is similar to that used in [39, 56, 66, 67, 75]. The STAMP benchmarks were modified to use a single global cxspinlock to allow reexecution of overflowed transactions under mutual exclusion.

5.5.1 Evaluating notifying transactions

To evaluate notifying transactions, we compare raw performance, restart rates, and bandwidth consumption of TagTM using notifying transactions vs. MetaTM's default configuration on 16 and 32 cpu CMP and SMP machines. MetaTM uses a linear backoff policy, and implements contention management policies within the simulator, making this a comparison which will highlight behaviors induced by notifying transactions.

Because many of the STAMP benchmarks have critical sections that cause

	Configuration
Processor	Pentium-4-like x86 instruction set, 3 GHz, 1 IPC
Store buffer	32 entries, 1 cycle access
L1	separate i+d, 32 KB, 4-way, 64-byte line, lru replacement, 3-cycle access
L2	unified 4 MB 8-way, 64-byte line, 12-cycle access
Memory	1GB capacity, 200 cycle access time, 12GB/sec bandwidth
Memory Hierarchy	SMP: Each core has a private L2. CMP: all cores share a single L2.
TM parameters	Timestamp contention management, linear backoff policy, cache line granularity conflict detection.
Overflow strategy	transactional lock elision

Benchmark	*Description*
config, pmake, mab	These benchmarks report transactions created in TxLinux (see 3) a Linux-variant operating system which relies on transactions rather than spinlocks for most polling synchronization [70, 75]. transactions for synchronization, instead of spin-locks The workloads involve user-mode applications (configure, make, modified andrew benchmark) which are running on the transactional OS.
bayes	learns the structure of a Bayesian network "-v32 -r4096 -n5 -p30 -s1 -i2 -e4 -t CPUs"
genome	a gene-sequencing bioinformatics application, "-g16384 -s32 -n4194304 -t CPUs"
intruder	signature-based intrusion detection system "-a10 -l32 -n65216 -s1 -t CPUs"
kmeans	implements a K-means clustering algorithm, " -m40 -n40 -t0.0009 -i random-n65536-d32-c16.txt -p CPUs"
vacation	models a multi-user database, "-c CPUs -n4 -q60 -r1048576 -u90 -t1048576"

Table 5.4: The upper table represents architectural parameters of simulated machines. TM parameters were held constant across all HTM designs. The lower table shows workloads used in evaluation. TxLinux and list are kernel-mode transactions, while the other benchmarks run in user-mode. All benchmarks use a number of threads equal to the number of processors, unless noted otherwise.

overflow in an L1-based HTM, the cost of rolling back and serializing with a single global lock can become a dominant cost for executing those benchmarks. To minimize noise in the data stemming from this effect, we modified the simulation to continue executing transactions in the presence of overflow: in simulation, version management is handled separately from memory hierarchy timing, making it possible to implement a mode that provides unbounded transactions with bounded cache size. Because latency for cache misses taken on transactional lines that have overflowed in this mode will still effect overall execution latency, it is impossible to com-

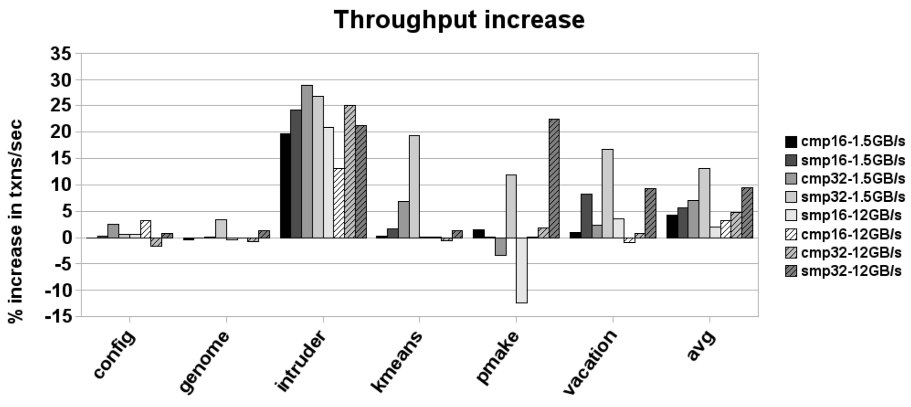


Figure 5.4: Notifying transactions throughput increase over MetaTM for all configurations.

pletely normalize out the effects of overflow: however, since MetaTM and TagTM use the same cache geometry, such effects should impact both designs equally.

It should also be noted that using this technique to remove overflow as dominant factor has the side effect of ensuring that critical sections waiting for notification using the `xquery_tqc` instruction will never retry too early due to eviction or reallocation of lines in **TQ*** state–retry will occur only when a remote conflicter actually commits or aborts, or when coherence traffic for which an abort can be inferred occur. Consequently, for workloads that have significant overflow rates, this methodology will tend to inflate the potential benefit notification can bring: however, it is arguable that "best effort" HTM is a poor fit in general for such workloads. For workloads that have minimal overflow, such as intruder, these data represent a realistic picture of the potential benefit of notification.

All benchmarks in this study run on top of TxLinux, and the behavior of the OS is modeled: TxLinux also uses transactions for it's own synchronization, (often executing many more transactions than the workload itself). TxLinux executes a significant fraction of it's transactions in interrupt context: the side-effect is that some interplay between kernel mode and user mode transactions can take place:

typically in the form of overflows caused in user mode transactions by associativity evictions of **T*** lines for user transactions occurring during interrupt handlers. While the effect is observable in the form of higher overflow numbers for user-mode benchmarks that have higher system time, we do not evaluate the impact of this interplay in detail.

We evaluate both speedup and increase in transaction throughput for all benchmark. In general, the later is the more appropriate metric because many benchmarks can have variable numbers of transactions during the life of the program. The TxLinux benchmarks execute a large fraction of their transactions in interrupt context, and variability is unavoidable in this case. The `bayes` and `kmeans` benchmarks from STAMP also show considerable variability because the number of transactions executed by these workloads is non-deterministic. In cases where benchmarks have consistent numbers of transactions, throughput and speedup are essentially the same information, whereas benchmarks that show variable transaction numbers introduce noise into the speedup numbers because of the difficulty attaining samples whose execution time can be meaningfully compared.

Tables 5.5 and 5.6 show statistics for experiments for benchmarks running on 16 cpu and 32 cpu CMP and SMP machine models, with bandwidth to main memory limited to 12GB/sec to model a reasonable modern machine, and 1.5GB/sec to model a bandwidth-constrained environment. Figure 5.4 shows the percentage increase in throughput (transactions/second), and figure 5.6 shows the raw speedup of TagTM over MetaTM. In general, the data show that notifying transactions can improve performance by reducing the costs of repeated restarts in most cases, and can occasionally harm performance as well. The geometric mean speedup is between 1.05× and 1.18× for the bandwidth constrained machines, and between 1× and 1.16× for machines with more reasonable memory bandwidth budgets.

Figure 5.5 shows the fraction of restarts eliminated by TagTM relative to

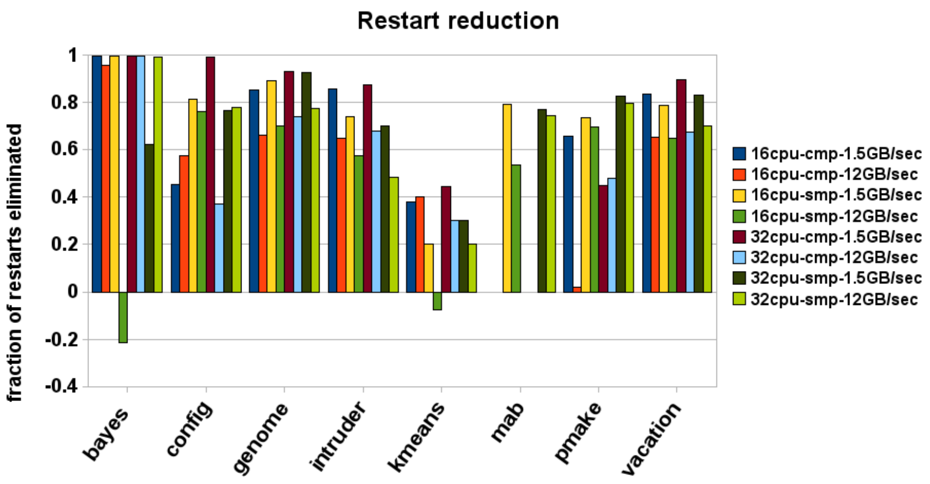


Figure 5.5: Fraction of restarts in MetaTM eliminated in TagTM by notifying transactions for 16 and 32 cpus, CMP and SMP organizations, with memory bandwidth limited to 1.5GB/sec and 12GB/sec.

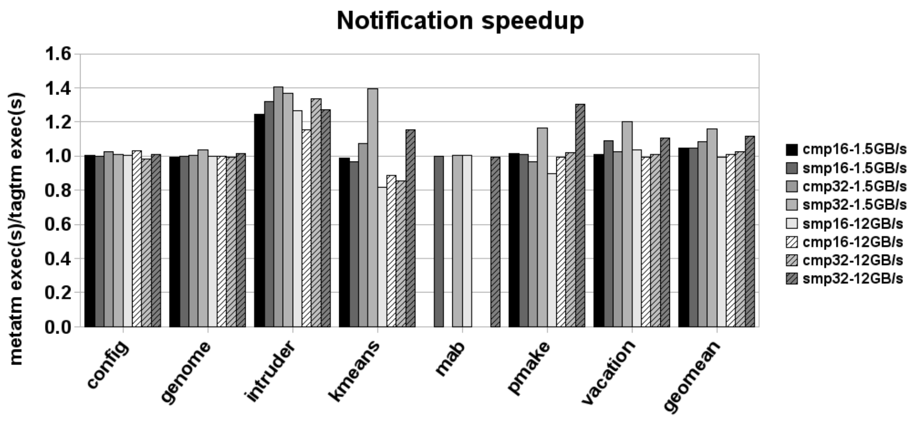


Figure 5.6: Notifying transactions speedup for all machine configurations.

MetaTM. Notification dramatically reduces restart rates for all but a handful of cases, showing that the mechanism is very effective at delivering on its promise of helping software to avoid contention. For many benchmarks such as `bayes` and `intruder`, notification eliminates the vast majority of restarts. However, two samples contradict the trend: `kmeans` and `bayes` both show increased restart rates on the 16 cpu SMP with 12GB/s memory bandwidth. Both of these benchmarks show high variability in both transaction count and transaction size: in these cases the increase in transaction restart rate shows a change in the profile of transactions being executed in the benchmark, rather than the inability of TagTM to help manage contention. These data points represent runs of the benchmark that differed by more than 10% in the number of transactions, and for `bayes`, a concomitant variation over 10% in the average read-write set size. The degree to which avoiding contention translates to improvements in performance varies with level contention and the percentage of time workloads spend in transactions. Intruder, which spends approximately 30% of its execution time in transactions and has 30-50% conflict rates can see a dramatic change from reduced restarts, while removing 100% of the restarts from `mab`, which has 1% restart rate, and spends very little time in transactions will have a neglible effect.

Notification is a mechanism that impacts workloads most in the presence of contention. Benchmarks that have significant rates of transaction restarts due to conflict, such as `intruder` and `vacation` stand to benefit the most. The `intruder` benchmark is characterized by many small transactions and high contention, which is precise the type of workload that notification should be able help. 30% to 50% of transactions restart in `intruder` and higher core counts correlate with much higher contention. Intruder benefits significantly from notifying transactions, showing speedups between 1.2× and 1.4× for all machines, and deriving the biggest speedups for bandwidth constrained machines, where spinning on a cache line in **TQ*** allows it to reduce pressure on the memory system. Throughput increases 19% to 28% on machines with a 1.5GB/sec bandwidth limit, and from 13% to 25% for machines with a 12GB/sec limit. Similarly, `vacation` has restart rates in the 30% range: TagTM increases transaction throughput for `vacation` on all bandwidth-constrained machines, with a high watermark of 17% on 32 cpu SMP.

bnc	**bw**	**htm**	**exec**		**pctrst**		**bkcyc/nftx**		**ovpct**	
bayes	12	metatm	0.024	0.029	11	15	56210/–	60081/–	27	25
(∼1534)		tagtm	0.060	0.056	14	12	–/229	–/184	28	26
	1.5	metatm	0.056	0.048	11	16	190198/–	103267/–	27	28
		tagtm	0.027	0.045	9	10	–/108	–/149	24	24
config	12	metatm	0.547	0.611	1	1	2/–	6/–	0	0
(∼4682761)		tagtm	0.530	0.607	1	1	–/11830	–/20942	0	0
	1.5	metatm	3.298	3.689	1	2	2/–	18/–	0	0
		tagtm	3.291	3.686	1	2	–/6862	–/29389	0	0
genome	12	metatm	0.339	0.348	0	1	1/–	1/–	33	33
(1180046)		tagtm	0.339	0.350	0	1	–/4755	–/5482	33	33
	1.5	metatm	0.748	0.785	0	0	1/–	4/–	33	33
		tagtm	0.752	0.785	0	0	–/1759	–/3748	33	33
intruder	12	metatm	0.246	0.460	30	50	220/–	383/–	7	12
(1587020)		tagtm	0.214	0.364	29	51	–/381527	–/671216	7	6
	1.5	metatm	1.073	2.428	30	50	953/–	1727/–	12	10
		tagtm	0.862	1.840	29	50	–/323182	–/655812	9	7
kmeans	12	metatm	1.464	1.557	17	7	22/–	23/–	0	0
(∼5232896)		tagtm	1.649	1.901	18	7	–/565494	–/174279	0	0
	1.5	metatm	2.320	2.961	13	15	617/–	610/–	0	0
		tagtm	2.352	3.068	13	15	–/423853	–/483219	0	0
mab	12	metatm	0.231	0.350	1	1	5/–	4/–	0	0
(∼4133451)		tagtm	0.223	0.349	1	1	–/12707	–/26118	0	0
	1.5	metatm	1.361	2.111	1	2	9/–	9/–	0	0
		tagtm	1.339	2.114	1	2	–/7779	–/40540	0	0
pmake	12	metatm	0.021	0.020	2	3	2/–	26/–	0	0
(∼193473)		tagtm	0.021	0.022	2	3	–/1186	–/1566	0	0
	1.5	metatm	0.105	0.112	3	4	8/–	71/–	0	0
		tagtm	0.104	0.111	3	4	–/1115	–/3644	0	0
vacation	12	metatm	0.444	0.494	34	34	98/–	97/–	26	25
(1048576)		tagtm	0.448	0.476	34	35	–/321611	–/321917	24	24
	1.5	metatm	2.098	2.415	28	29	317/–	255/–	31	23
		tagtm	2.080	2.217	27	30	–/152905	–/206988	23	29

Table 5.5: Statistics for Notifying transactions, 16 cpu CMP and SMP. Data columns have two entries per cell: the leftmost figure corresponds to a 16 cpu CMP machine, while the rightmost figure is for a 16 cpu SMP machine. The table shows the benchmark in the **bnc** column with the number of transactions in parenthesis. (Some benchmarks have variability in the number transactions, indicated by the use of “ ”). The **bw** column indicates the upper bound on memory bandwidth used in the experiment, in GB/sec. The **exec** column is the user time for STAMP benchmarks and kernel time for TxLinux benchmarks. The **pctrst** and **bkcyc** columns are the percentage of transactions that restarted at least once. The **bkcyc/nftx** column indicates the average backoff cycles per transaction (blue) and the number of transactions that used notification at least once (magenta): because TagTM does not use backoff, the two numbers are mutually exclusive. The **ov** columns shows the percentage of transactions that overflow the transaction support in the L1 cache.

The `vacation` benchmark also has large transactions, which lead to high rates of overflow (in the 25%-30% range). If these experiments were to actually abort overflowed transactions (rolling back and grabbing a lock is the overflow strategy in both MetaTM and TagTM), the benefits of notification would be masked by the detrimental performance impact of having to serialize nearly a third of the transactions in the benchmark.

The `bayes` benchmark implements an algorithm for learning Bayesian networks where probability distributions and conditional dependences among them are learned from observed data using a hill-climbing strategy: the running time depends on the order in which edges are added to a graph, with the result that the same inputs can generate not only highly variable running times, but highly variable numbers of transactions. The `kmeans` benchmark, a K-means clustering algorithm, also can generate different clusters under different execution interleavings, resulting in variability both in run-time and numbers of transactions. However, `kmeans` does have significant contention and no overflow. While the speedup numbers appear to show a loss in performance for `kmeans` this is a case where runs that have speedup less than 1 execute as much as 30% more transactions. The transaction throughput, on the other hand, shows that `kmeans` benefits significantly from notification on bandwidth-constrained machines, and has no significant loss on the 12GB/sec bandwidth machines. This correlates with higher contention: restart rates for `kmeans` ranged from 13% to 25% on the bandwidth constrained machines, and 7% to 18% restarts otherwise.

Benchmarks with medium to no contention such as `genome`, and the TxLinux kernel benchmarks `config`, and `mab` see (not surprisingly) very little impact on performance. `pmake` has the highest contention rate (2-5% restart rate) and derives some benefit from notification, particularly on SMP 32 cpu machines where throughput is increased 12% and 23% for 1.5GB/sec and 12GB/sec bandwidth lim-

bnc	**bw**	**htm**	**exec**		**pctrst**		**bkcyc/nftx**		**ovpct**	
bayes	12	metatm	0.022	0.015	11	17	105531/–	45716/–	24	22
(∼1394)		tagtm	0.021	0.013	10	14	–/118	–/194	25	25
	1.5	metatm	0.024	0.021	16	17	266178/–	152482/–	23	24
		tagtm	0.025	0.029	12	15	–/147	–/210	25	25
config	12	metatm	0.403	0.448	1	2	4/–	11/–	0	0
(∼5117933)		tagtm	0.411	0.445	1	2	–/19873	–/28541	0	0
	1.5	metatm	2.503	2.971	2	2	5/–	11/–	0	0
		tagtm	2.443	2.950	2	2	–/12515	–/42083	0	0
genome	12	metatm	0.206	0.224	1	1	2/–	5/–	25	21
(1180046)		tagtm	0.208	0.222	1	1	–/6456	–/9131	21	25
	1.5	metatm	0.708	0.836	0	1	10/–	31/–	27	27
		tagtm	0.707	0.807	0	1	–/3018	–/7296	25	25
intruder	12	metatm	0.341	0.853	37	52	822/–	2015/–	2	1
(1587036)		tagtm	0.255	0.671	37	53	–/488817	–/720629	2	2
	1.5	metatm	1.486	5.059	38	50	1382/–	566/–	1	2
		tagtm	1.056	3.702	38	50	–/441259	–/667803	1	2
kmeans	12	metatm	0.731	0.986	17	11	24/–	52/–	0	0
(∼5517946)		tagtm	0.855	0.856	17	11	–/545345	–/287980	0	0
	1.5	metatm	2.364	5.113	25	45	694/–	302/–	0	0
		tagtm	2.200	3.662	23	43	–/980215	–/1792793	0	0
mab	12	metatm	0.141	0.216	1	1	6/–	5/–	0	0
(∼4298303)		tagtm	0.139	0.217	1	1	–/26761	–/31185	0	0
	1.5	metatm	0.871	1.264	2	2	7/–	8/–	0	0
		tagtm	0.867	1.260	2	2	–/42836	–/47014	0	0
pmake	12	metatm	0.016	0.016	2	3	6/–	115/–	0	0
(∼235108)		tagtm	0.015	0.013	2	3	–/1323	–/2427	0	0
	1.5	metatm	0.080	0.120	2	5	5/–	451/–	0	0
		tagtm	0.083	0.103	2	5	–/1604	–/6407	0	0
vacation	12	metatm	0.439	0.551	48	49	247/–	293/–	24	25
(1048576)		tagtm	0.435	0.500	47	48	–/376678	–/401434	24	24
	1.5	metatm	2.202	2.910	42	44	1423/–	872/–	29	27
		tagtm	2.148	2.424	39	44	–/189500	–/302816	25	23

Table 5.6: Statistics for Notifying transactions, 32 cpus CMP and SMP. Data columns have two entries per cell: the leftmost figure corresponds to a 32 cpu CMP machine, while the rightmost figure is for a 32 cpu SMP machine. The table shows the benchmark in the **bnc** column with the number of transactions in parenthesis. (Some benchmarks have variability in the number transactions, indicated by the use of “ ”). The **bw** column indicates the upper bound on memory bandwidth used in the experiment, in GB/sec. The **exec** column is the user time for STAMP benchmarks and kernel time for TxLinux benchmarks. The **pctrst** and **bkcyc** columns are the percentage of transactions that restarted at least once. The **bkcyc/nftx** column indicates the average backoff cycles per transaction (blue) and the number of transactions that used notification at least once (magenta): because TagTM does not use backoff, the two numbers are mutually exclusive. The **ov** columns shows the percentage of transactions that overflow the transaction support in the L1 cache.

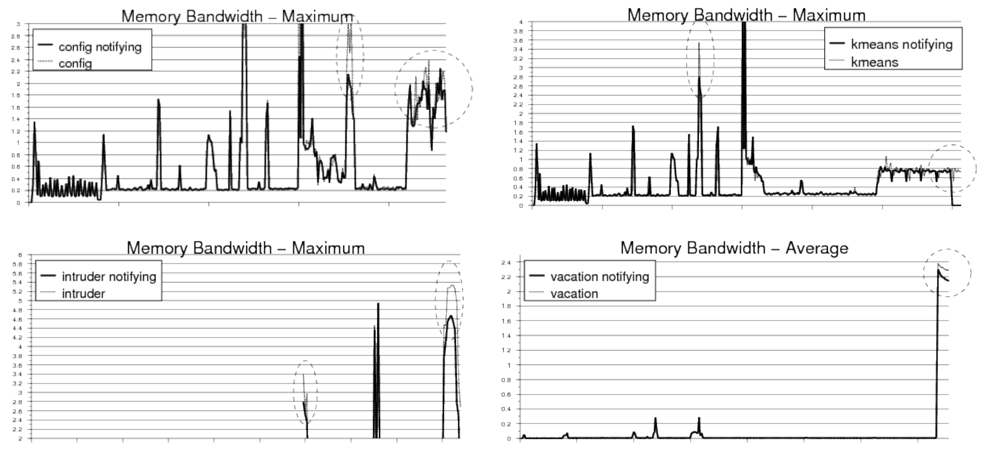


Figure 5.7: Time series profiles of peak or average transaction-related bandwidth usage of representative benchmarks on a simulated 16 CPU SMP machine. Regions where bandwidth usage is high and notifying transactions demonstrate significant bandwidth savings are circled. For many benchmarks, savings are insignificant when bandwidth usage is low but increase significantly when usage goes up. Units are in GB/sec bandwidth is not artificially limited.

its respectively. However, `pmake` is also the only benchmark in which TagTM has a significant negative impact on performance. In no other case does notifying transactions have a significant negative impact on performance: in general, the mechanism helps when contention is high, and does not perturb the system otherwise.

Bandwidth consumption in notifying transactions

Notifying transactions can affect performance through channels other than work wasted due to transaction restarts. Because notifying transactions selectively stall transactions and reduce the number of restarts, notifying transactions can reduce bandwidth pressure on the memory system. Figure 5.7 compares bandwidth consumed by transactional memory operations for TagTM and MetaTM. Particularly when bandwidth consumption is high, notifying transactions can show a significant reduction in peak bandwidth usage. Circled regions in the figure indicate periods of high bandwidth usage where peak bandwidth is significantly reduced in TagTM

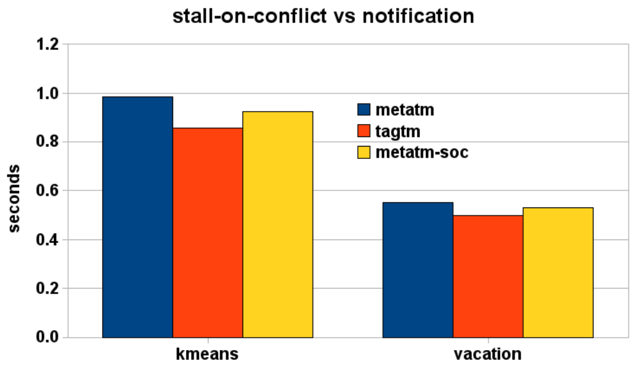


Figure 5.8: MetaTM vs TagTM vs Stall on conflict, 32 cpu SMP, 12GB/sec.

relative to MetaTM. In the config benchmark, for example, a normal MetaTM run peaks 52% higher during the first circled peak than the notifying run, while intruder sees a 14% delta.

5.5.2 Notifying transactions vs Stall-on-conflict

Stall-on-conflict [58], in which the HTM subsystem stalls a requesting memory operation in response to a detected conflict, rather than invoking the contention manager, is very similar in spirit to notifying transactions. Notifying transactions are similar to stall-on-conflict [58] in that the mechanism allows a conflicting transaction to wait in hopes that a conflict will resolve. However there are some fundamental differences. Stall-on-conflict leaves conflicting transactions in progress, increasing the probability that additional conflicts can arise for other cache lines held in the read-write set of the stalled transaction. Stall-on-conflict introduces the need for deadlock detection/avoidance, and brings up thorny policy issues around asymmetric conflict. Most importantly, stall-on-conflict is not software-visible, and it causes transactions to stall in the coherence layer: decisions to wait or retry can be better made by software. Notifying transactions give software flexibility to decide how to handle contention, which hardware-based backoff and stall-on-conflict policies cannot provide.

Despite these differences, we evaluate the performance of TagTM relative to MetaTM and MetaTM using stall-on-conflict mode, to understand any tradeoffs on performance. Figure 5.8 shows execution (user) time in seconds for `kmeans` and `vacation` (chosen because they have non-trivial contention). While stall-on-conflict improves MetaTM performance on these benchmarks, TagTM outperforms both variants of MetaTM, primarily because notification allows the losing transaction to release its transactional resources while it waits, reducing conflict rates further. Moreover, use of notification leaves the decision about when to wait in the hands of the programmer rather than in the coherence fabric.

5.5.3 Evaluating Transaction Annotation

We evaluate transaction annotation against software contention management by extending the MetaTM framework and TxLinux to support trapping into the OS (TxLinux) to run contention management algorithms in software. Implementing contention management in software required solving several of the race conditions discussed in section 5.2.1. Processors must save state for active transactions at the time of a conflict, until it is requested by a software handler to make a contention management decision. Conflicting transactions on remote processors are not stalled while software handlers run, so the handler must ensure that if a transaction commits while the handler executes, the local transaction is restarted regardless of the contention management decision.

Asymmetric conflicts are always resolved in favor of a non-transactional thread, and so do not invoke software handlers. While asymmetric conflict does not occur in the user-mode benchmarks from STAMP, asymmetric conflict does occur in TxLinux. Software handlers in TxLinux execute in a separate non-transactional interrupt context, and rely on asymmetric conflicts to restart transactions by reading

	rst-pct	**rst/tx**	**res-pct**	**speedup**
bayes	16.7	55.9	91.5	0.54
genome	0.7	0.1	67.4	0.97
kmeans	19.3	1.1	0.1	0.99
pmake	2.9	0.2	16.6	1.0

Table 5.7: Comparing SW handlers and transaction annotation. The **rst-pct** shows the percentage of transactions that restart, the **rst/tx** column is the number of restarts per transaction. The **res-pct** column is the percentage of transactions that restarted at least once, for which a software handler was able to successfully resolve a conflict. The **speedup** column is the throughput of the SW hander-based implementation over TagTMusing transaction annotation. All experiments shown use a 16 cpu CMP.

a memory location known to have been written by the transaction (written explicitly after the `xbegin` instruction) Strong isolation is then used to resolve the race conditions between a remote conflicter and a local software handler. Once a conflict handler has reached a decision, if that decision is in favor of the local transaction, the remote transaction must be aborted. Reading the location written by the remote conflict just after it's `xbegin` instruction *outside a transaction* ensures that if the remote conflicter is still active, the asymmetric conflict will cause it to abort, enforcing the local contention management decision. Conversely, if the memory read returns the updated value written by the remote transaction, then the transaction has already committed (if it were active, isolation would ensure the previous value were returned), and the contention management decision must be reversed (the local transaction aborts).

In addition to asymmetric conflicts, other cases exist where software handlers cannot be invoked. Because of the significant complexity in software and hardware support for reentrant conflict handlers, software conflict handlers are not invoked for processors already running a conflict handler. Conflict handlers are not invoked for specialized transactional CAS instructions (**xcas**, **xtest**), used for fairness when eliding locks in the kernel. Unless contention management policy is duplicated in both hardware and software (requiring similar mechanisms to transaction annotation even in systems with software contention management), these situations must fall back to a policy such as requester-wins, which is susceptible to livelock.

To isolate the effects of the decision to use or not use software handlers, we change some of the restrictions from section 5.5.1. Transactional lock elision is used to handle overflows, and we do not allow software to use notifying transactions so that conflicts are not ameliorated by other mechanisms. Additionally, this comparison takes place with unconstrained memory bandwidth. When a transactional conflict cannot be resolved in software, it is resolved by the simulator using the timestamp policy.

Table 5.7 shows statistics for benchmarks run with software handlers and run with TagTM using transaction annotation. Note that both user and kernel benchmarks run contention management algorithms in kernel mode in our implementation. Actual performance deltas are depend on the conflict rate and the ability of TagTM to actually resolve the conflict using software handlers. For the `genome` benchmark, 67% of restarting transactions were successfully handled using SW contention handlers, but the low conflict rate (less than 1%) ameliorates the performance impact, resulting in a slowdown of only 3%. The `pmake` benchmark follows the same pattern. In contrast, `bayes` conflicts can be resolved with software handlers over 90% of the time: in combination with a significant conflict rate (20% of transactions restart, and most restart many times), the result is a performance penalty of over 46%. The `kmeans` benchmark is characterized by a high conflict rate and many many short transactions: in this case the vast majority of conflicts occur when a conflict is already being handled by the detecting CPU, with the result that the vast majority of conflicts cannot be resolved with handlers. This implementation of software handlers defaults to hardware contention management when this occurs, so `kmeans` sees no significant performance delta, but makes little use of software contention management. In cases where software handlers impact performance, the source of the performance delta is increased time spent handling contention. The total restarts and total number of cycles spent backing of (using linear backoff policy) grow dramatically when software contention management is used, primarily because conflicting transactions remain active longer, increasing the window of opportunity for other conflicts to occur.

Table 5.7 also shows the percentage of time a restart could not be resolved by a software handler either because the conflict was asymmetric, the conflict was caused by a transactional atomic instruction, or handling the conflict in software would require the handler to be re-entrant. Note that the kernel `pmake` is the only context in which the former two can occur. In the STAMP benchmarks `bayes`, `genome`, and `kmeans`, conflicts can only be unresolvable for the latter reason. The vast majority of conflicts cannot be handled by software and must fall back on a less flexible mechanism. If the requirements for HTM are flexible contention management, flexible policy will have to be available at the hardware level so that frequent fall-backs do not result in a *de facto* inflexible system, begging the question of why a system builder would include software contention management in the first place.

Finally, it is worth observing that this implementation of software contention management handlers relies on support for strong isolation from the HTM, and in particular on the policy that non-transactional operations win in any asymmetric conflicts. Other HTMs which use different policies for asymmetry (e.g. LogTM [58] stalls non-transactional operations on asymmetric conflict) would have to rely on a different and likely much more expensive mechanism for handling race conditions between remote transactions and local contention handlers.

5.6 XMESI implementation details

This section explores implementation details of the **XMESI** protocol used to support TagTM. As detailed in the previous sections, **XMESI** is the substrate from which support for local flexible hardware-based contention management and notifying transactions can be composed. The protocol supports features that are useful in any cache-based HTM, which are explored here. The section concludes with a detailed enumeration of the state space and design for an **XMESI** implementation for a split-transaction bus, and details techniques used to establish the correctness of the **XMESI** implementation used to simulate TagTM.

5.6.1 Reducing latency for newly committed data

Transactional access to non-transactionally modified, exclusive cache lines introduces a source of latency in a transactional protocol that is absent from traditional protocols such as MESI. When a line is cached in **M**, it is the only copy of the committed value in the system. If this value is read, and subsequently written in a transaction, a writeback must occur because the cache line is about to be used for speculation, and failure to write the globally committed value back before speculating will result in lost updates. In the common, uncontended case where a processor is running transactions, but working only on data that is cache resident, repeated transactional updates, cause the line to cycle between the **M** and **TMM** state, effectively requiring a writeback for each successful transaction that would not be required in the absence of speculation. These additional write backs can cause significant latency.

XMESI addresses the performance costs associated with repeated write-backs of newly-committed transactional writes by allowing some state transitions that require communication with lower cache levels to occur without blocking. At the same time, we allow the lowest level to elide communication for these misses altogether. This translates to requiring non-blocking lower level caches, or buffering of memory operations between levels for operations that cause certain types of transactional transitions.

The transitions for which **XMESI** can implement non-blocking transitions

	TS	**TMU**	**TMM**
S	$\surd$		
M		$\surd$	$\surd$

Table 5.8: State transitions that can occur without blocking in higher level caches and which can elide traffic in caches at lower hierarchy levels, assuming that the transition does not cause a transactional conflict.

at higher cache levels and elide traffic on misses at the lowest level are detailed in table 5.8.

5.6.2 Interaction with Pause/Resume

Many HTM proposals include mechanisms to suspend and resume the current transaction to allow software to execute some operations outside the transactional context. Suspend primitives are a fundamental building block for open nesting [60, 62], escape actions [59, 96], and can facilitate interrupt handling in OSes that synchronize using HTM [70]. Consequently, most HTM designs support some form of suspend/resume. In MetaTM, the `xpush` and `xpop` instructions are a form of suspend and resume (requiring transactions to suspend and resume in LIFO order), specialized to the problem of allowing interrupt handlers in TxLinux. If a cache-based HTM provides support such as this for starting new transactions while another hardware transaction is paused, the possibility for read-sharing between transactions on the same CPU arises, along with the concommitant problem of how to encode membership in multiple read sets for such lines. The problem of transactions read-sharing a cache line on the same core can also arise if the cores support SMT [92]–the problem is not unique to systems that support suspend/resume. Support for multiple transactions for each thread context and conflict detection through caching introduces the problem of sharing the same processor. This problem will not occur in a signature-based design [94].

Addressing same-CPU sharing requires some mechanism to map a single cache line to multiple transactions. The most obvious approach augments the line with multiple TXID fields, which is similar to mechanisms proposed to handle closed nesting [48, 59]. This method is unattractive because it involves increasing the size of every cache line in all levels of the hierarchy to handle a rare corner case. A more attractive solution involves enhancing set associativity logic to handle this case explicitly: if a second transaction reads a line that is already cached and marked with another transaction ID, we allocate a second line in the same set to cache the line. The result is that the same data is cached twice in the set. This solution is similar to the mechanism proposed in [48] to handle nested transactions (which may require multiple speculative versions). In the worst case, this solution can cause overflows for other transactions, potentially necessitating restarts. But since the mechanism is only needed for transient conditions, it should not become a barrier to forward progress, and subsequent retries can succeed without overflow.

5.6.3 Overflow and Prefetch

XMESI provides support to communicate overflow types to the CPU, which sets bits in the TXSW (transactional status word). As a result, software can distinguish between structural and transient overflows. **XMESI** also supports communication of conflict type (asymmetric or transactional) as well as information about whether remote conflicting transactions are suspended, or stalled for conflicts. This enables a local CPU to avoid stalling for doomed transactions. Bus requests serving prefetch instructions are labeled in **XMESI** instructions, which enables any controller that is caching the line in a state that would cause its local transaction to abort to raise a line similar to the shared line in MESI. This label allows the initiating controller to avoid changing local state for that line, effectively resulting in a silent (but harmless) failure of the prefetch instruction, and allowing remote transactions to avoid restarts. Controllers seeing a bus read labeled as a prefetch for any line cached in a transactional state (T*) nack the request, allowing the remote CPU to

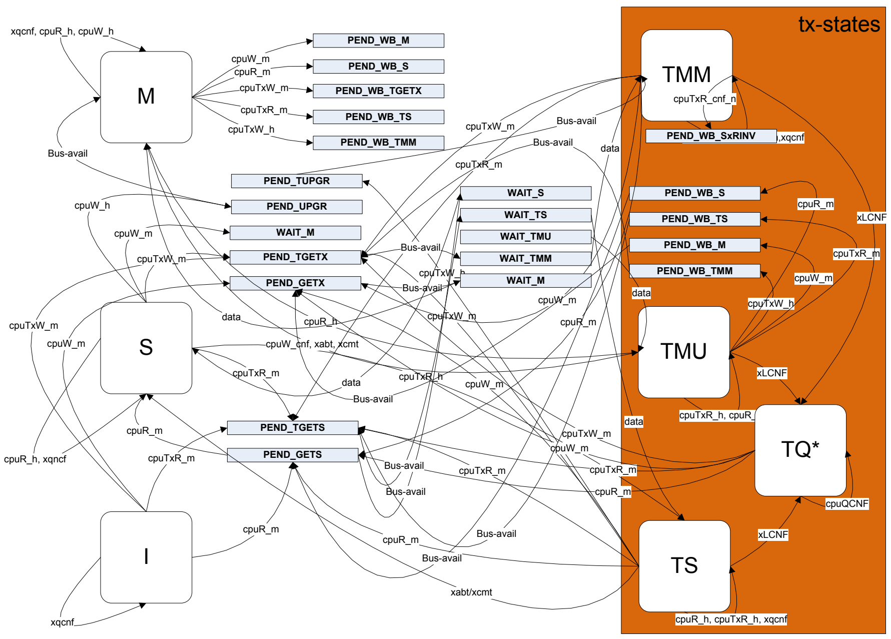


Figure 5.9: The **XMESI** protocol. Controllers at different levels of the hierarchy run slightly different variations on the protocol. States in the box labeled “lowest level”, along with **M**, **S** , and **I** represent the state space for caches at the lowest level of the memory hierarchy (just above main memory). States in the box labeled “upper levels” are the state space for caches at all other levels. The **TMU** state indicates a line that was previously in **M**, and has been read but not written in a transaction. The value is dirty with respect to main memory, but is the current globally committed value. The **TMM** state indicates a line that has been written in a transaction, and is therefore both speculative and modified with respect to main memory. The **TQS** and **TQM** states are states that indicate a line was recently involved in a conflict–these states have the same semantics as **I** for all purposes except interaction with the **xquery_tqc** instruction (see section 5.4.2). All edges labeled **cpuTxR, cpuTxW** indicate memory operations from the processor and are parameterized with a transaction identifier (TXID) and **TxTAG**. Edges labeled with **xCMT** indicate a transaction commit event. Edges labeled **xLCNF(r)** and **xLCNF(w)** indicate a remote conflict response to a locally initiated read or write respectively, and indicate that the local transaction has lost arbitration and must abort. Edges labeled **SxWINV** and **SxRINV** indicate transitions triggered by the commit or abort of a remote transaction that conflicted with the local one.

commute the prefetch to a no-op.

5.6.4 Detailed Implementation and State Space

This section examines coherence state machine details for a split-transaction bus implementation of **XMESI**. Figure 5.9 shows the **XMESI** state machine including transient states. Table 5.9 details the list of actions the cpu can take, broken down and renamed according to whether the reference is transactional or not, and according to whether the cache line involved is present (in any coherence state), whether the transaction id on the cache line matches the current transaction id of the processor, and whether the **xC** bit is set on that line, indicating the lines involvement in a recent conflict (in the R/W set of the winning transaction). Note that the CPU is capable only affecting coherence state only by reading and writing (either in the context of a transaction or not), or executing the `xend`, `xretry`, or `xquery_tqc` instructions. The transactional atomic instructions `xcas` and `xtest` appear to the cache as transactional reads and writes. For convenience in representing the coherence state machine in tabular form, we take a cross-product of these actions and relevant affecting state. For example, a read generated by the cpu appears in the table as **cpuR_m** and **cpuR_h** according to whether the line is already present in some valid state; **cpuR_cnf** indicates a cpu read to a line that has a non-zero transaction id, indicating a potential asymmetric read.

Table 5.11 lists actions that may be taken on the bus. Note the absence of an upgrade request type. Upgrades (**S** $\rightarrow$ **M** transitions) are handled using the **GETX** request (get exclusive). When the requesting cpu requests an exclusive copy, it snoops it's own contents–if a local copy in **S** state is present, it asserts the shared line, inhibiting a response from memory, preventing an unnecessary response. This technique is safe because the presence of a local **S** copy indicates that no modified copies can exist elsewhere, and the globally committed version of the data is already present [20].

Like many MESI implementations, **XMESI** relies on three wired-OR sig-

name	**R/W**	**tx**	**h**	**m**	**xC**	**description**
cpuR_m	R					local read miss
cpuR_m	R			×		local read miss
cpuR_m_n	R				×	local read miss (notify)
cpuR_m_n	R			×	×	local read miss (notify)
cpuR_cnf	R		×			local asymmetric read conflict
cpuR_cnf_n	R		×		×	local asymmetric read conflict (notify)
cpuR_h	R		×	×		local read hit
cpuTxR_m	R	×				local transactional read miss
cpuTxR_m	R	×		×		local transactional read miss
cpuTxR_m_n	R	×			×	local transactional read miss (notify)
cpuTxR_m_n	R	×		×	×	local transactional read miss (notify)
cpuTxR_cnf	R	×	×			local transactional read conflict
cpuTxR_cnf_n	R	×	×		×	local transactional read conflict (notify)
cpuTxR_h	R	×	×	×		local transactional read hit
cpuW_m	W					local write miss
cpuW_m	W			×		local write miss
cpuW_m_n	W				×	local write miss (notify)
cpuW_m_n	W			×	×	local write miss (notify)
cpuW_cnf	W		×			local asymmetric write conflict
cpuW_cnf_n	W		×		×	local asymmetric write conflict
cpuW_h	W		×	×		local write hit
cpuTxW_m	W	×				local transactional write miss
cpuTxW_m	W	×		×		local transactional write miss
cpuTxW_m_n	W	×			×	local transactional write miss (notify)
cpuTxW_m_n	W	×		×	×	local transactional write miss (notify)
cpuTxW_cnf	W	×	×			local transactional write conflict
cpuTxW_cnf_n	W	×	×		×	local transactional write conflict (notify)
cpuTxW_h	W	×	×	×		local transactional write hit
xabt	`xretry`			×		local programmer initiated abort
xcmt	`xend`			×		local commit transaction
xqcnf	`xquery`			×		query cache line state == **TQM** or **TQS**

Table 5.9: Actions triggered by the cpu that can affect cache line states. **R** stands for read and **W** stands for write. The **tx** column indicates whether the given read or write was executed while cpu has an active transaction. The **h** column indicates whether the line requested is in cache or not (hit/miss). The **m** column indicates whether the **TXID** field on the cache line matches the **TXID** field in the local cpu's **TXSW** (transaction status word). Note that **TXID** values of zero indicate non transactional actions, so, for example, if the **m** column is blank for a non-transactional read, this indicates the line is cached in some transactional state, which indicates an asymmetric conflict. The **xC** column indicates whether the **xC** bit on the cache line is set, indicating that the line was involved in a transactional conflict, and the transaction running on the local cpu won that conflict.

state	**T**	**description**
I	S	Invalid
S	S	Shared
M	S	Modified
TS	S	Shared, read in a transaction
TMU	S	Modified, read in a transaction
TMM	S	Exclusive, written in a transaction
TQS	S	Invalid, lost conflict to remote read
TQM	S	Invalid, lost conflict to remote write
PEND_GETS	T	wait(bus), share request
PEND_GETS_SxINV	T	wait(bus), share request, notify
PEND_GETX	T	wait(bus), exclusive request
PEND_GETX_SxINV	T	wait(bus), exclusive request, notify
PEND_UPGR	T	wait(bus), **S** $\rightarrow$ **T**
PEND_TGETS	T	wait(bus), tx share request
PEND_TGETS_SxINV	T	wait(bus), tx share request, notify
PEND_TGETX	T	wait(bus), tx exclusive request
PEND_TGETX_SxINV	T	wait(bus), tx exclusive request, notify
PEND_TUPGR	T	wait(bus), tx upgrade, **TS** $\rightarrow$ **TMM**
PEND_TUPGR_SxINV	T	wait(bus), tx exclusive request, notify
PEND_WB_S	T	wait(bus), writeback share request
PEND_WB_S_SxINV	T	wait(bus), writeback share request, notify
PEND_WB_M	T	wait(bus), writeback exclusive request
PEND_WB_M_SxINV	T	wait(bus), writeback exclusive request, notify
PEND_WB_TS	T	wait(bus), writeback tx share request
PEND_WB_TS_SxINV	T	wait(bus), writeback tx share request, notify
PEND_WB_TGETX	T	wait(bus), writeback tx exclusive request
PEND_WB_TGETX_SxINV	T	wait(bus), writeback tx exclusive request,notify
PEND_WB_TMM	T	wait(bus), writeback **TMU** $\rightarrow$ **TMM**
PEND_WB_TMM_SxINV	T	wait(bus), writeback **TMU** $\rightarrow$ **TMM**, notify
PEND_WB_SxRINV	T	wait(bus), read conflict resolved
PEND_WB_SxWINV	T	wait(bus), write conflict resolved
PEND_WB_SxWINV_TMM	T	wait(bus), write conflict resolved , next **TMM**
PEND_LCNF_TS	T	wait(bus), lost conflict reply, next state **TS**
PEND_LCNF_TMU	T	wait(bus), lost conflict reply, next state **TMU**
PEND_LCNF_TMM	T	wait(bus), lost conflict reply, next state **TMU**
WAIT_S	T	wait(shared copy)
WAIT_M	T	wait(exclusive copy)
WAIT_TS	T	wait(shared copy)/tx permissions
WAIT_TMU	T	wait(exclusive copy)/tx permissions
WAIT_TMM	T	wait(exclusive copy)/tx permissions

Table 5.10: The **XMESI** state space. The **T** column indicates whether a state is **S**table or **T**ransient.

msg	**tx**	**description**
gets		request for shared copy
gets_sxinv		request for shared copy, piggyback notify
getx		request for exclusive copy
getx_sxinv		request for exclusive copy, piggyback notify
data		memory responds with requested cache line
data_xC		memory responds with requested cache line, a tx conflict occurred
tgets	×	request for tx shared copy
tgets_sxinv	×	request for tx shared copy, piggyback notify
tgetx	×	request for tx exclusive copy
tgetx_sxinv	×	request for tx exclusive copy, piggyback notify
wb		writeback of exclusive copy
wb_WINV		writeback of exclusive copy, **xC** was set
xLCNF(R)	×	lost conflict response (CM has ruled against reading requester)
xLCNF(W)	×	lost conflict response (CM has ruled against writing requester)
SxRINV	×	notification that remote conflicting reader has aborted/committed
SxWINV	×	notification that remote conflicting writeer has aborted/committed

Table 5.11: Actions triggered by messages observed on the interconnect by the bus-side controller.

nals to coordinate snooping and memory responses to requests. The *share* line is asserted by any controller caching a line in **S** and the *modify* line is asserted by a controller whose cache contains an **M** copy. **XMESI** relies on a *snoop-valid* line to indicate that all caches have produced snoop results. Additionally, **XMESI** relies on *sxrinv* and *sxwinv* lines to piggyback notifications on other messages. For instance, the **wb_sxrinv** message type indicates a writeback occurring for a line that may be cached in **TQS** in one or more caches. Over and above the traditional **GETS** (get shared) and **GETX** (get exclusive) requests, **XMESI** uses **TGETS** and **TGETX** to indicate transactional variants of gets. The bus is augmented with **TXID** and **TxTAG** lines as well to parameterize each request. In general, where bus requests in MESI are (command,address) pairs, bus requests in **XMESI** become (command, address, txid, txcookie) tuples.The **xLCNF** and **Sx*INV** request are (cmd,address,txid) tuples that communicate a lost conflict (similar to a nack) and a notification that a remote conflicter has committed, respectively. Note that in general, each command type needs a **Sx*INV** variant because any action that causes a local cache eviction can cause a line with a valid **TXID** and it's **xC** bit to be

state	**action**	**next**	**resp**	**comments**
I	**cpuR_m**	**PEND_GETS**	wait(bus)	read miss
	cpuTxR_m	**PEND_TGETS**	wait(bus)	txnl read miss
	cpuW_m	**PEND_GETX**	wait(bus)	write miss
	cpuTxW_m	**PEND_TGETX**	wait(bus)	txnl read miss
	xqcnf	**I**		return 0
S	**cpuR_m**	**PEND_GETS**	wait(bus)	assoc eviction
	cpuR_h	**S**		read hit
	cpuTxR_m	**PEND_TGETS**	wait(bus)	tx read miss, assoc evict
	cpuTxR_h	**TS**		tx read hit
	cpuW_m	**WAIT_M**	wait(bus)	write miss, assoc evict
	cpuW_h	**PEND_UPGR**		upgrade to **M**
	cpuTxW_m	**PEND_TGETX**	wait(bus)	tx write miss, assoc evict
	xqcnf	**S**		`xquery_tqc` returns 0
M	**cpuR_m**	**PEND_WB_S**	wait(bus)	assoc evict, writeback
	cpuR_h	**M**		read hit
	cpuTxR_m	**PEND_WB_TGETX**	wait(bus)	assoc evict, writeback
	cpuTxR_h	**TMU**		tx read hit
	cpuW_m	**PEND_WB_M**	wait(bus)	write miss, assoc evict
	cpuW_h	**M**		write hit
	cpuTxW_m	**PEND_WB_TGETX**	wait(bus)	tx write miss, assoc evict
	cpuTxW_h	**PEND_WB_TMM**	wait(bus)	tx write hit, writeback
	xqcnf	**M**		`xquery_tqc` returns 0

Table 5.12: State transitions for blocks in **M**, **S**, or **I** state. The **next** column indicates the next coherence state for the line, the **resp** column indicates any message that needs to be sent as part of the transition. In the interest of compact representation, only state-action pairs for which a transition must occur are shown.

evicted. This is an overflow that will cause the transaction to abort, and the **xC** bit indicates that a remote transaction may be awaiting notification for relevant state changes on that line.

Table 5.10 shows the comprehensive state space for split-transaction bus-based **XMESI**, including both stable and transient states. Tables 5.12, 5.13, and 5.14 show state transitions for cache lines in response to cpu-side events, while tables 5.16 and 5.15 summarize responses to interconnect-side events.

5.6.5 Support for notification in L2 caches

Support for **TQ*** states in an SMP-like organization requires minor changes to lower level caches to ensure that coherence events for lines an upper level cache holds in

state	action	next	action	N	O	cmnts
TS	**cpuR_m**	**PEND_GETS**	wait(bus)		×	evict
	cpuR_m_n	**PEND_GETS_SxINV**	wait(bus)	×	×	evict
	cpuR_cnf	**TS**				asym R-R
	cpuR_cnf_n	**TS**				asym R-R
	cpuR_h	**TS**				asym R-R
	cpuTxR_m	**PEND_TGETS**	wait(bus)		×	evict
	cpuTxR_m_n	**PEND_TGETS_SxINV**	wait(bus)	×	×	evict
	cpuTxR_cnf	**TS**				R-R conf
	cpuTxR_cnf_n	**TS**				R-R conf
	cpuTxR_h	**TS**				tx R hit
	cpuW_m	**PEND_GETX**	wait(bus)		×	evict
	cpuW_m_n	**PEND_GETX_SxINV**	wait(bus)	×	×	evict
	cpuW_cnf	**PEND_GETX**	wait(bus)			asym conf
	cpuW_cnf_n	**PEND_GETX_SxINV**	wait(bus)	×		asym conf
	cpuTxW_m	**PEND_TGETX**	wait(bus)		×	evict
	cpuTxW_m_n	**PEND_TGETX_SxINV**	wait(bus)	×	×	evict
	cpuTxW_cnf	n/a				local conf
	cpuTxW_cnf_n	n/a		×		local conf
	cpuTxW_h	**PEND_TUPGR**				tx upgr
	xabt	**S**	*			abort
	xcmt	**S**	*			commit
	xqcnf	**TS**				return 0
TMU	**cpuR_m**	**PEND_WB_S**	wait(bus)		×	evict, wb
	cpuR_m_n	**PEND_WB_S_SxINV**	wait(bus)	×	×	evict, wb
	cpuR_cnf	**TMU**				asym R-R
	cpuR_cnf_n	**TMU**				asym R-R
	cpuR_h	**TMU**				R hit
	cpuTxR_m	**PEND_WB_TS**	wait(bus)		×	evict, wb
	cpuTxR_m_n	**PEND_WB_TS_SxINV**	wait(bus)	×	×	evict, wb
	cpuTxR_cnf	**TMU**				R-R conf
	cpuTxR_cnf_n	**TMU**				R-R conf
	cpuTxR_h	**TMU**				tx R hit
	cpuW_m	**PEND_WB_M**	wait(bus)		×	evict, wb
	cpuW_m_n	**PEND_WB_M_SxINV**	wait(bus)	×	×	evict, wb
	cpuW_cnf	**M**				asym conf
	cpuW_cnf_n	**M**		×		asym conf
	cpuTxW_m	**PEND_WB_TMM**	wait(bus)		×	evict, wb
	cpuTxW_m_n	**PEND_WB_TMM_SxINV**	wait(bus)	×	×	evict, wb
	cpuTxW_cnf	n/a				local conf
	cpuTxW_cnf_n	n/a		×		local conf
	cpuTxW_h	**PEND_WB_TMM**	wait(bus)			
	xabt	**M**				abort
	xcmt	**M**				commit
	xqcnf	**TMU**				return 0

Table 5.13: State transitions for blocks in the **TS** and **TMU** state. Note that associativity evictions for lines in these states constitute overflow, so the **TXOV** signal must be raised (**O** column). The **next** column is the next coherence state for the line, the **resp** column indicates any necessary response for the transition. The **N** column indicates whether the transition requires notification.

state	action	next	action	N	O	comments
TMM	**cpuR_m**	**PEND_GETS**	wait(bus)		×	evict
	cpuR_m_n	**PEND_GETS_SxINV**	wait(bus)	×	×	evict
	cpuR_cnf	**PEND_GETS**	wait(bus)			asym R-W
	cpuR_cnf_n	**PEND_GETS_SxINV**	wait(bus)	×		asym R-W
	cpuTxR_m	**PEND_TGETS**	wait(bus)		×	evict
	cpuTxR_m_n	**PEND_TGETS_SxINV**	wait(bus)	×	×	evict
	cpuTxR_cnf	**PEND_LCNF_TMM**	wait(bus)			local wins
	cpuTxR_cnf	**I**				local loses
	cpuTxR_cnf_n	**PEND_SwRINV**	wait(bus)	×		local loses
	cpuTxR_h	**TMM**				tx R hit
	cpuW_m	**PEND_GETX**	wait(bus)		×	evict
	cpuW_m_n	**PEND_GETX_SxINV**	wait(bus)	×	×	evict
	cpuW_cnf	**M**				asym conf
	cpuW_cnf_n	**PEND_SxWINV**	wait(bus)	×		asym conf
	cpuTxW_m	**PEND_TGETX**	wait(bus)		×	evict
	cpuTxW_m_n	**PEND_TGETX_SxINV**	wait(bus)	×	×	evict
	cpuTxW_cnf	**TMM**				local conf
	cpuTxW_cnf_n	**PEND_SxWINV_TMM**	wait(bus)	×		local conf
	cpuTxW_h	**TMM**				tx W hit
	xabt	**I**				abort
	xcmt	**M**				commit
	xqcnf	**TMM**				return 0
TQ*	**cpuR_m**	**PEND_GETS**	wait(bus)			
	cpuTxR_m	**PEND_TGETS**	wait(bus)			tx R miss
	cpuW_m	**PEND_GETX**	wait(bus)			write miss
	cpuTxW_m	**PEND_TGETX**	wait(bus)			tx R miss
	xqcnf	**I**				return 1

Table 5.14: State transitions for blocks in the **TMM** and **TQ*** states. The **next** column indicates the next coherence state for the line, the **msg** column indicates any message that needs to be sent as part of the transition. The **N** column indicates whether the transition requires any additional **SxRINV** or **SxWINV** messages to remote conflicters that a relevant event has occurred. Note that like other **T*** states, **TMM** requires the **TXOV** signal to be raised on overflows (associativity evictions for lines marked transactional), indicated by the **O** column. Note that **TQS** and **TQM** indicate a line that was involved in a remote transactional conflict where the conflicting transaction was a reader, and a writer respectively. The two are condensed into a single table because the states share the same semantics with respect to cpu-side actions, and differ only in response to bus-side actions.

state	**msg**	**next**	**response**	**s**	**m**	**xC**	**cmnts**
S	**gets**	**S**		×			
	getx	**I**					
	tgets	**S**		×			
	tgetx	**I**					
M	**gets**	**WAIT_WB_S**	**wb**		×		
	getx	**WAIT_WB_I**	**wb**		×		
	tgets	**WAIT_WB_S**	**wb**		×		
	tgetx	**WAIT_WB_I**	**wb**		×		
TS	**gets**	**TS**		×			
	getx	**S**	abort				asym. conf
	tgets	**TS**		×			read-read conf
	tgetx	**TS**	**xLCNF**	×		×	W-R, local win
	tgetx	**TQS**					W-R, local abt
TMU	**gets**	**WAIT_WB_TS**	**wb**		×		asymmetric R-R
	getx	**WAIT_WB_S**	**wb**		×		asymmetric R-W
	tgets	**WAIT_WB_TS**	**wb**		×		R-R conf–wb
	tgetx	**WAIT_WB_TQM**	**wb**		×		W-R, local abt
	tgetx	**TMU**	**xLCNF**			×	W-R, local win
TMM	**gets**	**I**					asymmetry, abt
	getx	**I**					asymmetry, abt
	tgets	**TMM**	**xLCNF**			×	R-W, local win
	tgets	**TQS**					R-W, local abt
	tgetx	**TMM**	**xLCNF**			×	W-R, local win
	tgetx	**TQM**					W-R, local abt
TQS	**getx**	**I**					conflicter dead
	tgetx	**I**					new conflict
	wb	**I**					conflicter dead
	wb_WINV	**I**					conflicter dead
	SxRINV	**I**					notified!
TQM	**gets**	**I**					conservative
	getx	**I**					conflicter dead
	tgets	**I**					conservative
	tgetx	**I**					conservative
	wb	**I**					conflicter dead
	wb_WINV	**I**					conflicter dead
	xLCNF(R)	**I**					conservative
	xLCNF(W)	**I**					conservative
	SxRINV	**I**					notified!
	SxWINV	**I**					notified!

Table 5.15: Bus side controller responses to messages snooped that match a line cached in any valid state, or **TQ***. Lines are shown *only* for request-state pairs that require some response from the controller that snoops that bus request. The **s** column indicates whether the *share* line should be asserted. The **m** column indicates whether the *modify* line should be asserted. **xC** indicates whether the **xC** bit on the local cache line should be set on this transition. In the comments column, transitions from **TQ*** states marked "conservative" represent situations where the event may or may not indicate that a remote conflicting transaction has committed or aborted. Transitioning to **I** on these events may cause cpus spinning with the `xquery_tqc` instruction to retry transactions sooner than ideal, but avoids the risk of allowing spinning cpus to miss relevant notifications and spin too long.

state	event	next	action	xC
PEND_GETS	bus avail	**WAIT_S**	**gets** on bus	
PEND_GETS_SxINV	bus avail	**WAIT_S**	**gets+SxRINV**	
PEND_GETX	bus avail	**WAIT_M**	**getx** on bus	
PEND_GETX_SxINV	bus avail	**WAIT_M**	**getx+SxWINV**	
PEND_TGETS	bus avail	**WAIT_TS**	**tgets** on bus	
PEND_TGETS_SxINV	bus avail	**WAIT_TS**	**tgets+SxRINV**	
PEND_TGETX	bus avail	**WAIT_TMM**	**tgetx** on bus	
PEND_TGETX_SxINV	bus avail	**WAIT_TMM**	**tgetx+SxWINV**	
PEND_WB_S	bus avail	**PEND_GETS**	**wb** on bus	
PEND_WB_S_SxINV	bus avail	**PEND_GETS**	**wb+SxRINV**	
PEND_WB_M	bus avail	**PEND_GETX**	**wb** on bus	
PEND_WB_M_SxINV	bus avail	**PEND_GETX**	**wb+SxWINV**	
PEND_WB_TS	bus avail	**PEND_TGETS**	**wb** on bus	
PEND_WB_TS_SxINV	bus avail	**PEND_TGETS**	**wb+SxWINV**	
PEND_WB_TMM	bus avail	**PEND_TGETX**	**wb** on bus	
PEND_WB_TMM_SxINV	bus avail	**PEND_TGETX**	**wb+SxWINV**	
PEND_WB_TGETX	bus avail	**PEND_TGETX**	**wb** on bus	
PEND_UPGR	bus avail	**M**	**GETX**, share	
PEND_TUPGR	bus avail	**TMM**	**TGETX**, share	
PEND_SxRINV	bus avail	**I**	**SxRINV**	
PEND_SxWINV	bus avail	**I**	**SxWINV**	
PEND_SxWINV_TMM	bus avail	**TMM**	**SxWINV**	
PEND_LCNF_TS	bus avail	**TS**	**xLCNF** on bus	×
PEND_LCNF_TMU	bus avail	**TMU**	**xLCNF** on bus	×
PEND_LCNF_TMM	bus avail	**TMM**	**xLCNF** on bus	×
WAIT_S	**data**	**S**		
WAIT_S	**data_xC**	**S**		
WAIT_M	**data**	**M**		
WAIT_M	**data_xC**	**M**		
WAIT_TS	**data**	**TS**		
WAIT_TS	**data_xC**	**TS**		×
WAIT_TMU	**data**	**TMU**		
WAIT_TMU	**data_xC**	**TMU**		×
WAIT_TMM	**data**	**M**		
WAIT_TMM	**data_xC**	**M**		

Table 5.16: Bus side controller responses to messages snooped that match a line cached in transient states.

TQ* are visible to that cache. This can be accomplished by allocating a single bit on cache lines in lower level caches that indicates that a line in **I** in a lower level cache was recently involved in a transactional conflict. This allows interconnect-side controllers to continue to observe and propagate coherence traffic for such lines, but note that lower level caches are free to treat the line as invalid in all respects, including reallocating the line to cache a different block of memory. When this occurs the lower level cache must (conservatively) invalidate copies in upper level caches. However, even if lower level caches do not augment the state MESI state space beyond the additional transient states needed to communicate contention management results, support for the **XMESI** augmented set of bus commands must be present: the bus must be capable of being driven with (command, address, txid, **TxTAG**) tuples rather than simple (command, address) pairs traditionally associated with bus implementations.

5.7 Verifying Correctness

5.7.1 Random stress testing

To verify that the correctness of **XMESI**, an approach similar to that described by Wood et. al [93] was used. In the aforementioned approach, CPU and interconnect-side cache controllers are exercised in simulation by a virtual CPU that selects randomly from among a predefined set of operations whose results are known; the simulation runs choosing operations randomly and checking results until either an error condition is detected, or the user stops the simulation. In the limit where the simulation runs, all possible test cases for the controllers are covered by the test.

The presence of transactions complicates this approach because transactions can conflict. Memory operations may cause speculative execution on the CPU to roll back, introducing the possibility that a write is squashed or a read has no meaningful result. The presence of suspend primitives such as `xpush` and `xpop` makes it possible for transactions to be aborted while the are suspended, introducing the possibility that an `xpop` operation may result in a retry. Because transactions may nest, `xbegin` and `xend` can result in more than one correct result as well: `xbegin` can cause a transition from “no active transactions” or “an active transaction”, or can simply flat nest resulting in no change in the active transaction state. An `xend` operation on a nested transaction may result in no active transactions or the current transaction can still be active. In light of this, our approach to random testing does not select randomly from among predefined operations with known results, but rather generates operations randomly. All simulated CPUs run a program that continually, based on a pseudo-random number selects from among the following in a tight loop:

- Start a transaction. No upper bound is set on the nesting depth. Note that MetaTM supports flat nesting only. (Probability 4%).
- If there is a current transaction, suspend it using `xpush`. Do not `xpush` if there is already a suspended transaction. (Probability 1%).
- If there is a current transaction, end it. (Probability 4%)
- if there is an xpushed transaction, resume it using `xpop`. Do not `xpop` if no xpushed transactions exist. (Probability 2%).
- Load a random address (within a specified buffer range that is much larger than L1 cache). Note that whether the load is transactional depends on what other operations have executed before this one. (Probability 70%)
- store a random value to an address selected by the same method as above. Transactional semantics for the operation are determined as above. (Probability 20%)

The stress testing framework supports two modes: one in which all transactions are well formed (`xbegin`/`xend` and `xpush`/`xpop` are always paired correctly), and one in which transactions are not constrained to be well formed. Note that in the latter mode, because transactions are started, ended, suspended, and resumed arbitrarily, conflicts can be asymmetric, between transaction on the local CPU and between a transaction on this CPU and a remote one. Transactions generated using this method are short. However, we are primarily interested in state transitions in the cache.

To increase stress on the coherence implementation, timing modules in the simulation are additionally enhanced to randomly perturb stall times associated with traffic in the memory hierarchy. To this end, the bus model is enhanced with a mode that effectively randomizes the access time to main memory, and each level of cache is similarly enhanced to randomize the access time for interrogating and writing to caches, as well as timings for coherence traffic. The result is increased likelihood of covering all interleavings.

5.7.2 Inclusion and sibling invariant checking

To further increase confidence in the correctness of the implementation, the simulator module implementing **XMESI** caches was enhanced to include a modes which do inclusion and invariant checking after every simulated operations which interacts with the memory hierarchy (memory operations, coherence events, and instructions that manipulate the state of active transactions on the cpu such as `xbegin` and `xpush`.

When **inclusion checking** mode is on, all caches check the inclusion property with all caches below it in the memory hierarchy. The inclusion property for **XMESI** involves some minor extension of the MESI inclusion property. Like MESI inclusion, **XMESI** requires that any line cached in a higher level cache must also be cached in all lower level caches in that tier of the hierarchy. Also like MESI

local state	**legal sibling states**	**legal higher-level states**
I	**I, TS, S, TE, E, TMM, TMU, M**	**I**
TQS	**I, TS, S, TE, E, TMM, TMU, M**	**I, TQS**
TQM	**I, TS, S, TE, E, TMM, TMU, M**	**I, TQM**
TS	**I, TS, S, TQM, TQS**	**I, TS**
S	**I, TS, S, TQM, TQS**	**I, TS, S**
TE	**I, TQM, TSQ**	**I, TS, TE**
E	**I, TQM, TSQ**	**I, TS, S, TE, E**
TMU	**I, TQM, TSQ**	**I, TS, S, TE, E, TMM, TMU, M**
TMM	**I, TQM, TSQ**	**I, TS, S, TE, E, TMM, TMU, M**
M	**I, TQM, TSQ**	**I, TS, S, TE, E, TMM, TMU, M**

Table 5.17: Legal inclusion pairs and pairs of **XMESI** states for sibling caches in a memory hiearchy. Note that the inclusion properties do admit some combinations that are safe, but should not arise in practice. Asymmetry introduces some intersting cases as well. Consider that if a transaction is suspended using `xpush`, and a line in TMU is conflicted with an asymmetric local read, the read will hit in L1 at the same time as the transaction aborts. The next state in L1 should be S, while it is safe to leave the L2 in TMU until the transaction resumes with `xpop` and discards its read set, returning the L2 state to S.

inclusion, **XMESI** inclusion requires that cached copies in lower level caches be copies whose permissions are of equal or greater strength. While MESI inclusion respects the total order M, E, S, I, **XMESI** inclusion uses the partial order: ((M, TMM, TMU), (E, TE), S, TS), (TQM, TQS), I. The legal inclusion state pairs for **XMESI** are shown in table 5.17.

While inclusion checking looks for **XMESI** invariant violations for caches in the same tier of the hierarchy, **sibling invariant checking** looks for **XMESI** invariant violations across caches at the same level of the hierarchy. The **XMESI** sibling invariants include the familiar MESI invariants (e.g. only M copy exists in any tier of the hierarchy), but is extended to handle transaction semantics as well. Table 5.17 shows the legal states for a block with same address cached in a sibling cache.

When inclusion and sibling invariant checking are turned on in simulation, inclusion and invariants are checked after every operation. Needless to say, this can be quite computationally expensive, so the cache simulation module uses a mark-dirty strategy to reduce overhead for lines that have not changed since the last check.

When a cache line is touched as part of an operation, it is marked dirty and added to a dirty list. This allows post operation inclusion and invariant checking to work only on lines that have changed since the last invariant check.

These tools, in combination with random stress testing described in section 5.7.1 exposed a great many problems (many 10s) in the cache implementation and necessitated approximately 2-person months of debugging.

Bugs recently exposed by stress testing:

- A corner case in the implementation of the **share** line (which caches assert when they snoop a **gets** on the bus for a line cached locally, allowing the requester to distinguish the next state between **E** and **S**). The corner case failed to assert the share line for lines in **M** observing a remote read. After the write-back, the cache holding the **M** copy will move to **S**, but failure to assert the share line allows the requester to move to **E** introducing multiple exclusive copies and violating an invariant. For various reasons the corner case bug occurred only in a CMP configuration.

- For some types of coherence traffic, it is useful for the cache module (in the simulator) to infer it's position in the hierarchy, which it did using some assumptions about connections to higher level caches and simulated CPUs. When a transactional store buffer was introduced for work discussed in chapter 6, those assumptions were violated, and in certain cache organizations this could lead to incorrect inference, and ultimately, to invariant violations.

- An incorrect timing penalty computed for snooping. A corner case existed where the cost of broadcasting to multiple caches (which can occur in parallel) was being summed (rather than taking the max). This bug would not have affected any results because we assess penalties only for actions in reaction to snoops that require the bus (e.g. write-backs), and the snoop itself is

0, yielding the same result for sum and max operations over all snoopers. However, randomizing penalties exposed the incorrect math in the code.

5.7.3 Comparison with a known (assumed correct) MESI implementation

In the absence of transactional memory operations, **XMESI** should behave exactly the same as MESI. To verify that implementation of the MESI subset of the **XMESI** state machine, We ran the same workload (Linux 2.6.16 boot) using the same cache organizations built with both the **XMESI** cache module and the `simics` [47] `g-cache` module, which supports a MESI implementation Weassume to be correct. Both implementations were modified to dump cache state in the same format after every memory operation, yielding traces that could be compared using the `diff` utility. This approach also exposed many of bugs in the **XMESI** cache implementation. After several weeks of debugging effort, the **XMESI** implementation yields identical traces to `g-cache` up to 100,000,000 memory operations, after which we terminate the simulation.

5.7.4 Use of asserts

The **XMESI** cache and transaction versioning C++ code have, collectively 108 asserts related to checking the correctness of state transitions and version management, including 50 dedicated to the cache model and 58 dedicated to transactional state management and version management

5.8 Conclusion

This chapter has proposed TagTM, and explored mechanisms that avoid and manage contention among transactions in an HTM. *Notifying transactions* can significantly improve performance under contention, and can reduce pressure on memory bandwidth for transactions that must restart. The chapter examined implementation details for supporting flexible contention management policies in an HTM, positing *transaction annotation* as a mechanism that allows contention management decisions to be rendered locally at nodes where conflicts are detected. Transaction annotation eliminates the the design complexities incurred by software conflict handlers suggested in previous designs. The bulk of the mechanisms rely on an underlying transactional coherence protocol, called **XMESI**. This chapter also explored details of **XMESI** implementation, including a split-transaction bus implementation, and examined the interaction of coherence and high level HTM features such as pause and resume. **XMESI** offers a mechanism to allow software defined contention management policies to be made locally at cache controllers, and provides the basic substrate for support of notifying transactions.

Chapter 6

HTM Design comparisons

Despite a tremendous investment of research effort and the proliferation of designs for hardware transactional memory [?, 5, 9, 12, 18, 19, 22, 36, 39, 48, 58, 68, 70, 71, 85, 94], comparing the merits of different HTM design points in an empirical setting remains problematic. This is largely because designs have been evaluated under specialized assumptions on diverse simulation platforms, and because evaluating the behavior of an implementation has relied on performance comparisons against locks or against variants of the same design, rather than against representatives of distant points in the design space. The goal of this chapter is to address this difficulty by bringing diverse designs together in one simulation platform, allowing their behavior to be examined side by side. We provide an evaluation and comparison of different TM-aware coherence protocols and HTM designs from the literature including MetaTM [70, 75], LogTM [58], and Sun's Rock [22].

6.1 Methodology

All work in this comparison is based on the MetaTM 2.0 framework [70, 75], which runs in the Simics machine simulator [47] (all experiments were run with version 3.0.31). The **XMESI** protocol is implemented in a separate Simics module. We simulated both CMP and SMP organizations using 16 and 32 cpus, and ran both the

Name	**coh**	**C-D**	**V-M**	**OV**
MetaTM	**XMESI**	eager, L1 cache	eager, L1 cache	SLE
Lazy	**XMESI**(*)	lazy, L1 cache	eager, L1 cache	SLE
xRock	MESI	eager	eager, store buffer	SLE
TagTM	**XMESI**(**)	eager	eager, L1 cache	SLE
xLogTM	**MESI**	eager, perfect signatures	eager, logging	N/A

Figure 6.1: Descriptions of the HTM design models evaluated. An overflow strategy of SLE means speculative lock elision: if a transaction overflows the transaction is rolled back and a lock is acquired instead using mechanisms to revert to exclusion described in [75]. While Lazy, TagTM, and MetaTM all use **XMESI**, only TagTM actually uses support for notifying transactions.

TxLinux kernel benchmarks from MetaTM [70], as well as a subset of the STAMP benchmark suite [54]. The machine parameters and command lines are detailed in table 6.1.

6.1.1 HTM models

The approach in this study relies on implementing different design models by making different modifications to the MetaTM HTM platform [70]. MetaTM was modified to support different version management mechanisms (logging, store-buffer, and L1 caches), and different coherence protocols (**XMESI**, **TMESI** [87], and MESI). With the exception of designs that rely on logging for version management, all models considered are "best-effort" designs with no explicit mechanism for handling failure due to overflowing transactional hardware. For these designs, we modify `cxspinlocks` to take a strategy similar to speculative lock elision [66]. If a transaction overflows in a best-effort design, the system rolls back and takes a lock whose acquisition is arbitrated by the contention manager. We modified TxLinux cxspinlocks to retry with a lock on overflow, and modified the STAMP benchmarks [54] to use a single global cxspinlock to handle overflow cases. We implemented a locking version of the STAMP benchmarks as well, making it possible to compare all designs against locks. Note also that many kernel transactions conditionally perform I/O, requiring all HTM models to support cxspinlocks (see chapter 3). For all HTM models, when transactions are executed in the kernel, the transaction must abort and re-execute with mutual exclusion if I/O occurs. While use of logging avoids the need for an elision-based approach to overflow, it does not avoid the need for elision in the presence of I/O. To handle these cases we enhance all designs considered to support the `xcas` and `xtest` instructions required for cxspinlocks. Table 6.1 summarizes the HTM models considered in this chapter.

The baseline model in this study, MetaTM, is an **XMESI**-based design. The design is similar to TagTM in that the model takes advantage of the transaction annotation features supported by **XMESI** to avoid trapping to software to handle contention, but does not rely on notifying transactions for notification. We also include TagTM in the study.

To simulate behavior of a log-based implementation such as LogTM [58] (eager versioning, eager conflict detection) or TokenTM [12], MetaTM is modified to log writes to a per-thread undo log, and TxLinux was modified to coordinate setup and management of log areas when new threads are created. Aborts costs are modeled by actually walking the log and replacing old values in hardware, incurring the full memory latency of the walk, and stalling the local processor until the abort completes; this approach neglects latency for instructions for variations of the design that walk the log in software. The implementation models LogTM-SE [94] with perfect signatures: the simulation uses unmodified MESI for cache coherence, and conflicts are detected in the simulator. Additionally, conflict resolution is performed in the simulator, so no cost for contention management handlers is modeled. Because this model incurs no overhead for contention management (unlike other models in this study) and has no false conflicts, this models best case performance for a log-based design. This model, called xLogTM, relies on a store buffer to help tolerate log write latency, as described in [58].

To simulate the behavior of Sun's Rock, MetaTM was augmented with a

	Configuration
Processor	Pentium-4-like x86 instruction set, 3 GHz, 1 IPC
Store buffer	32 entries, 1 cycle access
L1	separate i+d, 32 KB, 4-way, 64-byte line, lru replacement, 3-cycle access
L2	unified 4 MB 8-way, 64-byte line, 12-cycle access
Memory	1GB capacity, 200 cycle access time.
Memory Hierarchy	SMP: Each core has a private L2. CMP: all cores share a single L2.
TM parameters	Timestamp contention management, linear backoff policy, cache line granularity conflict detection.

Benchmark	*Description*
config, pmake, mab	These benchmarks report transactions created in TxLinux, a Linux-variant operating system with several subsystems converted to use transactions for synchronization, instead of spinlocks [70, 75]. The workload involves several user-mode applications (configure, make, modified andrew benchmark) which are running on the transactional OS.
bayes	learns the structure of a Bayesian network "-v32 -r4096 -n5 -p30 -s1 -i2 -e4 -t CPUs"
genome	a gene-sequencing bioinformatics application, "-g16384 -s32 -n4194304 -t CPUs"
intruder	signature-based intrusion detection system "-a10 -l32 -n65216 -s1 -t CPUs"
kmeans	implements a K-means clustering algorithm, " -m40 -n40 -t0.0009 -i random-n65536-d32-c16.txt -p CPUs"
vacation	models a multi-user database, "-c CPUs -n4 -q60 -r1048576 -u90 -t1048576"
yada	Delaunay mesh refinement "-a20 -i ttimeu10000.2"

Table 6.1: The upper table represents architectural parameters of simulated machines. The TM parameters were held constant across all HTM designs. The lower table shows workloads used in evaluation. TxLinux and list are kernel-mode transactions, while the other benchmarks run in user-mode. All benchmarks use a number of threads equal to the number of processors, unless noted otherwise.

transactional store buffer. The default size is 32 entries, but we also experiment with different buffer sizes as well as an unbounded mode, enabling us to gauge the potential profitability of dedicating more resources to speculative writes. This model is called xRock. The xRock model aborts the current transaction if the store buffer fills, lines read in a transaction are evicted from an L1 cache, or if a TLB miss occurs, as described in [22]. Rock's TM support is subject to a number of other exotic abort

bnc	**tx**	**variable**	**tx-len**	**contention**
bayes	1600	yes	long	10%
genome	919,276	no	medium	1-2%
kmeans	5400000	yes	short	10-20%
intruder	1587019	no	medium	40-50%
vacation	1048576	no	medium	30-50%
yada	63616	yes	long	45-50%
config	4500000	yes	short	1-3%
mab	4200000	yes	short	0-1%
pmake	190000	yes	short	3%

Table 6.2: HTM comparison benchmark characterization.

conditions such as function calls and other "difficult instructions", which this model does not attempt to emulate, making xRock an optimistic characterization of Rock's performance.

The Lazy model is a lazy-lazy system similar to TCC [29], without TCC's requirement that all memory traffic be transactional. We modified MetaTM to support a lazy conflict detection mode. Version management is still handled by the L1 cache. [1] Support for lazy conflict detection required some minor modifications to **XMESI**, allowing it to take note of conflicts but elide state transitions for them, and allowing it to violate the **XMESI** invariant against having multiple outstanding copies in **TMM** state.

6.2 Workload characterization

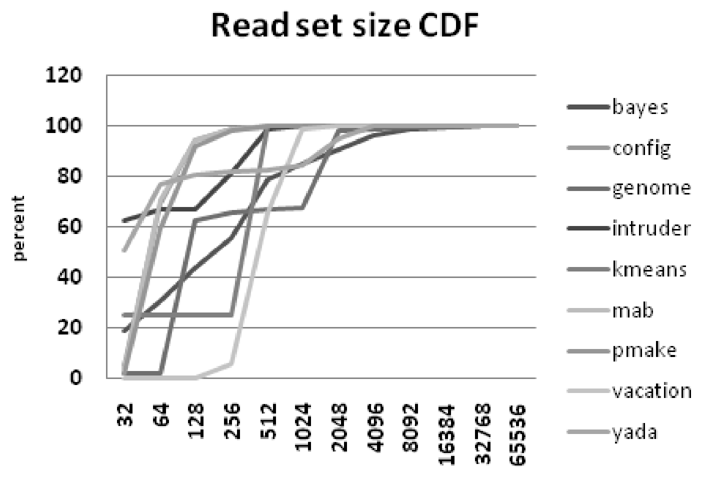
Table 6.2 provides an overview of transactional workload characteristics, and Figures 6.2 and 6.3 summarize read and write set sizes for the workloads examined in this study. Read-write set histograms were captured on a simulated 16 cpu CMP machine using the MetaTM HTM model running with idealized (unbounded) space

[1]The notion that buffering writes in an L1 cache constitutes eager or lazy version management is a matter of semantic taste. Writes are buffered in an L1 cache, and must be published on commit through a coherence state transition: from this perspective, this is lazy version management. On the other hand, writes are performed on the cache lines that will become the globally committed if the transaction succeeds, which is fundamentally eager.

		bayes	**genome**	**intruder**	**kmeans**	**vacation**	**yada**	**config**	**mab**	**pmake**
≤32	R	19.2	2.5	62.6	25.0	-	51.1	5.4	4.2	2.4
	W	19.6	2.2	33.3	25.0	-	41.0	18.1	32.5	38.1
≤64	R	11.5	0.1	4.1	-	-	25.7	64.1	66.3	57.1
	W	11.5	1.7	33.3	-	0.1	10.1	69.1	55.3	45.3
≤128	R	13.2	60.1	0.6	-	-	4.2	24.8	23.8	32.7
	W	15.7	95.7	0.5	-	5.8	30.6	7.0	5.6	6.5
≤256	R	12.0	3.5	14.3	-	5.9	0.9	4.6	4.3	5.8
	W	12.5	0.3	32.6	75.0	88.2	0.2	4.9	5.4	7.4
≤512	R	22.9	1.0	17.3	75.0	59.9	0.6	0.8	1.3	1.9
	W	4.3	0.1	0.2	-	5.9	2.3	0.7	1.2	2.6
≤1024	R	6.3	0.8	1.1	-	32.8	1.8	-	-	0.1
	W	11.7	-	-	-	-	1.2	-	-	0.1
≤2048	R	5.9	30.4	-	-	1.5	10.8	0.1	-	-
	W	7.1	-	-	-	-	0.3	0.1	-	0
≤4096	R	5.5	0.5	-	-	-	4.8	-	-	-
	W	5.9	-	-	-	-	14.0	-	-	-
≤8192	R	2.2	0.6	-	-	-	-	0.2	-	-
	W	1.6	-	-	-	-	0.3	-	-	-
≤16384	R	0.7	0.3	-	-	-	-	-	-	-
	W	5.4	-	-	-	-	-	0.1	-	-
≤32768	R	0.7	0.3	-	-	-	-	-	-	-
	W	1.6	-	-	-	-	-	-	-	-
≤65536	R	-	-	-	-	-	-	-	-	-
	W	2.9	-	-	-	-	-	-	-	-

Figure 6.2: Read/Write set characterizations for benchmarks evaluated in this study, captured using MetaTM on a 16 cpu CMP. Sizes are in bytes.

for conflict detection and version management. Minor variations should be expected for TxLinux kernel benchmarks, which execute many transactions in interrupt handlers, and the STAMP benchmarks `kmeans` and `bayes`, which are written to execute a variable number of transactions. Each cell in the table indicates the percentage of transactions for which the read or write set falls into the size bin for that row. Read/write sizes are in bytes. The graphs depict the cumulative distribution function (CDF) for read set and write set sizes per benchmark respectively. RW-set sizes are particularly useful for understanding the behavior of the "best-effort" models, as these HTMs must abort and serialize transactions whose RW-set sizes exceed the hardware resources dedicated to the TM subsystem.



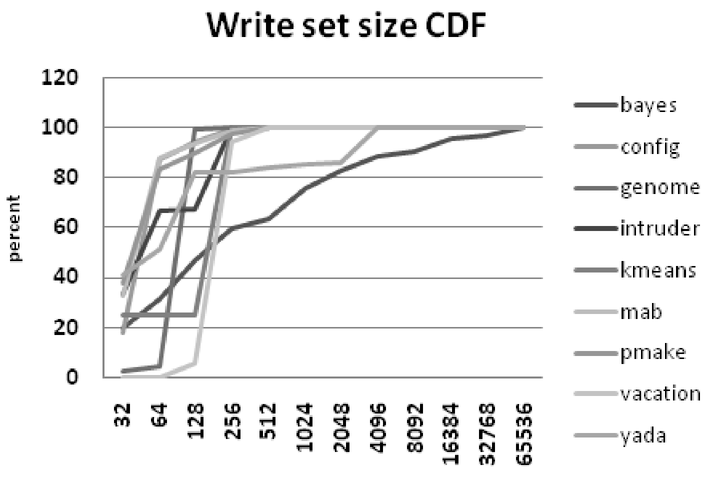


Figure 6.3: Read/Write set size CDFs for benchmarks evaluated in this study, captured using MetaTM on a 16 cpu CMP. Sizes are in bytes.

To understand the performance characteristics of the various HTM designs, we consider scalability (measured as speedup over sequential), the single-thread overhead introduced by each design, and the comparative performance of each design on each workload given the same cpu budget. Figures 6.5 and 6.6 show speedup over sequential execution (1-cpu w/locks) for all benchmarks on CMP and SMP respectively. Figure 6.7 shows speedup per benchmark, HTM design and memory organization over a lock-based implementation with the same CPU count and memory organization. Figure 6.4 shows single thread overhead for each HTM design on a representative subset of benchmarks. The performance of each design is largely determined by how much work is wasted restarting failed transactions either because of contention, exhausted hardware resources, I/O, or exceptions. Tables 6.4, 6.3, and 6.5 give the raw execution time along with statistical characterization of restart behavior for all benchmarks. Data are shown for 16 cpu and 32 cpu CMP and SMP machines, for all the basic HTM designs in the study, as well as locks, respectively.

6.3 Single thread overhead

Single thread overhead for HTM designs is measured by allowing the HTM to execute critical sections transactionally despite the lack of parallel threads with which to conflict. As a result, "best-effort" designs (Rock, LazyTM, MetaTM, and TagTM) have a non-zero abort rate due to transactions that overflow and retry. This effect is particularly visible on benchmarks with large transactions such as `genome`, where overflows are common and cause transactions to suffer a 20% loss with respect to locks. Benchmarks with minimal aborts due to overflow cannot, in general, contend on a single cpu, bringing performance for best-effort designs much closer to that of locks.

Kernel benchmarks such as `pmake` have non-zero abort rates due both to I/O (which necessitates restarts), and to the possibility of same-cpu conflicts introduced by the use of transactions in interrupt handlers. For this reason, single-thread overheads for all designs are high in kernel benchmarks, ranging from 14 to 34%. Rock and LogTM are the designs most challenged by kernel benchmarks. Rock must contend with both TLB miss restarts and I/O restarts, while LogTM's additional overhead for log-walks on aborts makes similar restart rates more expensive.

The single-thread overhead of LogTM in user benchmarks is notable as well.

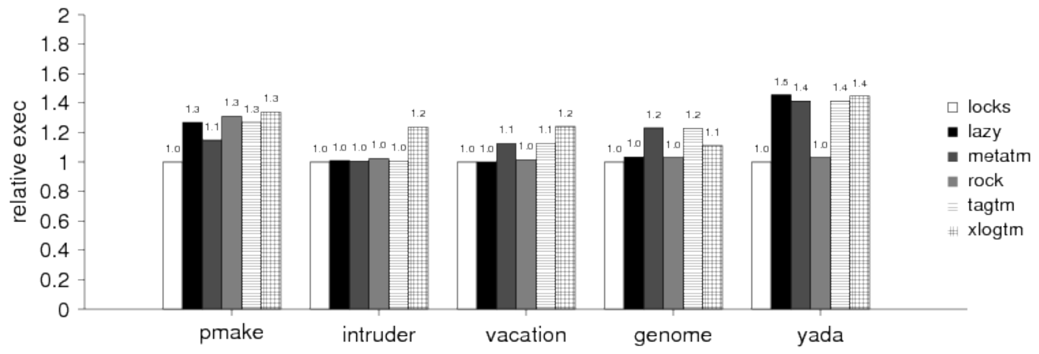


Figure 6.4: Single thread overhead (execution time on 1 cpu relative to 1 cpu locks) for HTM designs.

To put LogTM's overhead in perspective, consider `intruder`, which has minimal overflow rates for cache-based HTMs. The lack of contention on a uniprocessor, combined with the ability to commit transactions in cache allows the best-effort HTMs to have negligible overhead (1-2%) with respect to locks. In contrast, LogTM shows a 24% slowdown with respect to locks, despite encountering only 1 restarting transaction (caused by a context switch). `Intruder` spends 67% of it's execution time in transactions, and the additional cost of log writes put additional pressure on the L1 cache, because LogTM's log is in cachable virtual memory, creating a source of cache pressure not present in other designs. A similar effect is visible in `yada`, which has no restarts whatsoever, but spends 99.98% of it's execution in transactions. Yada's relatively larger write-sets (ranging from 32 bytes to 8KB) introduce significant additional overhead, and the L1 cache miss rate triples (from 0.4% for locks to 1.2% for LogTM). The result is a 44% performance loss on a single cpu. However, the cache-based designs introduce similar overhead for `yada` by requiring re-execution of critical sections that fail due to overflow. An interesting exception in this field is Rock, which runs only 2.9% slower than locks. 100% of transactions fail due to TLB misses in this case, and these failures occur very early

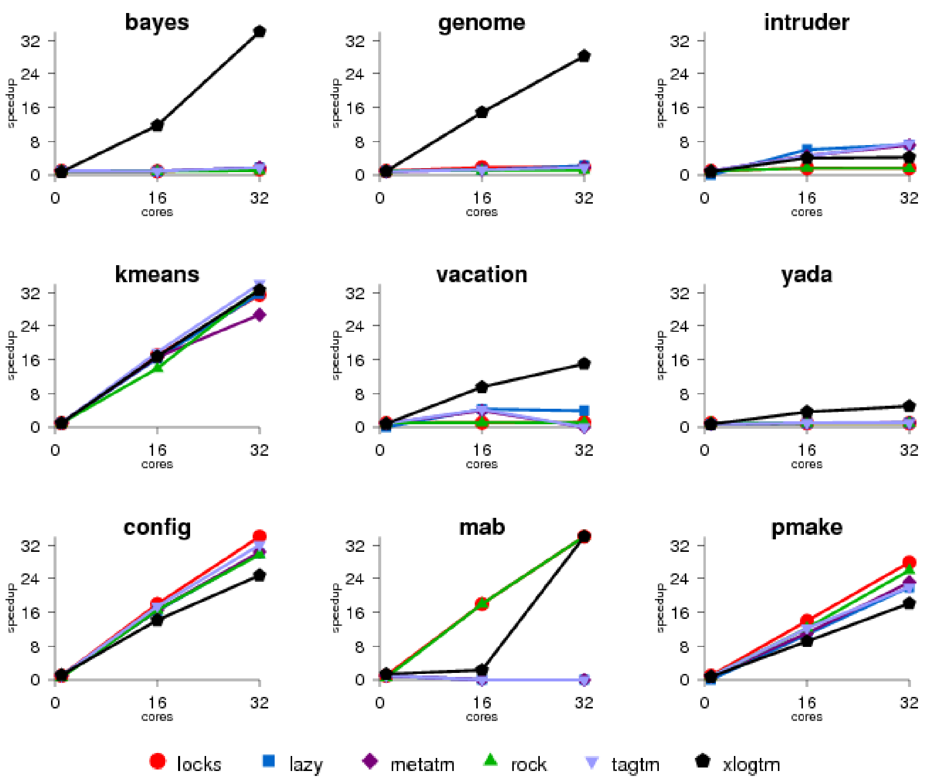


Figure 6.5: Speedup over 1 cpu locking for 1, 16, and 32 cpu CMP designs for all HTM models and locks.

in the transactions allowing Rock to introduce considerably less latency for failed speculation than the the other HTMs.

6.4 Comparing concurrent execution

The data for multiple CPU execution reveal that while some benchmarks have fundamentally poor scalability, the performance characteristics for benchmarks without such limitations differ dramatically across designs, and to a lesser degree across memory organizations. Performance tends to be determined by contention and/or overheads that derive from a design's affinity for the transaction sizes that characterize a workload. The tradeoff between the overheads associated with supporting

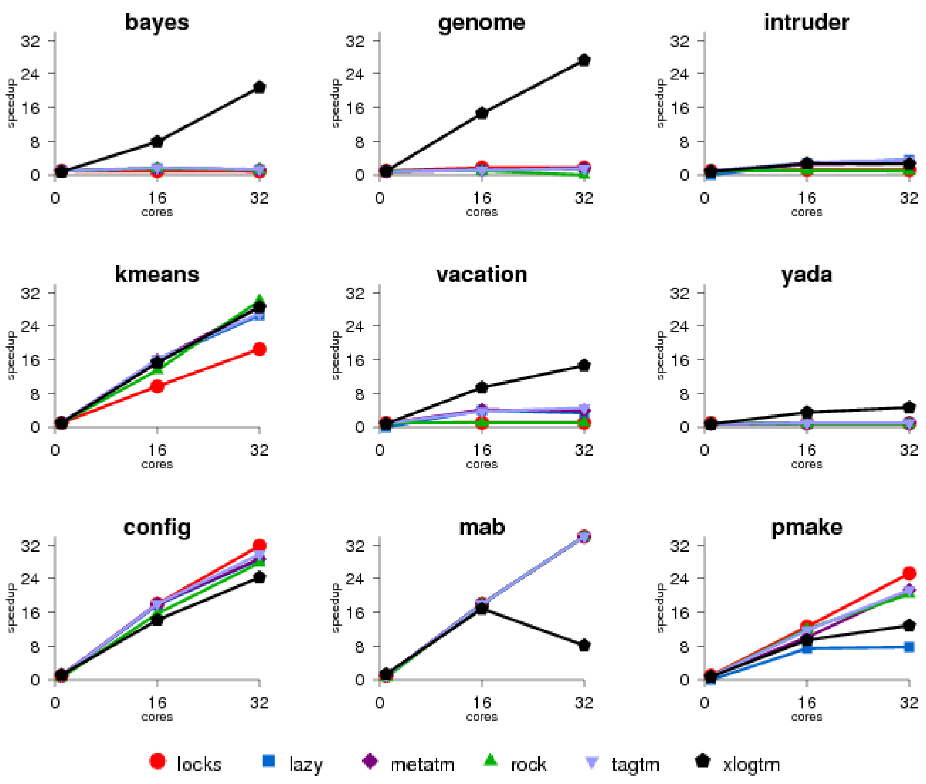


Figure 6.6: Speedup over 1 cpu locking for 1, 16, and 32 cpu SMP designs for all HTM models and locks.

large or unbounded transactions, and the performance penalties associated with eliding such support are very much in evidence.

Sources of scalability limitation fall roughly into three categories, in rough order of significance: overflow, contention, and fixed overheads for supporting the TM abstraction. Rock introduces the least overhead for TM support, dedicating only 32 store-buffer entries for speculative writes, and making no effort to support transactions that survive exceptions (or function calls). The cache-based HTMs (Lazy, MetaTM, and TagTM) rely on L1 versioning, incurring additional latency for speculation and additional cache misses associated with lost conflicts or publishing committed data. In contrast to Rock, the cache-based designs can support a much larger (but still limited) range of transaction sizes, effectively trading modest additional overhead for larger transaction support. LogTM has the largest common case overhead because it relies on (cachable) main memory for versioning. This design decision results in additional latency for writing to log areas and log-walking on abort, but affords workloads unbounded transaction sizes. Because the "best-effort" HTM designs must rollback and revert to mutual exclusion when hardware resources are exhausted, the ability of the HTM to provide good scalability for a workload is largely determined by transaction sizes and the ability of the HTM to support transactions that survive TLB misses and function calls.

Rock

Rock is crippled by aborts from TLB misses. In the STAMP benchmarks, Rock's restart rates are generally in the 50% to 100% range (with the exception of `kmeans`, which has a restart rate around 10%). For the vast majority of the benchmarks, TLB misses are responsible for close to 100% of restarts. Combined with the use of a single global lock to handle fall-back for failed transactions, the result is that Rock cannot leverage 16 or 32 cpus to outperform the single-cpu execution for `vacation` and `yada`, and does not scale beyond 2× speedup for `intruder`, `bayes`, and `genome`. The `kmeans` benchmark is the only user-mode benchmark in this study that is a good fit for Rock: `kmeans` has relatively milder contention and small transactions, allowing it to scale well on Rock. Because Rock's overhead for successful transactions is low compared to other HTM designs, it competes with the other "best effort" designs and outperforms LogTM for 32 cpu CMP and SMP machines running `kmeans`.

Rock is a much better fit for the TxLinux kernel benchmarks than for the the STAMP benchmarks. TxLinux transactions are (with very few exceptions) converted spinlock critical sections from Linux 2.6.16, resulting in small transactions (100s of instructions) and low conflict rates (1% to 5%). While the STAMP benchmarks handle overflow with a single global cxspinlock, overflows in TxLinux can

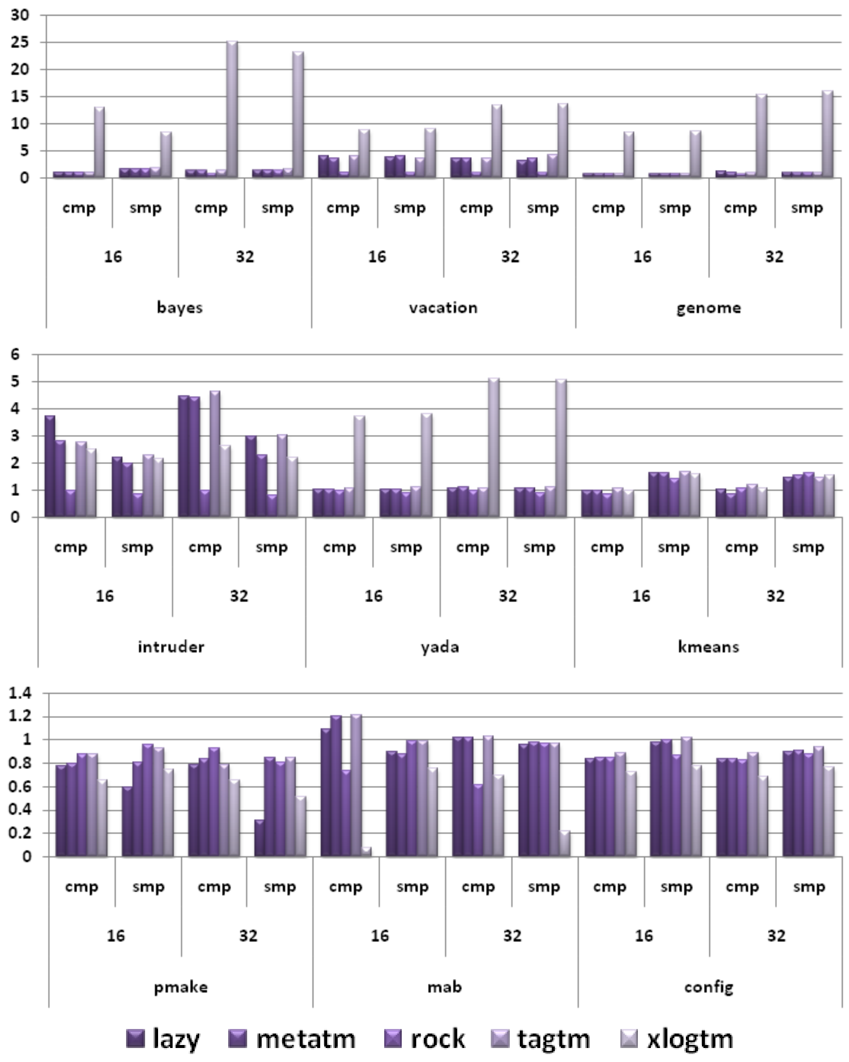


Figure 6.7: Speedup over over a lock-based implementation using the same number of cpus on the same memory organization.

default to the cxspinlocks derived from the original locking code. As a result, overflowed transactions in the kernel benchmarks serialize with much lower frequency; overflows rarely result in lock-based re-executions that contend for the same lock. Rock experiences much rarer overflow-based restart (less than 1%) and competes with other best-effort designs (beating cache-based designs for 32 cpu `pmake` and consistently outperforms LogTM for all kernel benchmarks. However, TLB misses still result in a 10% increase in restart rate over other HTM designs for these benchmarks. Section 6.4.4 explores a "Rock+" model with larger store-buffers and support for transactions that survive TLB misses.

6.4.1 LogTM

The LogTM design represents the opposite end of the spectrum from Rock, in that it trades higher fixed overheads for TM support in exchange for supporting unbounded transactions. For the STAMP benchmarks that execute a significant number of large transactions, LogTM outperforms all other designs by multiple integer factors. For example, `genome` transactions are large enough to cause 100% in the cache-based designs and Rock, but has very low contention (1% conflict-based restart rate). In this case, LogTM achieves near perfect scalability (15× on 16 cpus and 27-28× on 32 cpus), while none of the best-effort HTMs surpass a 2× speedup. The `vacation` benchmark represents a more nuanced scenario: The 1-15% overflow-based restart rate for the Lazy, MetaTM, and TagTM designs combined with a high baseline of contention (40% on 16 cpus and 55% on 32) limit the cache-based designs to a ceiling of 4× speedup for both 16 and 32 cpu machines. LogTM is not forced to serialize to support overflows, and achieves 9× for 16 cpu machines and nearly 16× speedup on 32 cpu machines. LogTM's more expensive conflict handling does limit its performance relative to the best-effort designs, and this effect is visible in cases where overflow is rare or non-existent. An increase in the rate of aborts causes the high cost of log-walking on abort to contribute significantly to execution time. For example, `intruder` has small transactions and high contention: MetaTM, Lazy, and TagTM outperform LogTM by as much as 50% for this workload.

On TxLinux kernel benchmarks, LogTM performs on average worse than the best-effort designs. TxLinux has both rare conflict and rare overflow. As a result, LogTM performance is impacted somewhat less dramatically by abort overhead, and

bnc	design	exec		rstpct		rst/tx		conf	ov	tlb			
intruder	locks	0.668	0.895										
	lazy	0.151	0.301	49	51	3	9	100	0.3	-	100	0.2	-
	metatm	0.153	0.395	52	48	7	28	100	0.2	-	100	0.1	-
	tagtm	0.144	0.297	53	50	9	55	100	0.2	-	100	0	-
	rock	0.681	1.110	91	59	2	1	0	0	100	3	0	97
	xlogtm	0.257	0.413	42	46	2	12	100			100		
vacation	locks	4.101	4.201										
	lazy	1.182	1.368	67	65	5	5	94	6	-	95	5	-
	metatm	0.871	1.168	88	88	17	17	99	1	-	99	1	-
	tagtm	0.851	1.108	88	88	17	17	99	1	-	99	1	-
	rock	4.311	4.542	100	100	1.0	1.0	0	0	100	0	0	100
	xlogtm	0.307	0.315	56	55	2	2	100			100		
genome	locks	2.615	2.811										
	lazy	2.190	2.982	19	20	0.2	0.2	0.4	100	-	4	96	-
	metatm	2.654	2.955	23	25	0.2	0.3	2	98	-	2	98	-
	tagtm	2.719	3.265	24	27	0.2	0.3	2	98	-	2	98	-
	rock	4.413	4.633	100	99	1.0	1.0	0	0	100	0	0	100
	xlogtm	0.171	0.178	1	1	0	0	100			100		
yada	locks	0.411	0.433										
	lazy	0.384	0.411	48	49	6	9	97	3	-	98	2	-
	metatm	0.381	0.407	47	49	167	236	100	0	-	100	0	-
	tagtm	0.383	0.398	48	48	639	804	100	0	-	100	0	-
	rock	0.419	0.485	97	68	1	0.8	0	0	100	0	0	100
	xlogtm	0.081	0.086	13	16	5	5	100			100		
kmeans	locks	0.858	0.860										
	lazy	0.852	1.014	16	11	0.2	0.1	100	0	-	100	0	-
	metatm	1.009	0.942	20	12	1	1	100	0	-	100	0	-
	tagtm	0.717	0.997	20	12	3	0.9	100	0	-	100	0	-
	rock	0.820	0.897	9	11	0.4	1	95	0	5	98	0	2
	xlogtm	0.828	0.945	7	14	0.4	1.0	100			100		
bayes	locks	0.279	0.426										
	lazy	0.373	0.317	79	71	5	3	62	0.2	-	61	0.3	-
	metatm	0.173	0.307	79	71	5	3	65	0.2	-	53	0.4	-
	tagtm	0.158	0.291	79	71	3	2	42	0.3	-	42	0.5	-
	rock	0.373	0.317	79	71	9	3	80	0.1	20	53	0.4	46
	xlogtm	0.012	0.019	10	11	6	8	100			100		

Table 6.3: Performance and characterization of STAMP benchmarks 32 cpu CMP and SMP. In each column, the leftmost figure corresponds to a machine with a CMP hierarchy, and the rightmost column indicates an SMP. User is the user time in seconds. Restarts indicates the total number of restarts occurring during the benchmark. The **tlbm rst** and **ovrst** figures indicate the number of transactions that restarted due to TLB misses and overflows respectively.

more dramatically by it's higher fixed overhead for successful transactions. However, in the kernel benchmarks, use of cxspinlocks introduces I/O as another source of restart.

bnc	**design**	**exec**		**rstpct**		**rst/tx**		**conf** | **ov** | **tlb**	
intruder	locks	0.665	0.846						
	lazy	0.180	0.385	43	53	2	4	98 | 2 | -	98 | 2 | -
	metatm	0.237	0.431	53	52	9	20	99 | 0.9 | -	100 | 0.4 | -
	tagtm	0.241	0.375	54	53	26	63	100 | 0.3 | -	100 | 0.1 | -
	rock	0.677	1.019	92	57	1	0.7	0 | 0 | 100	0 | 0 | 100
	xlogtm	0.269	0.395	32	45	1	5	100	100
vacation	locks	4.278	4.382						
	lazy	1.071	1.147	55	53	2	2	86 | 14 | -	87 | 13 | -
	metatm	1.180	1.124	75	75	7	7	96 | 4 | -	97 | 3 | -
	tagtm	1.102	1.223	74	75	9	9	97 | 3 | -	97 | 3 | -
	rock	4.267	4.515	100	100	1	1.0	0 | 0 | 100	0 | 0 | 100
	xlogtm	0.484	0.490	38	37	0.9	0.8	100	100
genome	locks	2.683	2.779						
	lazy	3.945	4.130	33	33	0.3	0.3	0 | 100 | -	0 | 100 | -
	metatm	3.941	4.113	33	33	0.3	0.3	0.5 | 99 | -	0.3 | 100 | -
	tagtm	3.997	4.111	33	33	0.3	0.3	0.3 | 100 | -	0.2 | 100 | -
	rock	4.457	4.606	100	99	1.0	1.0	0 | 0 | 100	0 | 0 | 100
	xlogtm	0.325	0.330	0.6	0.6	0	0	100	100
yada	locks	0.409	0.431						
	lazy	0.403	0.416	47	48	3	4	94 | 6 | -	95 | 5 | -
	metatm	0.402	0.421	47	48	94	122	100 | 0.2 | -	100 | 0.1 | -
	tagtm	0.389	0.399	47	47	2399	3044	100 | 0 | -	100 | 0 | -
	rock	0.420	0.484	98	67	1	0.7	0 | 0 | 100	0 | 0 | 100
	xlogtm	0.111	0.114	12	13	5	5	100	100
kmeans	locks	1.574	2.781						
	lazy	1.649	1.704	16	7	0.2	0	100 | 0 | -	100 | 0 | -
	metatm	1.623	1.699	19	7	1	0.5	100 | 0 | -	100 | 0 | -
	tagtm	1.513	1.669	20	7	3	0.5	100 | 0 | -	100 | 0 | -
	rock	1.931	1.991	8	6	0.3	0.5	95 | 0 | 5	97 | 0 | 3
	xlogtm	1.605	1.762	6	8	0.4	0.5	100	100
bayes	locks	0.424	0.409						
	lazy	0.354	0.230	89	80	3	1	49 | 0.1 | -	7 | 0.5 | -
	metatm	0.344	0.220	89	80	3	1	49 | 0.1 | -	7 | 0.5 | -
	tagtm	0.324	0.210	89	80	3	1	49 | 0.1 | -	7 | 0.5 | -
	rock	0.424	0.250	89	80	3	1	49 | 0.1 | 51	7 | 0.5 | 92
	xlogtm	0.033	0.049	7	10	13	17	100	100

Table 6.4: Performance and characterization of STAMP benchmarks 16 cpu CMP and SMP. In each column, the leftmost figure corresponds to a machine with a CMP hierarchy, and the rightmost column indicates an SMP. User is the user time in seconds. Restarts indicates the total number of restarts occurring during the benchmark. The **tlbm rst** and **ovrst** figures indicate the number of transactions that restarted due to TLB misses and overflows respectively.

Because LogTM transactions are being used in this case to support cxspinlocks–LogTM transactions must abort for I/O as well as contention. The best effort HTMs out-perform LogTM by speedup margins that range from 10% (`config`) to 60%

(`pmake`, 32 cpu SMP). It also worth noting that *none* of the HTM designs outperform locks on the TxLinux kernel benchmarks. Because OS critical sections are small and optimized to avoid contention the real argument for HTM in the Linux 2.6 kernel revolves not around minimal-effort scalability improvement, but around code simplification. From this perspective, better performance from best effort designs does not necessarily make the case that LogTM is a poor fit for kernel transactions.

The LogTM model also experiences some pathologies executing the `mab` benchmark: while it is generally within 10% of the performance of the other HTM models for kernel benchmarks, xLogTM has 5× slowdown for 32 cpu SMP and 10× slowdown for 16 cpu CMP on `mab`. In both cases contention is the culprit. In the 16 cpu case, only 60,796 transactions restart, but they restart repeatedly for a cumulative 913,275 total restarts. The `vma_adjust` function is called when a process' address space must be modified either for instance when expanding the area mapped by a vma, or when calling `mprotect`. Restarts for transactions started in `vma_adjust` account for over 24 million wasted cycles alone. Another culprit is the `do_notify_parent` function, which wastes over 18 million cycles due to I/O restarts that occur in nested transactions. Repeated restarts in LogTM are expensive due both to the additional cost of rolling back changes from the log, but because of growing backoff times: a very few transactions restart enough times that their backoff times between subsequent attempts (using an exponential backoff policy) become very large, exceeding 200,000 cycles and in one case exceeding 40,000,000 cycles. The result is severely limited forward progress and pathological performance. The critical sections involved in the 32 cpu case are similar. LogTM could likely benefit significantly in these cases from using notifying transactions.

Because any realizable implementation of LogTM-SE will use hardware signatures that can saturate, these empirical data underestimate the impact of false conflicts for LogTM running transactions that overflow. The approximations are likely reasonable on the assumption that a log-based implementation would more likely resemble TokenTM [12] than LogTM-SE [94]. TokenTM presents an alternative log-based design relying on per cache-line token meta data for all of physical memory. The technique allows a log-based implementation to get within 8% of the performance of perfect signatures in the worst case on `genome`. In the absence of some mechanism such as TokenTM to ameliorate the affects of saturation, false conflicts that result from limited signature size can have a significant impact: `delaunay` (removed from the current version of STAMP) slows down by a factor of 9$\times$ with realizable signatures. Yen et al. characterize the performance impact of limited signature size in greater detail [95], and show that even with significant improvements in signature-representation techniques, limited signature size eventually throttles performance for benchmarks with large transactions. For example, the difference between 1k and 2k signature bits results in a 2$\times$ performance delta on `labyrinth`, and `bayes` requires 16-32k bit signatures to achieve scalable performance.

6.4.2 Cache-based designs

The Lazy, MetaTM, and TagTM designs represent a different set of tradeoffs in the design space from Rock or LogTM. By using caches for version management, these designs incur higher overhead than Rock, but still encounter hard upper bounds on transactions sizes stemming from associativity and capacity evictions of transactional from L1 caches. Since all three designs use the same cache geometries, the primary differences between the models is different approaches to contention. The Lazy model detect conflicts at commit time while TagTM and MetaTM both detect conflicts early. Lazy conflict detection can benefit some workloads by avoiding restarts for conflicts that may resolve by commit time, but can harm performance for some workloads by increasing the latency from the time a conflict occurs to the time it is resolved. TagTM and MetaTM both use the **XMESI** protocol, and so differ in this study only by TagTM's use of notifying transactions, allowing TagTM

P	bnc	design	exec		rstpct		rst/tx		**conf** | **ov** | **tlb**	
32	pmake	locks	0.005	0.006						
		lazy	0.006	0.017	3	4	0.2	1	100 | 0 | -	100 | 0 | -
		metatm	0.006	0.007	3	3	0.4	0.4	100 | 0 | -	100 | 0 | -
		tagtm	0.006	0.007	3	2	0.2	0	100 | 0 | -	100 | 0.1 | -
		rock	0.005	0.007	17	16	0.2	0.6	28 | 0.5 | 71	74 | 0.2 | 26
		xlogtm	0.008	0.010	3	3	0.2	0.9	100	100
	mab	locks	0.072	0.107						
		lazy	0.126	0.121	22	1	0.2	0	100 | 0.1 | -	99 | 0.6 | -
		metatm	0.119	0.120	22	1	0.2	0	100 | 0.1 | -	99 | 0.6 | -
		tagtm	0.106	0.109	22	1	0.2	0	100 | 0.1 | -	99 | 0.6 | -
		rock	0.116	0.119	15	1	0.2	0	10 | 0.1 | 90	0 | 0.6 | 99
		xlogtm	0.103	0.495	2	1	0	0.2	100	100
	config	locks	0.134	0.152						
		lazy	0.160	0.169	1	2	0	0	100 | 0 | -	100 | 0 | -
		metatm	0.157	0.169	1	2	0	0	100 | 0 | -	100 | 0 | -
		tagtm	0.153	0.169	1	2	0	0	100 | 0 | -	100 | 0 | -
		rock	0.163	0.174	12	10	0.2	0.2	1 | 0.7 | 98	26 | 0.6 | 73
		xlogtm	0.195	0.219	3	3	0.2	0.2	100	100
16	pmake	locks	0.010	0.011						
		lazy	0.012	0.018	3	3	0.3	0.4	100 | 0 | -	100 | 0 | -
		metatm	0.012	0.013	3	3	0.2	0.3	100 | 0.1 | -	100 | 0 | -
		tagtm	0.011	0.011	3	3	0.1	0	100 | 0.3 | -	100 | 0.2 | -
		rock	0.011	0.011	21	21	0.3	0.3	33 | 0.5 | 67	27 | 0.5 | 73
		xlogtm	0.015	0.014	3	3	0.3	0.4	100	100
	mab	locks	0.122	0.180						
		lazy	0.166	0.209	1	1	0	0	99 | 1.0 | -	99 | 1 | -
		metatm	0.155	0.205	1	1	0	0	99 | 1.0 | -	100 | 0.3 | -
		tagtm	0.145	0.195	1	1	0	0	99 | 1.0 | -	99 | 1 | -
		rock	0.165	0.407	16	16	0.2	0.2	0 | 0.1 | 100	0 | 0.1 | 100
		xlogtm	1.749	0.240	2	1	0.3	0	100	100
	config	locks	0.245	0.264						
		lazy	0.281	0.293	1	1	0	0	95 | 5 | -	97 | 3 | -
		metatm	0.279	0.283	1	1	0	0	95 | 5 | -	97 | 3 | -
		tagtm	0.269	0.273	1	1	0	0	95 | 5 | -	97 | 3 | -
		rock	0.291	0.307	11	11	0.1	0.2	0 | 0.8 | 99	0 | 0.7 | 99
		xlogtm	0.340	0.340	2	2	0	0.1	100	100

Table 6.5: Performance and characterization of TxLinux benchmarks on CMP and SMP machines. In each column, the leftmost figure corresponds to a machine with a CMP hierarchy, and the rightmost column indicates an SMP. User is the user time in seconds. Restarts indicates the total number of restarts occurring during the benchmark. The **tlbm rst** and **ovrst** figures indicate the number of transactions that restarted due to TLB misses and overflows respectively.

to avoid the use of backoff heuristics. This study does not model memory bandwidth constraints. Since notifying transactions has the most benefit for high contention benchmarks under constrained bandwidth, it's benefits are less pronounced, but

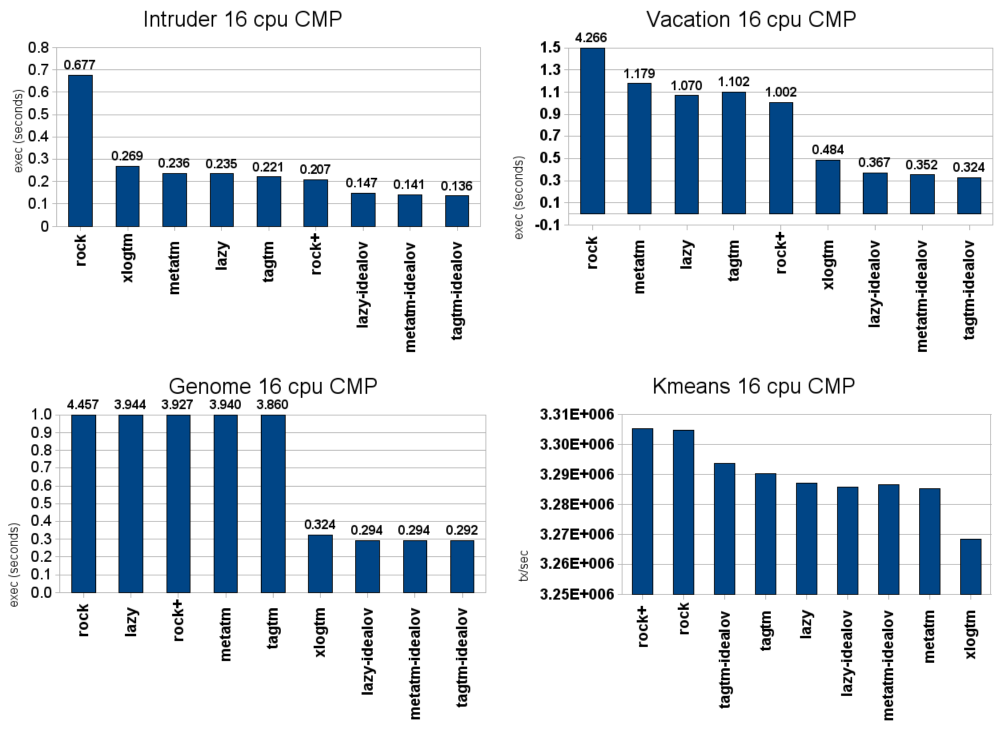


Figure 6.8: Runtime for a selection of STAMP benchmarks across several design points, including idealized overflow handling. Lower is better in all cases except for `kmeans`, where data are shown as throughput in transactions per second because `kmeans` has a variable number of transactions from run to run. All data are for 16 cpu CMP machines.

tangible in this environment.

Like Rock, the cache-based designs take profound performance losses when overflow is common. For `genome`, none of the cache-based designs scale past 2×. For benchmarks with less common overflow and higher contention, such as `intruder` and `kmeans`, all three designs outperform LogTM, which has much higher latency for aborted transactions. The different approaches to contention are visible here as well: TagTM is the best performer for `intruder` and `kmeans` for 3 of the 4 simulated machines.

6.4.3 Overflow and Contention

Because overflow has a first order impact in the benchmarks in this study, the HTM models were evaluated under ideal overflow conditions. While the approach does allow transactional lines to be evicted from L1 caches, (incurring normal latency for the specified cache geometry), transactions were not forced to restart on overflow. Note that in this section, the "rock+" model allows transactions that survive exceptions, but does not model an unbounded transactional store buffer (larger store buffers for Rock are considered in section 6.4.4). Figure 6.8 shows raw execution time for a subset of the stamp benchmarks, including variations on the models that idealize overflow by providing unbounded space for speculative writes and unbounded tracking of read/write sets.

The data show that for benchmarks with large transactions, the ability to handle overflow gracefully is the determinant for performance: where overflow is not a significant cost, the ability to handle contention gracefully becomes the fundamental differentiator. For example, `genome` has transaction restart rates of 30% for the cache-based designs and 0.6% for the xLogTM model, indicating that about 1/3 of the transactions in `genome` cause overflow, but contention is quite rare. Figure 6.8's graph for `genome` shows that this results in pathologically poor performance (near serial) for the cache-based designs. However, the idealized versions of those designs (lazy-idealov, metatm-idealov, and tagtm-idealov) show no significant performance differences from each-other and show a performance win over the xLogTM design only because the xLogTM model does not idealize the cost of log-writes and log-walks on aborts.

In contrast, `vacation` has restart rates just under 40% for xLogTM and ranging from 55-75% for the cache-based HTMs, indicating that `vacation` has both significant rates of overflow *and* significant contention. For `vacation`, the Rock model shows the poorest performance because 100% of transactions restart for TLB misses. The rock+ model however, has faster run-times than the cache-based model, outperforming MetaTM by 18%. This delta represents the higher cost of attempting transactions that are doomed to overflow with larger buffers. The cache-based designs can execute much further into transactions that eventually overflow, while Rock and Rock+ will overflow and fail much sooner. The longer period of speculation before overflow incurs (on average) higher latency for overflowing transactions, and translates to poorer performance. The idealized cache-based designs outperform xLogTM primarily because of xLogTM's overhead for unbounded transactions. These designs also show significant differences from each-other that derive from strategies for handling contention. The idealized TagTM model uses *notifying transactions*, which affords 13%, and 9% performance gains over the idealized LazyTM and MetaTM models, respectively.

The `intruder` benchmark has restart rates in the range of 30-45% for xLogTM and 40-55% for the cache-based designs: `intruder` has high contention, but transactions seldom overflow (less that 1% of restarts are due to overflow). The Rock model performs worst here, with the vast majority of restarts (92%) being caused by TLB misses: removing this restriction allows the Rock+ model to perform best among the designs that do not idealize overflow, outperforming TagTM, MetaTM, and xLogTM by 14%, 7%, and 30% respectively. Overflow is rare enough in `intruder` that contention management is a significant performance differentiator both the idealized and non-idealized cache-based designs. TagTM outperforms LazyTM and MetaTM by 6-7%, while the idealized TagTM, outperforms MetaTM by 4% and LazyTM by 8%.

The `kmeans` graph in figure 6.8 illustrates a similar situation to that in `intruder`. Because `kmeans` executes a variable number of transactions, and has significant variance in run-time as a result, data shown are throughput in transactions per second. `kmeans` has both minimal contention (5-8%) and very small

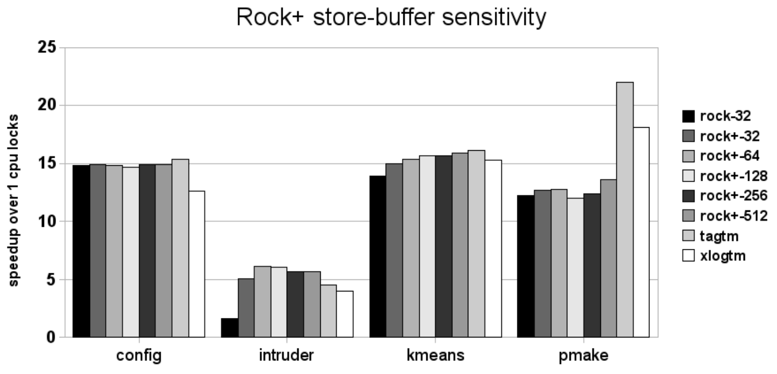


Figure 6.9: Speedup over 1 cpu locks for the Rock+ HTM model, with different store buffer sizes. The original Rock model (rock-32), TagTM, and LogTM speedup are depicted as well for reference.

transactions– no overflows occur in either Rock or the cache based designs. In this case performance is determined primarily by overhead for common case uncontended transactions. xLogTM has the highest such overhead, while Rock has the least: as a result both Rock and Rock+ outperform xLogTM by 2%. The TagTM model performs best among the cache-based designs, but because contention is minimal, contention management effects are likewise minimal.

6.4.4 Rock+

Previous sections demonstrated that the TM support in Rock has limitations that are crippling for the workloads examined in this study. In particular, the use of a 32-slot store queue to buffer speculative writes puts a severe limit on the size of transaction read-write sets; for many workloads, such as `genome` a significant fraction of transactions overflow a 32-slot store queue, with the result that the transaction must roll back and acquire a lock. The design decision to abort on exceptions such as TLB-misses has an even more profound effect: many benchmarks take a large number of TLB-misses in transactions (such as `vacation`, which takes a TLB-miss in every transaction). In the STAMP benchmarks, where rolling back on overflow or exceptions results in the acquisition of the single global lock, the result can be

		exec	**rstpct**	**rst/tx**	**bkcyc/tx**	**txcyc**	**ovrst/tx**
config	rock-32	0.30	11.1	3	482	444269	5190 (0.8)
	rock+-32	0.29	1.2	1	511	428471	5476 (5.4)
	rock+-64	0.29	1.3	1	512	446576	4989 (4.8)
	rock+-128	0.30	1.2	1	511	452589	4936 (5.1)
	rock+-256	0.29	1.2	1	512	438544	238 (0.2)
	rock+-512	0.29	1.2	1	512	440053	227 (0.2)
intruder	rock-32	0.6770	92.1	125.5	37	915	4 (0.0)
	rock+-32	0.2122	54.4	872.8	827	6581	123625 (0.9)
	rock+-64	0.1755	49.1	555.4	434	5637	72387 (0.8)
	rock+-128	0.1777	49.5	579.7	466	5698	74185 (0.8)
	rock+-256	0.1902	51.5	675.5	575	6041	92466 (0.9)
	rock+-512	0.1908	51.8	680.3	574	6054	93134 (0.9)
kmeans	rock-32	1.9302	8.2	32.7	8	671	1 (0.0)
	rock+-32	1.5869	5.9	30.6	8	671	0 (0.0)
	rock+-64	1.6371	5.9	30.4	8	670	1 (0.0)
	rock+-128	1.7169	6.0	30.9	8	671	0 (0.0)
	rock+-256	1.7162	6.0	31.1	8	670	1 (0.0)
	rock+-512	1.6910	6.0	30.9	8	671	0 (0.0)
pmake	rock-32	0.02	20.9	11	583	34405	300 (0.5)
	rock+-32	0.02	3.5	16	655	34462	376 (0.8)
	rock+-64	0.02	3.1	8	608	33957	48 (0.2)
	rock+-128	0.02	3.3	15	651	33627	27 (0.1)
	rock+-256	0.02	2.6	3	588	33660	19 (0.1)
	rock+-512	0.01	3.0	5	607	33629	25 (0.1)

Table 6.6: Statistics for the xRock+ HTM model running on a 3GHz 16 core CMP. The **tx** column is the number of transactions executed by the benchmark, **exec** is execution time (system time for TxLinux, user time for STAMP), **rstpct** is the percentage of transactions that restart. The **rst/tx** column shows the average number of restarts per transaction; **bkcyc/tx** shows the average number of backoff cycles per transaction, while **txcyc** is the average number of cycles in the transaction and **ovrst/tx** is the average number of overflow restarts per transaction (shown with the percentage of overflowed transactions in parentheses).

near complete serialization of the benchmark.

To evaluate the potential profitability of removing some of these restrictions, some "Rock+" HTM models were evaluated. The Rock+ models do not abort on difficult instructions or exceptions, and use larger store queues ranging from 64 to 512 slots to increase the upper bound on transactional write set sizes. Table 6.6 shows statistics for variations on Rock+, while Figure 6.9 shows speedup of the different models over sequential execution (1 cpu/locks). Figure 6.9 also provides speedup data for the original Rock model as well as TagTM and LogTM for reference. The

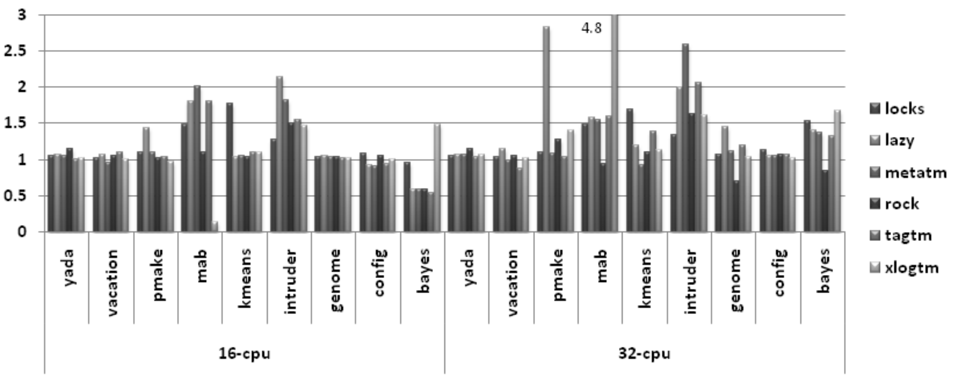


Figure 6.10: Speedup of CMP machines over SMP machines, per benchmark, per HTM design.

data show that the greatest potential gain for Rock is avoiding aborts on exceptions. The Rock+-32 model shows performance that is comparable to TagTM and LogTM for all but the `pmake` benchmark, and even outperforms them for `intruder`. Increasing store buffer size beyond 32 slots is does not always yield a substantial increase in scalability. The benchmarks not shown yield similar results, with the caveat that none of the Rock+ models were able to make a significant improvement on `genome`, whose transaction sizes consistently cause overflow in L1 caches, making a larger store buffer a moot point.

6.4.5 Design sensitivity to organization

Figure 6.10 shows the speedup of CMP machines over SMP machines, per benchmark and HTM design. The choice of CMP versus SMP organization reveals that for many benchmarks, HTM designs are not particularly sensitive to cache organization. For workloads that do not have inherent scalability limitations (due to sharing), 32cpu machines show (an expected, if understated) performance improvement over 16cpu machines, and performance on CMP machines tends to be slightly better than that for SMP machines. Improved performance for CMP machines is largely due to the fact that working sets fit in the 4MB L2 cache. However, the presence of contention does cause all designs to perform significantly worse on SMP than on CMP machines. High contention results from significant write sharing of data, which on an SMP machine will cause additional coherence traffic and L2 misses that will not occur in a CMP where the L2 cache is shared. As a result higher contention benchmarks such as `intruder` show a performance improvement in a CMP organization: the effect is more pronounce in the cache based designs, where failed transactions must also invalidate speculatively written lines, introducing even more latency for subsequent cache misses.

6.4.6 Memory Bandwidth

Tables 6.7 and 6.8 show measured ideal bandwidth consumption for all benchmarks on 16 and 32 cpu machines respectively. The harmonic means across all benchmarks are provided as well. Figure 6.11 depicts time-series data for the peak bandwidth per 20,000 cycle epoch: the plots rely on a moving average over 5 points (100,000 cycles). The data show that for while bandwidth needs can vary significantly from workload to workload, and across designs, average bandwidth needs are modest, showing that these benchmarks make good use of caches to reduce pressure on main memory. where contention is minimal, all approaches have similar average bandwidth consumption. For example, `genome` has low restart rates due to contention: on 16 cpus the average bandwidth consumption ranges from 0.2 to 0.3 GB/sec for CMP machines and from 0.4 to 0.7 GB/sec for SMP machines. Similar trends are observable for `pmake`. The presence of contention introduces wider variability in both peak and average bandwidth across designs due to the different approaches the designs use to handle contended transactions.

The lowest overall bandwidth demands come from first from locks and second from the xLogTM design. In the case of locks, the use of mutual exclusion to protect shared resources reduces traffic for shared cache lines with respect to the HTM designs: TM allows multiple threads to enter critical sections, with the result

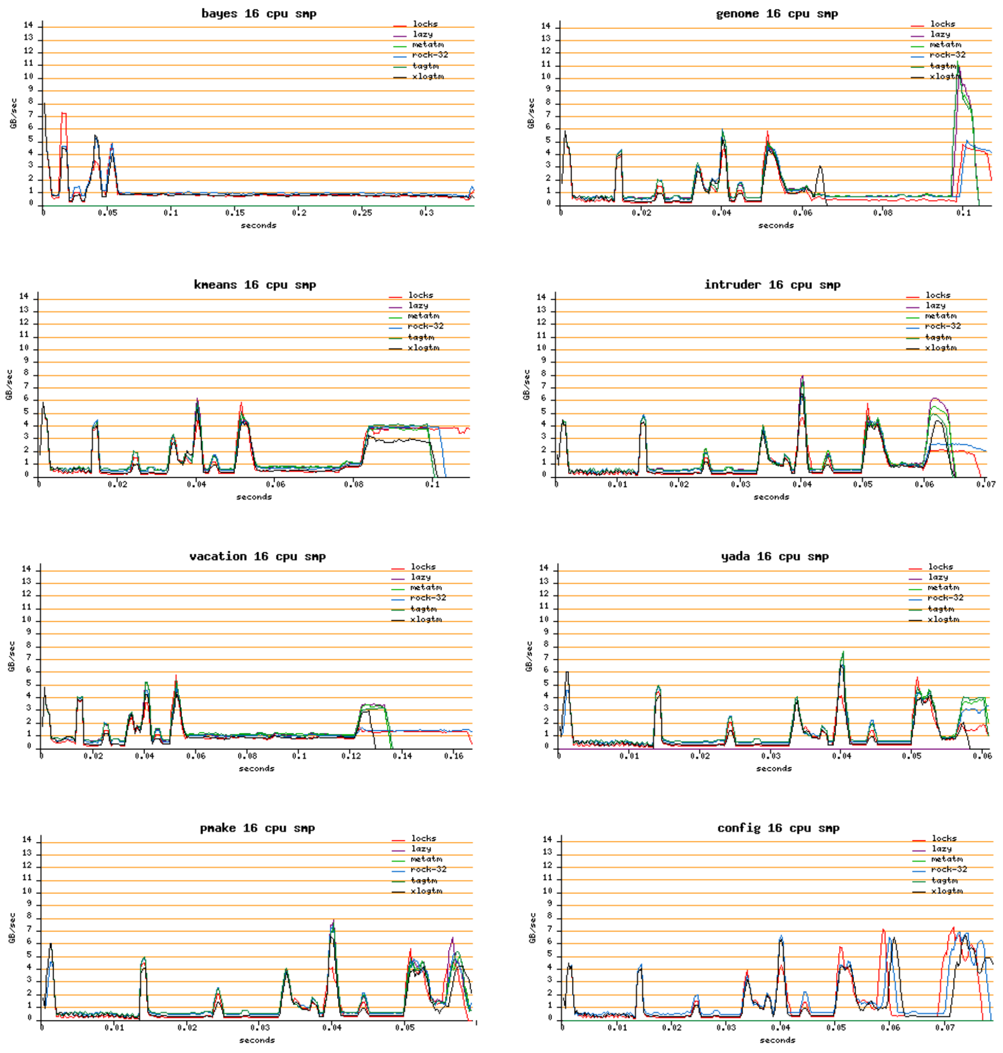


Figure 6.11: Peak Bandwidth over 20,000 cycle epochs for 16 cpu SMP machines: each sampled point is a moving average of the peak over the previous 5 epochs.

that coherence traffic for lines accessed in those critical regions can be generated by more than the single cpu holding the lock. Moreover, both Linux and the locking variant of STAMP used in this study use test-and-test-and-set spinlocks, which minimizes coherence traffic for contended lock variables. HTM, with the exception of TagTM, in general does not have similar mechanisms for minimizing traffic for

			bayes	**genome**	**kmeans**	**intruder**	**vacation**	**yada**	**config**	**mab**	**pmake**	**harmean**
cmp	locks	avg	0.0	0.2	0.1	0.1	0.3	0.0	0.5	0.2	-	0.1
		max	27.9	27.9	27.9	27.9	27.9	27.9	32.9	33.7	-	32.7
	lazy	avg	-	0.3	0.4	0.4	0.6	0.1	-	-	0.2	0.3
		max	-	40.8	27.9	27.9	27.9	27.9	-	-	46.2	31.7
	metatm	avg	-	0.3	0.4	0.3	0.6	0.1	-	-	0.2	0.3
		max	-	39.9	27.9	27.9	27.9	27.9	-	-	35.0	30.5
	rock-32	avg	0.1	0.2	0.4	0.2	0.4	0.1	0.8	0.5	0.2	0.2
		max	27.9	27.9	27.9	27.9	27.9	27.9	32.1	33.0	36.9	29.6
	tagtm	avg	-	0.3	0.4	0.3	0.6	0.1	-	0.5	0.2	0.3
		max	-	40.0	27.9	27.9	27.9	27.9	-	35.4	32.4	30.8
	xlogtm	avg	0.0	0.3	0.2	0.1	0.4	0.1	0.6	0.1	0.1	0.1
		max	28.4	28.4	28.4	28.4	28.4	28.4	43.0	28.4	28.4	29.5
smp	locks	avg	0.1	0.5	1.3	0.5	0.7	0.1	0.7	0.5	0.2	0.2
		max	43.0	43.0	43.0	43.0	42.0	43.0	43.0	43.0	43.0	42.8
	lazy	avg	-	0.5	0.6	0.8	0.7	-	-	-	0.3	0.5
		max	-	43.8	43.7	43.8	43.8	-	-	-	43.7	43.8
	metatm	avg	-	0.5	0.6	0.8	0.7	0.2	-	0.7	0.3	0.4
		max	-	43.7	43.7	43.7	43.7	43.7	-	43.7	43.7	43.7
	rock-32	avg	0.1	0.7	0.6	0.8	-	0.2	0.9	-	0.2	0.3
		max	43.7	43.7	43.7	43.7	-	43.7	43.7	-	43.7	43.7
	tagtm	avg	-	0.5	0.6	0.6	0.7	0.2	-	-	0.2	0.4
		max	-	43.7	43.7	43.7	43.7	43.7	-	-	43.7	43.7
	xlogtm	avg	0.1	0.4	0.5	0.5	0.5	0.2	-	0.5	0.2	0.2
		max	43.0	43.0	43.0	43.0	43.0	43.0	-	43.0	43.0	43.0

Table 6.7: Bandwidth consumption metrics (average and peak) in GB/sec for locks, MetaTM, xLogTM, and xRock, with 16 CPUs. A dash indicates an unavailable data point. The **harmean** column is the harmonic mean across all benchmarks.

contended lines. Failed transactions invalidate speculatively written lines, and retry until successful, so repeated failed retries can generate the same coherence traffic repeatedly. The addition of the retry-on-overflow semantics of the "best-effort" HTMs introduces additional retry scenarios, and subsequent lock acquisition that must be coordinated with active transactions through the use of `cxspinlocks`.

While LogTM introduces extra writes for logging old values in transactions, restarts are less expensive in terms of subsequent cache misses because speculative lines need not be invalidated. Because the log resides in cachable virtual memory, and is private, cache lines used for logging will only be evicted from caches due to

			bayes	**genome**	**kmeans**	**intruder**	**vacation**	**yada**	**config**	**mab**	**pmake**	**harmean**
cmp	locks	avg	0.0	0.1	0.1	0.1	-	0.0	0.5	0.2	0.1	0.1
		max	28.0	28.0	28.0	28.0	-	31.8	66.4	33.2	36.4	32.4
	lazy	avg	-	0.3	0.4	0.5	0.7	0.1	-	-	0.2	0.3
		max	-	79.7	28.9	39.8	28.2	37.4	-	-	35.2	36.8
	metatm	avg	-	0.3	0.4	0.4	-	0.1	-	-	0.2	0.2
		max	-	80.5	28.2	34.3	-	28.6	-	-	37.3	36.0
	rock-32	avg	0.1	0.2	0.3	0.2	0.3	0.1	-	0.4	0.2	0.2
		max	28.1	28.8	28.1	28.1	28.1	29.1	-	29.9	33.5	32.7
	tagtm	avg	-	0.3	0.3	0.3	-	0.1	-	-	0.2	0.2
		max	-	80.7	28.2	34.7	-	28.2	-	-	39.2	36.3
	xlogtm	avg	0.0	0.2	0.1	0.1	0.4	0.1	0.6	0.3	0.1	0.1
		max	29.8	30.3	30.3	29.8	29.8	30.3	46.1	29.8	29.8	31.2
smp	locks	avg	0.1	0.8	-	0.7	-	0.1	0.7	0.5	0.1	0.2
		max	43.7	43.7	-	43.7	-	43.7	43.7	43.7	43.7	43.7
	lazy	avg	-	-	0.7	1.1	1.1	0.3	-	-	0.3	0.5
		max	-	-	44.6	44.6	44.6	44.6	-	-	51.0	45.8
	metatm	avg	-	0.6	0.6	1.1	0.9	0.2	-	-	0.2	0.4
		max	-	79.7	44.6	44.6	44.5	44.6	-	-	44.6	48.1
	rock-32	avg	0.1	-	0.6	1.0	1.0	0.2	0.8	-	0.2	0.3
		max	44.4	-	44.3	44.3	44.4	42.1	61.6	-	44.3	45.8
	tagtm	avg	-	0.6	0.6	0.8	-	0.2	-	-	0.2	0.4
		max	-	79.4	44.6	44.6	-	44.6	-	-	44.6	48.9
	xlogtm	avg	0.1	0.3	0.5	0.7	0.5	0.1	-	-	0.2	0.2
		max	43.7	43.8	43.7	43.8	43.8	43.8	-	-	43.8	50.0

Table 6.8: Bandwidth consumption metrics (average and peak) in GB/sec for locks, MetaTM, xLogTM, and xRock, with 32 CPUs. A dash indicates an unavailable data point. The **harmean** column is the harmonic mean across all benchmarks.

capacity, and never due to coherence traffic generated by sharing. While LogTM does use more memory than the other designs to implement the TM abstraction, this extra memory usage does not translate to increased memory bandwidth pressure.

The ability of TagTM to reduce bandwidth pressure under contention has minimal impact on bandwidth consumption averaged across the whole benchmark, but it is visible in the time series data in cases where sharing introduces spikes in memory traffic. For instance, for `intruder` 16 cpu SMP, around 0.06 seconds into the benchmark, a spike occurs for which TagTM, reduces memory traffic by 1 GB/sec with respect the Lazy HTM and by 0.5 GB/sec with respect to MetaTM.

However, it is interesting to note that xLogTM's bandwidth needs in this phase are even lower, due to not requiring invalidation of cache lines for failed transactions. The fact that Rock must serialize transactions failed due to overflow in this phase is evinced by the fact that the phase takes twice as long to execute and has a bandwidth consumption profile similar to that of locks.

6.5 Conclusion

Ultimately, the data in this study reflect that even leaving cost and complexity considerations aside, the attractiveness of a particular design is tightly coupled with the envisioned workload. While xLogTM is clearly the most broadly applicable design, yielding predictable and scalable performance for workloads with large transactions, it also represents a point in the design space characterized by considerable investment of hardware mechanism: moreover, for programs that minimize sharing (a desirable property for any scalable program), the xLogTM design incurs the highest common case performance penalty, and would be an unlikely design point for fine-grain synchronization such as might occur in a modern OS. Conversely, while Rock represents a point in design space characterized by economy of hardware mechanism, the economy severely impacts the usability. It is interesting to note however, that this impact is much more a result of exception handling in transactions rather than from store-buffer geometry: the fact that overflowing transactions overflow *early* in Rock allows it to perform better than the heavier-weight cache-based designs for some workloads (e.g. `vacation`). The cache-based designs represent an interesting middle-ground: if a design goal of TM is to provide fine-grain performance with coarse-grain complexity, the ability to support somewhat larger transactions is a mandate.

It is also important to note that while the cache-based designs in this study differ from each other in terms of support for mechanism that address contention, similar mechanisms could be supported (to varying degree) in designs that are not-cache based. LogTM in particular could benefit from both *notifying transactions* and *transaction annotation*. Because Rock uses a store-buffer for writes, but still uses L1 cache lines to represent membership in read-sets, these mechanisms could be used, with some adaptation, in that context as well.

Chapter 7

Related work

Larus and Rajwar provide a thorough reference on TM research through the beginning of summer 2006 [42]. Optimistic synchronization [33], motivated early HTM designs [36]. Rajwar and Goodman explored transactional [66,67] execution of critical sections. Recent HTMs has focused on the architectural mechanisms that provide transactional memory [?, 5, 9, 12, 13, 18, 22, 48, 58, 70, 85, 94], language-level support for HTM [2, 15], and transactional resource virtualization [10, 19, 39, 68, 96] as well as hybridization [7, 45, 46]. This thesis focuses on the tradeoffs introduced by mechanisms proposed for supporting HTM, and while MetaTM [70, 75] and DATM [71] represent unique points in that space, the MetaTM infrastructure supports modes that emulate important elements from most of these designs.

I/O in transactions. Proposals for I/O in transactions fall into three basic camps: isolation escape hatches, delaying I/O until commit [29,30], or guaranteeing commit for transactions that have performed I/O [?, 5, 10]. All of these strategies have serious drawbacks.

Many HTM systems allow a transactional escape hatch known as an open nested transaction [59, 61, 63]. An open nested transaction can read the partial results of the current transaction and any changes it makes, including I/O operations, are not isolated. If the enclosing transaction restarts, the effect of the open-nested transaction must be undone by code provided by the programmer: the resulting programmer effort (writing and maintain compensating code) severely compromises the utility of open-nested transactions. Efficient hardware implementations of open nesting introduce correctness conditions that are subtle and easy to violate in common programming idioms [38], as well as restricting the transactional programming model. Delaying I/O is not possible when the code performing the I/O depends on its result, e.g., a device register read might return a status word that the OS must interpret in order to finish the transaction. Guaranteeing that a transaction will commit severely limits scheduler flexibility, and can, for long-running or highly contended transactions, result in serial bottlenecks or deadlock. Non-transactional threads on other processors which conflict the guaranteed thread will be forced to retry or stall until the guaranteed thread commits its work. This will likely lead to lost timer interrupts and deadlock in the kernel.

Scheduling. Carlstrom et al. [15] demonstrate a scheduler wherein the scheduler thread in a Java VM listens for conflicts on behalf of a yielded thread. The technique requires a dedicated core for the scheduler thread, which is very wasteful in an OS, and does not scale as there is no bound on the size of transaction sets amassed by the scheduler.

Zilles [96] explores modifications to the OS that allow micro-architectural events to modify task state and raise exceptions to invoke the scheduler, providing a mechanism for a thread involved in a transactional conflict to deschedule itself. While the TxLinux scheduler attempts to deschedule threads involved in multiple restarts, the mechanism is entirely under the control of of the OS, while the Zilles techniques puts the scheduler directly at the mercy of the hardware.

Mainstream operating systems such as Microsoft Windows [79], Linux [14] and Solaris [49] implement sophisticated priority-based pre-emptive schedulers, with different classes of priorities, and a variety of scheduling techniques for each class.

Bilge et. al. [3] explore hardware support for priority inheritance using spinlocks. The approach uses hardware to support priority inheritance which only provides an upper bound on priority inversion, while this work takes advantage of transactional hardware to avoid priority inversion before it occurs. The Linux RT patch [78] supports priority inheritance to help mitigate the effects of priority inversion: the Linux RT patch implementation converts spinlocks to mutexes, changing a busy-waiting primitive to a blocking primitive, and relying on the scheduler to react to inherited priority. By contrast, the *os_prio* policy allows the contention manager to nearly eliminate priority inversion without requiring the primitive to block or involve the scheduler.

Write-shared data. One approach to the problem of write-shared data is to make it the responsibility of the programmer not to write-share data in the first place. Such systems usually provide at least some performance analysis tools [17] to help programmers identify data hotspots, leaving them with these alternatives that ultimately lead to some combination of greater programming effort, decreased maintainability, reduced functionality, and more bugs. These costs are a significant price to pay for higher concurrency. Large-scale SMPs have provided mechanisms to help programmers deal with contented shared data. DASH [1] supported queue locks, update-writes, deliver instructions, and Fetch&Op in the directory protocol. Fetch&Op was found to be the most useful [44] and has received increased attention since [43]. (e.g. by having a dedicated cache [43]).

TM programming model extensions. Several proposed extensions to the TM programming model can be used to achieve higher performance, including privatization [89], early release [88], escape actions [96], open and closed nesting [60, 64], Galois classes [40], transactional boosting [34] and abstract nested transactions [32]. Because these techniques all change the programming model, they increase programmer effort, making increased complexity the price for better performance. They differ in their degree of applicability and the difficulty of reasoning involved, as well as the amount of additional compromises they force on their users. For example, using escape actions to implement a counter requires the programmer to also write a compensation block, which is a significant programmer burden. Moreover, semantics may be weakened when using this approach (e.g. a counter implemented this way is no longer monotonically increasing). In Galois and transactional boosting, the programmer needs to provide inverse operations for the concurrent data structures, which might be difficult (e.g., k-d tree), as well as define commutativity relationships between the various operations.

Thread-level speculation Designs for thread-level speculation [26, 74] are similar to DATM in their support for multiple versions of speculative data. In the TLS taxonomy, DATM merges its results with main memory lazily (via the CTM cache state). However, state management in DATM is much simpler than TLS. For example, of the five challenges to buffering state in TLS (including multiple speculative tasks per processor and multiple versions of a variable in a processor), DATM needs to deal with three of them (buffering and merging speculative state and multiple versions of the same variable at different processors). Current TM systems must deal with two of the challenges, buffering and merging speculative state.

TLS must squash speculation on dependence violation, and current designs tolerate some memory access conflicts. TLS tolerates a subset of the conflicts tolerated by DATM [74]. TLS systems must support a large number of speculative tasks, e.g., using a 6-bit local identifier [74]. DATM allows 3 suspended transactions using a 2-bit identifier, but suspended transactions cannot be the source or destination of a dependence in DATM. Suspended transactions in DATM are logically independent, unlike TLS tasks.

MVCC. Dependence-aware TM shares some conceptual ground with multiversion concurrency control (MVCC). The DATM implementation keeps track of multiple versions of an object when it is being modified concurrently by many transactions. MVCC (also called time-domain addressing [73]) also tracks multiple versions of each object. The key difference is that DATM is designed to specifically deal with hotspots, whereas hotspots are known to degrade performance of MVCC database systems. The techniques used by DATM (transactions dependences and forwarding data between transactions) are not found in MVCC systems. Moreover, the lack of durability of memory transactions makes it easier to implement efficiently. Unlike MVCC DATM is able to efficiently handle hotspots by forwarding data.

DATM and TM Serializability. Aydonat and Abdelrahman [6] have (simultaneously with us) identified that current transactional memory implementations apply a stronger form of serializability than conflict serializability, thus reducing the amount of useful concurrency in the system. They have a software system which does not accept all conflict serializable schedules as DATM does. In particular, their implementation would not allow concurrent updates to a shared counter.

TagTM. Notifying transactions are similar to stall-on-conflict [58] in that the mechanism allows a conflicting transaction to wait in hopes that a conflict will resolve. However there are some fundamental differences. Stall-on-conflict leaves conflicting transactions in progress, increasing the probability that additional conflicts can arise for other cache lines held in the read-write set of the stalled transaction. Stall-on-conflict introduces the need for deadlock detection/avoidance, and brings up thorny policy issues around asymmetric conflict. Most importantly, stall-on-conflict is not software-visible, and it causes transactions to stall in the coherence layer: decisions to wait or retry can be better made by software. Notifying transactions give software flexibility to decide how to handle contention, which hardware-based backoff and stall-on-conflict policies cannot provide. Notifying transactions share some common properties with the Alert-On-Update mechanism described by Shriraman et al. [86] in that it leverages coherence to notify conflicting or waiting threads of important transactional events, and both mechanisms rely on specialized coherence protocols. Alert-On-Update is a building block for STM implementation, and effectively works by detecting conflicts on cache-lines read in a speculative context. Notifying transactions, by contrast, provides a way for a committing transaction to notify a non-transactional thread that retrying a previously conflicting transaction may be profitable; the intended use of notifying transactions is a hint to software that is synchronizing with HTM.

The design comparison in Chapter 6 is related to area of TM virtualization in that several designs may often revert to locks to support atomicity and isolation for transactions that overflow hardware resources. Other approaches to virtualization include falling back to TM at page granularity [18, 19], and handling overflow directly with dedicated hardware [68, 91]. Other approaches restrict the concurrency of overflowed transactions in order to virtualize overflow [10, 29], which are fundamentally mechanisms to make overflowed transactions "inevitable." [90]. Retrying with a lock instead of a transaction requires some mechanisms for cooperation between locks and transactions [75] to ensure fairness between transactional and non-transactional critical sections. This approach is similar to speculative [66] or transactional [67] execution of critical sections.

Chapter 8

Conclusion

TxLinux is the first operating system that uses HTM as a synchronization primitive, and presents innovative techniques for HTM-aware scheduling and cooperation between locks and transactions. TxLinux demonstrates that HTM provides comparable performance to locks, and can simplify code while coexisting with other synchronization primitives in a modern OS. The cxspinlock primitive enables a solution to the long-standing problem of I/O in transactions, and the API eases conversion from locking primitives to transactions significantly. Introduction of transactions as a synchronization primitive in the OS reduces time wasted synchronizing on average, but can cause pathologies that do not occur with traditional locks under very high contention or when critical sections are sufficiently large for the overhead of HTM virtualization to become significant. HTM aware scheduling eliminates priority inversion for all the workloads we investigate, and enables better management of very high contention in ways that are not possible with traditional locks. However, it is unable to have a significant impact on the performance of workloads with normal contention profiles.

Dependence-aware transactions increase throughput by enabling concurrent execution of transactions that would otherwise conflict due to updating shared data structures. This thesis presents the design, and a prototype implementation of a dependence-aware hardware transactional memory system. Experimental results confirm the potential performance benefits of dependence-aware transactional memory as compared to traditional HTM implementations. DATM enables performance improvements that can be achieved through mechanisms that are completely transparent to the programmer.

This thesis has also presented TagTM, and explored mechanisms that avoid and manage contention among transactions in an HTM. We find that *notifying transactions* can improve performance under contention, and can significantly reduce pressure on memory bandwidth for transactions that must restart. We examine implementation details for supporting flexible contention management policies in an HTM, and suggest *transaction annotation* as a mechanism that allows contention management decisions to be rendered locally at nodes where conflicts are detected; transaction annotation eliminates the the design complexities incurred by software conflict handlers suggested in previous designs. This paper has contributed a new transactional coherence protocol called **XMESI**.

Finally, this thesis has contributed a detailed cross-product design comparison for hardware transactional memory. The study reveals that designs have performance characteristics that are tightly coupled with the envisioned workload. If TM's charter is fundamentally about ease-of-programming, designs that have natural support for unbounded transactions such as LogTM [58] or TokenTM [12] are attractive; however, such designs trade common case performance and contended performance for unbounded support, making these designs a less attractive fit for contexts where critical sections are short such as an operating system, or where contention may be common. The minimal TM support provided by Rock [22] represents the opposite end of this spectrum: a slightly enhanced Rock-like design can provide very good common case performance for short and small critical sections, but does not go far to deliver on TM's promise of easy parallel programming, as it's failure modes are common and diverse enough to relegate the mechanism to use by expert programmers. Cache-based designs fall squarely in the middle, representing a point in the design space that supports larger transactions than Rock, and is easily enhanced to support mechanisms that help programmers manage contention, but can be slower in the common case than Rock-like designs for uncontended transactions, and still require some awareness of cache geometry from programmers (albeit at a noticably higher threshold on transaction size) if good performance is to be achieved. The cache-based designs represent an interesting middle-ground: if a design goal of TM is to provide fine-grain performance with coarse-grain complexity, the ability to support somewhat larger transactions is a mandate.

Bibliography

[1] The stanford dash multiprocessor. *IEEE Computer*, 25(3), 1992.

[2] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37, New York, NY, USA, 2006. ACM.

[3] Bilge E. S. Akgul, Vincent J. Mooney III, Henrik Thane, and Pramote Kuacharoen. Hardware support for priority inheritance. *rtss*, 00:246, 2003.

[4] Alaa R. Alameldeen and David A. Wood. Variability in architectural simulations of multi-threaded workloads. In *HPCA '03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, page 7, Washington, DC, USA, 2003. IEEE Computer Society.

[5] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. volume 26, pages 59–69, Los Alamitos, CA, USA, 2006. IEEE Computer Society Press.

[6] U. Aydonat and T. Abdelrahman. Serializability of transactions in software transactional memory. In *TRANSACT*, 2008.

[7] Lee Baugh, Naveen Neelakantam, and Craig Zilles. Using hardware memory protection to build a high-performance, strongly atomic hybrid transactional memory. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*. June 2008.

[8] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, 1970.

[9] Colin Blundell, Joe Devietti, E. Christopher Lewis, and Milo M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. *SIGARCH Comput. Archit. News*, 35(2):24–34, 2007.

[10] Colin Blundell, Joe Devietti, E. Christopher Lewis, and Milo M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 24–34, New York, NY, USA, 2007. ACM.

[11] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Deconstructing transactions: The subtleties of atomicity. In *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*. Jun 2005.

[12] Jayaram Bobba, Neelam Goyal, Mark D. Hill, Michael M. Swift, and David A. Wood. Tokentm: Efficient execution of large transactions with hardware transactional memory. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*. Jun 2008.

[13] Jayaram Bobba, Kevin E. Moore, Luke Yen, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. Jun 2007.

[14] D. Bovet and M. Cesati. *Understanding the Linux Kernel.* OŔeilly Media, Inc., 3rd edition, 2005.

[15] Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. The atomos transactional programming language. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 1–13, New York, NY, USA, 2006. ACM.

[16] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 227–238, Washington, DC, USA, 2006. IEEE Computer Society.

[17] Hassan Chafi, Chi Cao Minh, Austen McDonald, Brian D. Carlstrom, JaeWoong Chung, Lance Hammond, Christos Kozyrakis, and Kunle Olukotun. Tape: a transactional application profiling environment. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 199–208, New York, NY, USA, 2005. ACM.

[18] Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Brad Calder, and Osvaldo Colavin. Unbounded page-based transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 347–358, New York, NY, USA, 2006. ACM.

[19] JaeWoong Chung, Chi Cao Minh, Austen McDonald, Travis Skare, Hassan Chafi, Brian D. Carlstrom, Christos Kozyrakis, and Kunle Olukotun. Tradeoffs in transactional memory virtualization. In *ASPLOS-XII: Proceedings of the*

12th international conference on Architectural support for programming languages and operating systems, pages 371–381, New York, NY, USA, 2006. ACM.

[20] David Culler, J. P. Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, August 1998.

[21] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 336–346, New York, NY, USA, 2006. ACM.

[22] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. pages 157–168, 2009.

[23] David Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC*, 2006.

[24] Edsger W. Dijkstra. The structure of the "the"-multiprogramming system. *Commun. ACM*, 11(5):341–346, 1968.

[25] E. Elnozahy, D. Johnson, and Y. Wang. A survey of rollback-recovery protocols in message-passing systems, 1996.

[26] María Jesús Garzarán, Milos Prvulovic, José María Llabería, Víctor Vi nals, Lawrence Rauchwerger, and Josep Torrellas. Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors. *ACM Trans. Archit. Code Optim.*, 2(3):247–279, 2005.

[27] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[28] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. Stmbench7: a benchmark for software transactional memory. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 315–324, New York, NY, USA, 2007. ACM.

[29] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 102, Washington, DC, USA, 2004. IEEE Computer Society.

[30] Tim Harris. Exceptions and side-effects in atomic blocks. *Sci. Comput. Program.*, 58(3):325–343, 2005.

[31] Tim Harris, Maurice Herlihy, Simon Marlow, and Simon Peyton-Jones. Composable memory transactions. In *PPoPP*, Jun 2005.

[32] Tim Harris and Srdan Stipic. Abstract nested transactions. In *Second ACM SIGPLAN Workshop on Transactional Computing*, 2007.

[33] Maurice Herlihy. Wait-free synchronization. volume 13, pages 124–149, New York, NY, USA, 1991. ACM.

[34] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 207–216, New York, NY, USA, 2008. ACM.

[35] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *SIGPLAN Not.*, volume 41, pages 253–262, New York, NY, USA, 2006. ACM.

[36] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[37] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, Jul 1990.

[38] Owen S. Hofmann, Donald E. Porter, Christopher J. Rossbach, Hany E. Ramadan, and Emmett Witchel. Solving difficult HTM problems without difficult hardware. In *ACM TRANSACT Workshop*, 2007.

[39] Owen S. Hofmann, Christopher J. Rossbach, and Emmett Witchel. Maximum benefit from a minimal htm. pages 145–156, 2009.

[40] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 211–222, New York, NY, USA, 2007. ACM.

[41] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *PPoPP*, 2006.

[42] Jim Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[43] James Laudon and Daniel Lenoski. The sgi origin: a ccnuma highly scalable server. volume 25, pages 241–251, New York, NY, USA, 1997. ACM.

[44] Daniel E. Lenoski and James P. Laudon. Retrospective: the dash prototype: implementation and performance. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 80–82, New York, NY, USA, 1998. ACM.

[45] Yossi Lev and Jan-Willem Maessen. Split hardware transactions: true nesting of transactions using best-effort hardware transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 197–206, New York, NY, USA, 2008. ACM.

[46] Yossi Lev, Mark Moir, and Dan Nussbaum. PhTM: Phased transactional memory. In *Workshop on Transactional Computing (TRANSACT)*, 2007.

[47] P.S. Magnusson, M. Christianson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. In *IEEE Computer vol.35 no.2*, Feb 2002.

[48] Austen McDonald, JaeWoong Chung, Brian D. Carlstrom, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural semantics for practical transactional memory. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 53–65, Washington, DC, USA, 2006. IEEE Computer Society.

[49] R. McDougall and J. Mauro. *Solaris Internals*. Prentice Hall, 2nd edition, 2006.

[50] Paul E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy Update Techniques in Operating System Kernels*. PhD thesis, 2004.

[51] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[52] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[53] John M. Mellor-Crummey and Michael L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. *SIGPLAN Not.*, 26(7):106–113, 1991.

[54] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. Stamp: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[55] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 69–80, New York, NY, USA, 2007. ACM.

[56] Mark Moir, Kevin Moore, and Dan Nussbaum. The adaptive transactional memory test platform: a tool for experimenting with transactional code for rock (poster). In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 362–362, New York, NY, USA, 2008. ACM.

[57] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[58] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. Logtm: log-based transactional memory. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 254–265, 2006.

[59] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in LogTM. In *ASPLOS-XII*. 2006.

[60] J. Eliot B. Moss. *Nested transactions*. MIT, 1985.

[61] J. Eliot B Moss, Nancy D. Griffeth, and Marc H. Graham. Abstraction in recovery management. *SIGMOD Rec.*, 15(2):72–83, 1986.

[62] J. Eliot B. Moss and Antony L. Hosking. Nested transactional memory: Model and architecture sketches. In *In Science of Computer Programming*, volume 63, pages 186–201. Dec 2006.

[63] J. Eliot B. Moss and Tony Hosking. Nested transactional memory: Model and preliminary architecture sketches. In *SCOOL*, 2005.

[64] Yang Ni, Vijay S. Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 68–78, New York, NY, USA, 2007. ACM.

[65] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *USENIX Summer*, pages 247–256, 1990.

[66] R. Rajwar and J. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *MICRO*, 2001.

[67] R. Rajwar and J. Goodman. Transactional lock-free execution of lock-based programs. In *ASPLOS*, October 2002.

[68] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *ISCA '05: Proceedings of the 32nd annual international symposium*

on Computer Architecture, pages 494–505, Washington, DC, USA, 2005. IEEE Computer Society.

[69] Hany Ramadan, Chris Rossbach, and Emmett Witchel. The Linux kernel: A challenging workload for transactional memory. In *Workshop on Transactional Memory Workloads*, June 2006.

[70] Hany E. Ramadan, Christopher J. Rossbach, Donald E. Porter, Owen S. Hofmann, Aditya Bhandari, and Emmett Witchel. Metatm/txlinux: transactional memory for an operating system. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 92–103, New York, NY, USA, 2007. ACM.

[71] Hany E. Ramadan, Christopher J. Rossbach, and Emmett Witchel. Dependence-aware transactional memory for increased concurrency. In *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 246–257, Washington, DC, USA, 2008. IEEE Computer Society.

[72] Hany E. Ramadan, Indrajit Roy, Maurice Herlihy, and Emmett Witchel. Committing conflicting transactions in an stm. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 163–172, New York, NY, USA, 2009. ACM.

[73] David P. Reed. Implementing atomic actions on decentralized data. page 163, 1979.

[74] Jose Renau, Karin Strauss, Luis Ceze, Wei Liu, Smruti Sarangi, James Tuck, and Josep Torrellas. Thread-level speculation on a cmp can be energy efficient. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 219–228, New York, NY, USA, 2005. ACM.

[75] Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Bhandari Aditya, and Emmett Witchel. Txlinux: using and managing hardware transactional memory in an operating system. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 87–102, New York, NY, USA, 2007. ACM.

[76] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is transactional programming actually easier? In *Proceedings of the Eigth Workshop on Duplicating, Deconstructing, and Debunking*, Jun 2009.

[77] Christopher J. Rossbach, Hany E. Ramadan, Owen S. Hofmann, Donald E. Porter, Bhandari Aditya, and Emmett Witchel. Txlinux and metatm: transactional memory and the operating system. *Commun. ACM*, 51(9):83–91, 2008.

[78] Steven Rostedt and Darren V. Hart. Internals of the rt patch. 2007.

[79] M. Russinovich and D. Solomon. *Microsoft Windows Internals: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000.* Microsoft Press, 4th edition, 2004.

[80] Bratin Saha, Ali-Reza Adl-Tabatabai, and Quinn Jacobson. Architectural support for software transactional memory. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 185–196, Washington, DC, USA, 2006. IEEE Computer Society.

[81] Bratin Saha, Ali reza Adl-tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *In Proc. of the 11th ACM Symp. on Principles and Practice of Parallel Programming*, pages 187–197. ACM Press, 2006.

[82] Daniel Sanchez, Luke Yen, Mark D. Hill, and Karthikeyan Sankaralingam. Implementing signatures for transactional memory. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 123–133, Washington, DC, USA, 2007. IEEE Computer Society.

[83] William N. Scherer, III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 240–248, New York, NY, USA, 2005. ACM.

[84] Arrvindh Shriraman and Sandhya Dwarkadas. Refereeing conflicts in hardware transactional memory. In *International Conference on Supercomputing*. Jun 2009.

[85] Arrvindh Shriraman, Sandhya Dwarkadas, and Michael L. Scott. Flexible decoupled transactional memory support. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*. Jun 2008.

[86] Arrvindh Shriraman, Sandhya Dwarkadas, and Michael L. Scott. Flexible decoupled transactional memory support. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 139–150, Washington, DC, USA, 2008. IEEE Computer Society.

[87] Arrvindh Shriraman, Michael F. Spear, Hemayet Hossain, Virendra J. Marathe, Sandhya Dwarkadas, and Michael L. Scott. An integrated hardware-software approach to flexible transactional memory. volume 35, pages 104–115, New York, NY, USA, 2007. ACM.

[88] Travis Skare and Christos Kozyrakis. Early release: Friend or foe? In *Workshop on Transactional Workloads*, 2006.

[89] Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 338–339, New York, NY, USA, 2007. ACM.

[90] Michael F. Spear, Maged M. Michael, and Michael L. Scott. Inevitability mechanisms for software transactional memory. In *Proceedings of the Third ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2008.

[91] Michael M. Swift, Haris Volos, Neelam Goyal, Luke Yen, Mark D. Hill, and David A Wood. Os support for virtualizing transactional memory. In *TRANSACT*, 2008.

[92] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *In 22nd Annual International Symposium on Computer Architecture*, pages 392–403, 1995.

[93] David A. Wood, Garth A. Gibson, and Randy H. Katz. Verifying a multiprocessor cache controller using random test generation. *IEEE Design and Test of Computers*, pages 13–25, August 1990.

[94] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. Logtm-se: Decoupling hardware transactional memory from caches. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 261–272. IEEE Computer Society, Washington, DC, USA, 2007.

[95] Luke Yen, Stark C. Draper, and Mark D. Hill. Notary: Hardware techniques to enhance signatures. In *MICRO '08: Proceedings of the 2008 41st IEEE/ACM*

International Symposium on Microarchitecture, pages 234–245, Washington, DC, USA, 2008. IEEE Computer Society.

[96] Craig Zilles and Lee Baugh. Extending hardware transactional memory. In *TRANSACT*, Jun 2006.

Vita

Chris Rossbach was born in Dayton, Ohio in 1970, son of Dennis and Dianne Rossbach. He graduated from Stanford in 1992 with a B.S. in computer systems engineering, and spent the next decade playing the guitar professionally in the San Francisco Bay area. In 2005 he started the doctoral program in the Department of Computer Sciences at the University of Texas at Austin.

Permanent Address: 308 W. Annie St
Austin, TX 78704

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.