The Dissertation Committee for Hany E. Ramadan
certifies that this is the approved version of the following dissertation:

Transactional Memory Concurrency: New Models and Systems

Committee:

Emmett Witchel, Supervisor

Lorenzo Alvisi

Don Batory

Maurice Herlihy

Keshav Pingali

Transactional Memory Concurrency: New Models and Systems

by

Hany E. Ramadan, M.S. ; 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 my parents

Acknowledgments

Thanks to all the members of Operating System and Architecture group.

HANY E. RAMADAN

The University of Texas at Austin

August 2009

Transactional Memory Concurrency: New Models and Systems

Publication No. ____________________

Hany E. Ramadan, Ph.D.
The University of Texas at Austin, 2009

Supervisor: Emmett Witchel

Transactional memory (TM) aims to bring the benefits of ACID transactions to the volatile world of program synchronization. Architectural trends are making software transactions more appealing, as more programmers struggle with the problems of locks as they exploit multi-core processors. This thesis applies TM, which until recently has been restricted to small benchmarks, to a large, real-life system: the Linux operating system kernel. I describe TxLinux, a version of Linux, which is the first OS to use transactional memory for synchronization. TxLinux runs on MetaTM, a simulator co-designed with TxLinux, which models an x86-based Hardware Transactional Memory (HTM) system. The TxLinux/MetaTM effort yields a characterization of real-life OS transactions, exposes previously unconsidered complications (including interaction with interrupts and stack memory) and allows sensitivity studies of various TM microarchitectural parameters. It also provides a flexible platform for future OS, TM and architecture research.

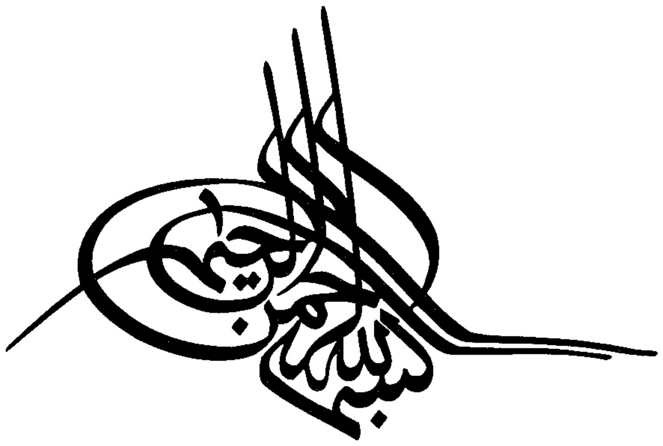
Next, I examine ways to increase concurrency by investigating the factors that inhibit concurrency in existing TM models and systems. These include avoidable implementation limitations, overly restrictive serialization models, and inexpressive APIs. After examining the nature of each limitation, I propose a solution for each one. I postulate that the conventional wisdom that every transaction is "for itself" and primarily relates to other transactions by conflicting with them, is a pervasive misperception. This thesis aims to demonstrate that there are other ways of thinking about the relation of one transaction to another. I present three different transaction models to show how (i) co-existence, (ii) cooperation, and (iii) coordination, can each solve important problems facing TM programmers today.

Co-existence of multiple transactions on the same processor is enabled using the *suspended transactions* model. This model, used by TxLinux, can reduce aborts and removes transaction length limitations imposed by interrupts. Cooperation of transactions that access the same data, using the *dependence-aware transactions* model, can transparently turn transaction aborts into commits. Drawing on serializability theory and notions of spheres of control (which predate ACID transactions), this model is able to accept more execution schedules than any existing TM design. Lastly, the coordination of multiple transactions in the *coordinated sibling transactions* model, provides programmers a simple and unified way of expressing intratransaction parallelism. This helps move transactions beyond being a drop-in replacement for locks (SLE-style) to instead helping programmers find more parallel work within their programs (both in speculative and non-speculative forms). All three models aim at increasing concurrency, while shifting complexity away from the programmer and into the TM system. I evaluate all three models, using either the MetaTM HTM, or one of the several software (STM) systems this thesis also develops.

Contents

List of Tables

List of Figures


بسم الله الرحمن الرحيم

Chapter 1

Introduction

A popular misunderstanding has it that transactions are bound to databases.

- Jim Gray and Andreas Reuter, *Transaction Processing* [29], p.171fn7

Scaling the number of cores on a processor chip has become an industry priority, with a reduced focus on improving single-threaded performance. Developing software that takes advantage of multiple processors or cores remains challenging because of well-known problems with lock-based code, such as deadlock, convoying, priority inversion, lack of composability, and the general complexity and difficulty of reasoning about parallel computation.

Programming with transactions has experienced a renaissance with the advent of these new multicore processors. Transactional memory [49], whether in hardware (HTM) as introduced by Herlihy and Moss [42], or in software (STM) as introduced by Shavit and Touitou [79], enables locks to be replaced with transactions, promising greater ease of use coupled with equal or better performance. Transactional memory has emerged as an alternative paradigm to lock-based programming with the potential to reduce programming complexity to levels comparable to coarse-grained locking without sacrificing performance.

Transactional memory provides the abstraction of atomic, isolated execution of critical regions. By *atomic*, we mean that if a memory transaction fails for any reason its effects are discarded: either all of its updates become globally visible, or none of them do. By *isolated*, we mean that no memory transaction sees the partial effects of any other transaction: uncommitted or speculative state is private to a transaction.

This thesis examines new transactional memory models and systems. These include the MetaTM (Chapter 3) simulated hardware model, which is the first proposed system that enables an operating system, TxLinux (Chapter 4), to use transactional memory. It also introduces the dependence-aware transactional model (Chapter 5) and software implementation (Chapter 6) which allows greater concurrency by committing transactions that would otherwise abort due to conflicts between them. The thesis also introduces the coordinated sibling transactions model (Chapter 7) and implementation (Chapter 8), which focus on making intratransaction parallelism a commodity. Chapter 2 provides basic background material to introduce transactional memory. The rest of this chapter motivates and provides and overview of the models and systems that are part of this thesis.

1.1 MetaTM and TxLinux

> These considerations suggest that operating system kernels may need to support transactions, or transactions will need to be tightly circumscribed in their interaction with the system.
>
> - James R. Larus and Ravi Rajwar, *Transactional Memory* [49], p.45

The first part of this thesis involves the TxLinux and the MetaTM components. TxLinux is a transactional operating system that is based on the Linux kernel. TxLinux is among the largest real-life programs that use HTM, and the first to leverage transactional memory within an operating system kernel. Unlike previous studies that have taken memory traces from a Linux kernel [5], TxLinux successfully boots and runs in a full machine simulator. To create TxLinux, we converted many spinlocks and some sequence locks in Linux to use hardware memory transactions. There are several important reasons to allow an OS kernel to use hardware transactional memory. Many applications (such as web servers) spend much of their execution time in the kernel, and scaling the performance of such applications requires scaling the performance of the OS. Moreover, using transactions in the kernel allows existing user-level programs to immediately benefit from transactional memory, as common file system and network activities exercise synchronization in kernel control paths. Finally, the Linux kernel is a large, well-tuned concurrent application that uses diverse synchronization primitives. An OS is more representative of large, commercial applications than the micro-benchmarks currently used to evaluate hardware transactional memory designs.

I examine the architectural support necessary to allow TxLinux to use hardware transactions. An HTM model, called MetaTM, is also proposed, implemented as a module in the Simics machine simulator. It is based on the x86 ISA and trap architecture. The x86 trap architecture and stack discipline create challenges for the interaction between interrupt handling and transactions. The problems posed by the x86 trap architecture are similar to those posed by other modern processors, and we believe that these problems are not adequately addressed in existing HTM proposals. The *suspended transactions* model is introduced as part of this work, and used by TxLinux.

After developing TxLinux and MetaTM, they are used to examine the characteristics of transactions, using relatively large workloads (millions of transactions). This data should be useful to designers of transactional memory systems. Many transactional memory designs in the literature have gone to great lengths to minimize one cost at the expense of another (e.g., fast commits for slow aborts). The absence of large transactional workloads, such as an OS, has made these tradeoffs very difficult to evaluate.

1.2 Dependence-aware transactions

A popular misunderstanding has it that all existing things, by nature, must be in conflict to survive.

- Alfarabi (10^{th} century philosopher, musician, logician, and social psychologist), paraphrased.

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* [29].

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* [29]. Even TM implementations that do not use locks [33, 56], including both eager and lazy systems, only provide concurrency equivalent to two-phase locking. Any data read or written by one transaction has an implicit lock on it that conflicts with any attempt to write the same data. The key insight of our work lies in taking advantage of the fact that using conflict serializability as the system's safety property increases concurrency relative to using two-phase locking.

This thesis introduces *dependence-awareness*, a transactional memory implementation technique that ensures conflict serializability. *Dependence-aware transactional memory* (DATM) manages conflicts by making transactions aware of dependences, and in some cases, by forwarding data values between uncommitted transactions. Dependence-awareness allows two conflicting transactions that are conflict-serializable to both commit safely, thereby increasing concurrency and making 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.

Most previous proposals to help TM deal with write-shared data 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. 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. Because dependence-awareness admits concurrency where current designs cannot, it provides good system performance without burdening programmers with exotic APIs.

1.3 Coordinated sibling transactions

> Intratransaction parallelism requires genuine support for nested transactions.
>
> -Jim Gray and Andreas Reuter, *Transaction Processing*

To date, TM has addressed only synchronization of threads, not how a program can use more threads to increase throughput or decrease latency. The traditional prescription for intra-transaction parallelism is a nested subtransaction model, extended to allow the nested transactions to run in parallel. This basic model works only if the parallel activity within the transaction is independent. However, the fact that the work was put into the same transaction to start with makes complete independence unlikely. Moreover, some kinds of parallelism, such as speculation, are not exploitable in this model. While this intra-transaction model is conceptually simple, it has not been widely used in a database context, or examined in much depth in the newer TM context.

This thesis suggests that *non-independent* parallel closed-nested transactions are easy to program and can improve performance. Recent TM systems have looked at linear (non-parallel) closed-nesting [58,60], but found its performance lacking [58]. Parallel nesting is as old as nesting itself, and has also been proposed recently in the TM context [2]. However, Agrawal's model treats each nested transaction as completely independent of its siblings. We believe this model insufficiently expresses some of the intuitive sources of parallelism, and should be complemented with siblings that can affect each other's outcomes.

This thesis introduces *xfork*, a programming construct designed to make it easier for programmers to express intra-transaction concurrency. It enables programmers to leverage additional cores to increase performance, while retaining the ease of use of the TM API. Xfork is the mechanism by which programmers create **coordinated sibling transactions**. These transactions are similar to parallel closed-nested transactions, but with added coordination semantics that make them more accessible and useful to programmers.

Coordinated sibling transactions allow programmers to treat parallel nested transactions as a group and to specify the semantics for the group as a whole. Real life shows that some siblings are independent, some act as one unit, and some cannot survive together. We formalize this relationship in our model as the *sibling coordination forms* OR, AND, and XOR, using a rough analogy to the Boolean functions. We show the utility of these forms for TM programmers and the natural fit with TM code structure.

The xfork API can be viewed as a blend of nested transactions, fork/join parallelism, distributed transaction coordination and speculative execution. If programmers are to focus on their application's algorithms, and parallelism within them, the runtime should shield them from the complexities of the low-level implementation of such coordination. The xfork implementation handles the threading and concurrent execution of the different forks, and performs the necessary coordination. Programs built with the xfork API benefit because they can use extra cores to speed up individual transactional units of work.

Chapter 2

Background

This section provides a brief introduction to transactional memory, focusing at a high level on the programming model, and primary strategies for different implementations.

2.1 Motivation: the critical region

Transactional memory, whether implemented in a software or hardware system, is principally a new *programming abstraction*. It assumes a threading model of parallelism, where multiple threads can be executing in parallel on multiple processing cores. The threads can either be directly executing code as specified by application programmers, or perhaps some application framework upon which application components are built. The purpose of the transactional memory abstraction is to solve problems related to synchronization among the different threads[1]. Two key problems are those of synchronizing access to data (i.e. providing mutual exclusion), as well as synchronizing program execution points (i.e. providing coordination). Of course, prior to the introduction of TM there already existed several constructs that

[1]Transactional memory is increasingly being examined for uses other synchronization, for example to help with recovery or security.

provide one or both of these properties for multi-threaded programs. Widely used mechanisms include locks, semaphores, condition variables, monitors and event-objects.

Some of the motivations and advantages of transactional memory can be seen if we consider the effort required on the part of programmers to obtain mutual exclusion. Mutual exclusion allows multiple threads to access shared resources by ensuring that threads do not see each other's intermediate stages of computation with respect to that data. Specifically, programmers first identify critical sections in their code. Critical sections are the regions of code that will access the shared resource (usually certain shared memory variables). This step, the identification of critical sections, is both mandatory and essentially the same, regardless of whether locks or transactions are to be used. Any difficulties in this step - as well as difficulties in the task of figuring out how parallel work will be divided across more than one thread - are not fundamentally made easier by transactional memory (aside from some benefits noted below) The two approaches diverge after the critical sections are identified.

2.2 Locking

When using locks to provide mutual exclusion, the programmer needs to explicitly acquire the appropriate lock at the beginning of each identified critical section, and to release it at the end of the critical section. The primary difficulties of the locking approach arise from figuring out what the *appropriate* locks are. The programmer defines (and names) the locks, and thus is entirely responsible for deciding what locks exist, and deciding on the mapping between locks and shared data. A critical decision is how many locks will be defined for use in the program: a relatively small number of locks (*coarse-grained locking*) or a relatively large number of locks (*fine-grained locking*). There are several significant tradeoffs that this decision involves.

Coarse-grained locking is significantly easier to use in multiple respects. Since fewer locks need to be defined, the mapping from locks to data is simpler, and consequently so is determining which lock is *appropriate* for a given critical section. Program maintainability also improves due to the easier locking discipline. The runtime overhead for lock acquisition is reduced, since fewer locks need to be acquired and released.

However, the greatest benefit of coarse-grained locking is that deadlocks are much easier to avoid. A deadlock occurs when two or more threads acquire locks in such an order that none of them can proceed since they hold locks that others need, while needing locks held by others. While it is possible to detect if a deadlock is about to occur (for example, by testing if a lock acquisition is taking a long time), it is usually not worth programmer's efforts to adopt this approach for any non-trivial program[2]. Therefore instead of handling deadlocks as they arise, most programs that use locks try to avoid deadlocks. The most common technique used to prevent deadlocks is to ensure that locks are acquired in a certain order (either a total or partial order) so as to eliminate the possibility of a deadlock. With coarse-grained locking, it is easier to specify a total order and to ensure the program's components adhere to the specified order.

Turning to fine-grained locking, the disadvantages parallel the advantages of coarse-grained locking: increased complexity of defining the larger number of locks, and the corresponding mapping between individual locks and the shared data each lock covers. Fine-grained locking also incurs higher runtime overhead due to the increased number of lock acquisitions, compared to coarse-grained locking. Fine-grained locking comes an increased effort by programmers to respect a lock ordering, and programmer maintainability suffers, with minor code changes frequently introducing new deadlocks, which are often difficult to catch during program testing.

[2]This requires writing logic to release acquired locks in the middle of the critical section, but only after restoring the system to some consistent state.

The main advantage of fine-grained locking, however, is the potential for greater concurrency compared to coarse-grain locking. While dependent on the actual application and system resources, as well as the contention on the various locks, it is possible for threads to be blocking waiting on one of the coarse-grained locks. While the blocking is necessary if the same data is to be modified, as the granularity of locking becomes coarser, it is less likely that threads contending for a lock actually need to access the same data.

The complexities of fine-grained locking, and to some extent even coarse-grained locking, become qualitatively worse as the software systems grow in size. Large-scale systems use several architectural and design approaches, such as components and software layers, which attempt to leverage separation of concerns and modularity to manage the complexity of the software. Synchronization would ideally be contained within each component or module, as this provides the well-known benefits of encapsulation. However, this makes it even harder to orchestrate proper synchronization to avoid deadlocks: the main issue is that what locks a function may need to acquire would then be unknown to its callers. The alternative is usually to break encapsulation boundaries, treating synchronization as a cross-cutting concern, but at the cost of added complexity. Deadlock freedom, for example, becomes a global property of the system which cannot be reasoned about looking solely at a single component.

2.3 Transactional memory

Transactional memory adopts the same model of parallelism (threading-based) as locks, and also leverages the same concept of identifiable critical sections within which shared data is accessed. However, it operates at a higher level, by allowing the programmer to simply demarcate the critical section as a *transaction*. This eliminates many pitfalls of locking (such as worrying about deadlock cycles due to lock ordering), and in several other respects is less error-prone (for example, ensuring that a lock was acquired in the appropriate mode, such as for reading or reading and writing). It is still the responsibility of the programmer to identify critical sections, but assuming this is done correctly, the programmer is guaranteed certain semantics about how those transactions will execute. The responsibility of how these semantics are provided is shifted to the underlying system implementation. Many proposals have been advanced, which can provide either hardware, software, or hybrid hardware-software implementations, with various implementation trade-offs, extensions to the basic model, and varying semantics. However, at its core every proposed TM system is ultimately conncerned with executing programmer-demarcated critical regions with a basic guarantee of mutual exclusion that allows them to work as a replacement programming construct for locks.

The primary constructs used to demarcate regions of program code to be a transaction will differ based on the system implementation. For example, in a hardware transactional memory (HTM) system, a typical interface would perhaps involve extending the architecture instruction set to include an instruction to start a transaction and one to end a transaction. In a software transactional memory (STM) system, a programming language may be extended to allow scoped *atomic blocks*, with all code within these blocks comprising a transaction. Other language design choices include for example a modifier that specifies that a given method is to be transactional, in which case all the code within the method executes transactionally. Managed languages (such as Java and C#) in which access to memory can be intercepted by the runtime (also called the virtual machine or execution engine) tend to have proposals with the simplest TM interfaces. The issue becomes more complex for unmanaged languages (such as C and C++), which may need more complicated interfaces, for example macros are often used by the transactional code when accessing shared data. In any case, it is the system that keeps track of which memory a transaction has accessed. This metadata is called a transaction's *working set*, which is composed of a *read set* and *write set*.

2.4 Transaction properties

What exactly are the properties that a TM system will provide to the programmer? While there are several ways of analyzing these properties[3] the most common approach is borrowed from the field of databases, the theory of transaction serializability. Database transactions are usually described as providing ACID properties: Atomicity, Consistency, Isolation and Durability. Atomicity means that a transaction will execute in an all-or-nothing manner. Therefore, if for some reason a transaction is unable to finish execution completely, it will not terminate half-way. Rather the implementation will ensure that the system state will be as if the transaction had not executed at all. Note that atomicity is a property that needs to be guaranteed even if no other transactions are concurrently executing. The second property, isolation, guarantees that each transaction executes as if were executing alone in the system. Thus intermediate results of other transactions are not visible to it and vice versa. The consistency property of transactions is that each committed transaction will transform the system from one valid state to another valid state. This property depends on the programmer ensuring the logic within each transaction is correct, and transaction boundaries are demarcated correctly. However, the underlying system can help ensure consistency. Databases for example usually allow for trigger functions to run as part of transaction commit, which ensure the system state is still consistent. For example, constraints between tables (e.g. foreign key

[3]For example, linearizability [44] provides higher-level description correctness of an abstract datatype in terms of the history of method invocations on a concurrent object. An execution is linearizable if all invocations and responses form a legal sequential history. A datatype is linearizable if all executions are linearizable. Transactional memory can also be analyzed in terms of shared memory as the abstract data type, however our discussion will be restricted to the serializability model.

constraints) are automatically verified, and a transaction that violates such a constraint is not allowed to commit. Finally, the durability property ensures that once a transaction has committed, then it is guaranteed to stay committed, within a specified failure model. For example, assuming no disk failures, a database transaction that commits is guaranteed to stay committed even if the machine experiences an unexpected reboot. The semantics of a TM transaction have focused on providing the atomicity and isolation properties of database transactions. Durability is not a property since the resources that transactional memory updates are volatile shared memory data.

A TM transaction can have one of three different outcomes: it may commit, abort, or be in an undefined state (i.e. still active, possibly even in an infinite loop). A transaction that has committed is one that has executed all the instructions within the critical section, and usually has modified some shared memory state and potentially affected control flow. Several systems also allow a transaction to abort, where control flow continues after the transaction code (or atomic block), but with no side effects on system state from the transaction. An abort can be due to an explicit abort command by the programmer, or can be the result of an unexpected exception. This is to be distinguished from cases where the runtime automatically decides to retry a transaction's execution, in which case the transaction first aborts, but then control flow transfers to the beginning of the transaction block for another execution attempt of the transaction (with intermediate state from the previous attempt being discarded). This latter kind of abort and restart arises, for example, in systems where isolation violations are avoided by forcing some transactions to abort and restart. An important benefit of transactions is that programmers can reason about system state changes at a higher level of granularity, that of an entire transaction block, which commits or aborts as a unit.

2.5 Version management

To provide the atomicity property, implementations must ensure that a transaction which has partially executed can at any time be aborted without affecting system state. This entails that if the transaction modifies data, the system must somehow ensure that the original value of the data (before the transaction started) are at all times available in case the transaction aborts. There are two primary approaches for performing this. The first is to leave the original value intact, and write the modified value into a separate per-transaction buffer, which is only copied over to the actual memory locations if the transaction commits. This approach is called *lazy version management* in HTM systems, and *deferred update* in STM systems. Another approach is to instead copy the original value to a per-transaction log-like structure, and allow the transaction to update the shared data in its original location. This approach is called *eager version management* in HTM systems and *direct update* in STM systems. In this approach, the logged values can be discarded if the transaction commits successfully, but if the transaction aborts the values are copied back to the original shared memory locations.

There are several other important decisions related to version management: a key one is the granularity of data. In STM systems, the main approaches taken are either word-level granularity or object-based granularity. In HTM systems, most systems use cache-line granularity. This is because in HTM systems the per-core cache is often used as a place where the speculative data is stored (whether with eager or lazy-version management). Abort and commit can be designed to be local cache operations. The version management strategy must be compatible with the conflict detection strategy, to ensure that the isolation and correctness are preserved. One of the issues that must be taken into account is if the granularity of conflict detection is smaller than the granularity of version management. For example, if conflict detection is done at a word-level, while version management is performed on an entire cache line, special care is needed to avoid lost update problems (similar to false sharing).

2.6 Conflict management

There are several concepts that are common to both hardware and software implementations, such as conflicts. A conflict occurs when two active transactions are accessing the same shared memory state and at least one of the accesses is a write. If both transactions are simply allowed to proceed, this would be a violation of the transaction isolation property. Therefore TM implementations usually intervene so as to preserve isolation. It is best to conceptually distinguish the *occurrence* of the conflict, its *detection* by the system, and the *resolution* of the conflict. These three events can happen together, or they can happen at different times, reflecting different implementation design choices. For hardware systems, systems can be roughly divided into those in which detection and handling of conflicts occurs as they occur (eager conflict management), and those in which conflicts are detected only when at least one of the transactions has completed and attempts to commit (lazy conflict management).

Conflicts are handled by either causing one of the transactions involved to abort and restart, or by delaying execution of one of the transactions. Note that certain choices may lead to starvation, livelock, or deadlock, but implementations usually take the responsibility of dealing with these anomalies such that they are not visible to the programmer. Bad contention management decisions should only manifest as lower performance for the programmer, but never as incorrect system behavior. The decision of what action to take in response to the conflict, and which transaction should win the conflict, comprises the *contention management* strategy of a given implementation. Various strategies have been proposed, from very simple ones such as oldest transaction wins (called the *timestamp* policy), to sophisticated ones that take into account more factors (e.g. transaction size, previous restarts, etc.).

The approaches taken by TM implementations to deal with conflicts differ across software and hardware implementations. In hardware, many systems (such as LogTM) are able to leverage existing cache coherence protocols (such as MESI) to inform them when other processors are accessing the same data (usually at the level of the cache line). These systems consequently tend to have an eager conflict management strategy. Other systems (such as TCC) have proposed new cache coherence protocols to implement lazy conflict management, where coherence traffic occurs at transaction commit time. The functionality of existing protocols such are MESI are not needed in these systems, since it is assumed that processors will always be running in a transaction (i.e. when any transaction commits, another is immediately started on the same processor). Signatures have also been proposed for use in hardware systems of both types (eager and lazy). A transaction signature used a fixed-size bitmap to summarize the shared memory locations that a transaction has accessed. The technique draws on bloom filters and the idea of hashing memory addresses to turn on certain bits in the bitmap. While it is possible to have false positives when checking if two transactions conflicts, correctness is always guaranteed if signatures are used for conflict detection. Hybrid software-hardware systems are able to leverage hardware coherence protocols to allow software layers to efficiently detect when conflicts occur.

Transactional memory systems implemented in software-only have to implement conflict detection using software metadata structures. Basically associated with each shared memory object (whether a word or an object, depending on the type of STM system), the system must keep track of which transactions are accessing that item within a transaction. When another transaction attempts to access the same item, the metadata will allow the conflict to be detected and handled appropriately. One way to classify systems is how transactional readers are handled. To reduce the metadata space and runtime overhead, systems may choose to have *invisible* readers, in which case the metadata doesn't actually keep track of which transactions have only read the shared data. In this case, a writer will not detect a conflict with any readers, but rather will proceed to update the shared memory item. Isolation is usually guaranteed by having any readers detect this violation with the use of timestamps that enable readers to detect at commit time whether any writers had intervened. Another approach is to have *visible* readers, where information about readers: from as little as a flag indicating that there exists one or more readers, to information about which transactions are reading. Each of these alternatives enables more sophisticated contention management policies, but at the cost of additional overhead.

One consequence of approaches that defer conflict detection is that in some systems transactions may end up executing at times without the benefits of isolation. While the system will ultimately detect the violation, the inconsistent execution (of what are called zombie transactions[4]) may result in behavior such as exceptions or failed assertions. These may be tolerated easily in a managed language setting, for example by retrying the transaction. More problematic is the possibility of a program entering an infinite loop. One way to handle this is to perform an eager validation on the backedge of loop control flow. Inconsistent execution may manifest itself by exceptions in managed type-safe programming languages such as Java or C#. Unsafe languages (such as C or C++) can cause incorrect behavior without raising an exception.

Another issue that implementations must deal with is the semantics of interaction of transactions and non-transactional memory accesses. This is usually a

[4]A zombie transaction is one that continues executing even though it will not be allowed to abort. What makes a zombie transaction different from a normal doomed transaction, is that it is executing with an inconsistent view of the system state, and thus may execute arbitrary code which, depending on the system, may or may not be reversible when the zombie is aborted.

greater concern in software implementations, since most hardware implementations are able to provide strong semantics without the high performance overhead that would be required in an software implementations. Even in an STM, if a program ensures that any shared data that is accessed in a transaction is never accessed outside a transaction, then well-defined semantics are easy to provide. *Weak isolation* STMs assume that programmers will ensure that this is the case. Otherwise, *strong isolation* is required, where the STM ensures that even non-transactional memory accesses will have well-defined semantics when interacting with transactions. An easy to understand semantic is single global lock atomicity (SGLA), where the semantics would be the same as if every transaction block was replaced with a critical section that uses a single global lock. Programs that have no data races have SGLA semantics, whether weak or strong atomicity is provided by the STM. Other progressively weaker semantics have been proposed, which have less overhead than SGLA implementations, but require more care on the part of programmers [54].

2.7 Nesting

One of the advantages of transactions is that it is easier (compared with locks) to compose critical regions that are written in different components that interact. For example, a call from one critical region that is holding a lock, into another component, which will attempt acquire another lock, may deadlock if ordering is not taken into account. With transactions, if an active transaction calls into another component, which then starts another transaction, the composition does not require any foreknowledge on the part of the programmer. This situation is an example of nested transactions, and systems can provide different types of nesting semantics. The simplest to understand, and most popular, is *flat nesting.* What happens is that if a running transaction (the outer one) encounters code that attempts to start another transaction (the inner one), the inner transaction simply gets subsumed under the outer transaction. Therefore, any aborts caused by the inner transaction result in the outer transaction aborting. Also, the modifications made by the inner transaction are visible to the outer transaction, but are not actually committed to the rest of the system until the outer transaction commits. It is as if the inner transaction's transaction boundaries were simply not there, in the case where an outer transaction is already active. In practice, systems can implement flat nesting with a simple per-core nest-counter, that keeps track of how many levels of nesting have occurred, using it to determine when the outermost transaction is actually committing.

Another kind of nesting is *closed nesting*, which has similar semantics to flat nesting if the inner transaction commits. However, the inner transaction can abort and restart without aborting the outer transaction. The advantage is less work needs to be retried when the inner transaction aborts (which can be important if the nested transaction is used to access data that is highly contended). However, closed nesting has a higher cost in terms of implementation complexity than flat nesting. Lastly, *open nesting* is the most sophisticated type of nesting, where the inner transaction's changes are made visible to all immediately when it commits. Even if the outer transaction aborts, the inner transaction's changes aren't rolled back if open nesting is used. This effectively provides the capability of releasing isolation on specific shared data in the middle of a transaction. However, the programmer needs to guarantee that the overall program semantics are still correct. Systems that provide open nesting usually allow the outer transaction to register a custom compensating action to be executed if the outer transaction aborts. The compensating action can proceed to undo the effects of the previously committed open nested transaction. However this compensation action runs after the open nested transaction has committed, and must take into account that the shared data could have been accessed and modified by other transactions in the meanwhile. Reasoning about compensation actions and the early release of isolation makes open nesting harder to program with, except for fairly simple and limited scenarios. Moreover, implementation complexity is greater than closed nesting.

2.8 Further background

Larus and Rajwar's survey [49] provide an excellent introduction to transactional memory. Our related work section provides additional information about specific research that is related to this thesis. We conclude this section by mentioning some of the main issues and directions TM researchers have looked at. Some of these are orthogonal to the work in this thesis. Those that are not will be introduced in more detail throughout the thesis where appropriate.

There are many other high-level TM questions related to coordination among transactions, as well as interaction of transactions with OS system calls, context switches, garbage collectors, finalizers, I/O, and other programming and system abstractions. The design space of hardware, software and hybrid implementations continues to be incrementally explored. For example, to consider HTM, proposals have been advanced that use standard bus-based protocols, as well as those that extend directory-based coherence protocols. The issue of virtualization - what to perform when on-chip hardware resources are exceeded (e.g. if the transaction working set exceeds the on-chip cache) has been the focus of many researchers. There are ongoing attempts to apply transactional memory to a variety of software systems (from compilers to game engines [84]).

Chapter 3

MetaTM Model

This chapter examines the architectural features necessary to support hardware transactional memory in the Linux kernel for the x86 architecture. It includes new proposals for interrupt-handling and thread-stack management mechanisms. Section 3.1 describes MetaTM a novel model for hardware transactional memory. Section 3.2 describes the mechanisms used to properly handle interrupts in a transactional operating system kernel, and Section 3.3 describes the related issue of dealing with stack memory.

3.1 Architectural model

In order to evaluate the how system performance is affected by different hardware design points, system called MetaTMwas built. MetaTM includes novel modifications necessary to support TxLinux. This section describes architectural features present in MetaTM.

Primitive	**Definition**
xbegin	Instruction to begin a transaction.
xend	Instruction to commit a transaction.
xpush	Instruction to save transaction state and suspend the current transaction.
xpop	Instruction to restore transactional state and continue the **xpush**ed transaction.
Contention policy	Choose a transaction to survive on conflict.
Backoff policy	Delay before a transaction restarts.

Table 3.1: Transactional instructions and policies in the MetaTM model.

3.1.1 Transactional semantics

Table 3.1 shows the transactional features in MetaTM. Starting and committing transactions with instructions has become a standard feature of HTM proposals [53], and MetaTM uses **xbegin** and **xend**. HTM models can be organized in a taxonomy according to their data version management and conflict detection strategies, whether they are eager or lazy along either axis [56]. MetaTM uses eager version management (new values are stored in place) and eager conflict detection: the first detection of a conflicting read/write to the same address will cause transactions to restart, rather than waiting until commit time to detect and handle conflicts.

MetaTM supports multiple methods for resolving conflicts between transactional accesses. One way of resolving transactional conflicts is to restart one of the transactions. MetaTM supports different contention management policies that choose which transaction restarts. Alternately, if a transaction requests a cache line owned by a different transaction, it can be stalled until the other transaction finishes. This stall-on-conflict policy requires deadlock detection to avoid circular waits. Whether a transaction waits before restarting and by how much is governed by the the backoff policy. MetaTM does not support an explicit abort or restart primitive, as TxLinux does not currently require either of these. MetaTM supports strong atomicity [11], the standard in HTM systems where a conflict between a non-transactional memory reference and a transaction is treated as a conflict between two transactions and thus always aborts and restarts the transaction.

MetaTM does not assume a particular virtualization design. When a transaction overflows the processor cache, MetaTM charges an overflow penalty (of 500 cycles) to model initialization of overflow data structures. Any reference to a cache line that has been overflowed must go to memory. MetaTM manages and accounts for the cache area used by multiple versions of the same data.

The cost of transaction commits or aborts are also configurable. Some HTM models assume software commit or abort handlers (e.g. LogTM specifies a software abort handler); a configurable cost allows the exploration of performance estimates for the impact of running such handlers.

3.1.2 Managing multiple transactions

MetaTM supports multiple active transactions on a single thread of control [71]. Recent HTM models have included support for multiple concurrent transactions for a single hardware thread in order to support nesting [53, 57, 60]. Current proposals feature close-nested transactions [53, 57, 60]. Open-nested transactions [53, 57], and non-transactional escape hatches [57, 83]. In all of these proposals, the nested code has access to the updates done by the enclosing (uncommitted) transaction. MetaTM provides completely independent transactions for the same hardware thread managed as a stack. Independent transactions are easier to reason about than nested transactions. The hardware support needed is also simpler than that needed for nesting (a small number of bits per cache line, to hold an identifier). There are several potential uses for independent transactions. TxLinux uses them to handle interrupts, as will be discussed in Section 3.2.

xpush suspends the current transaction, saving its state so that it may continue later without the need to restart. Instructions executed after an **xpush** are independent from the suspended transaction, as are any new transactions that may be started—there is no nesting relationship. Multiple calls to **xpush** are supported. An **xpush** performed when no transaction is active, is still accounted for by the hardware (in order to properly manage **xpop** as described below). Suspended transactions can lose conflicts just like running transactions, and any suspended transaction that loses a conflict restarts when it is resumed. This is analogous to the handling of overflowed transactions [26, 68], which also can lose conflicts.

xpop restores a previously **xpush**ed transaction, allowing the suspended transaction to resume (or restart, if it needs to be). The **xpush** and **xpop** primitives combine suspending transactions and multiple concurrent transactions with a LIFO ordering restriction. Such an ordering restriction is not strictly necessary, but it may simplify the processor implementation, and it is functionally sufficient to support interrupts in TxLinux. While **xpush** and **xpop** are implemented as instructions in MetaTM, they could also be implemented by a particular HTM design as groups of instructions. Suspending and resuming a transaction is very fast, and can be implemented by pushing the current transaction identifier on an in-memory stack.

3.1.3 Contention management

When a conflict occurs between two transactions, one transaction must pause or restart, potentially after having already invested considerable work since starting. Because transaction restarts cause threads to repeat work, there is potential for transactions to perform poorly when contention is high. Contention management is intended to reduce contention in order to improve performance. MetaTM model supports the contention management strategies proposed by Scherer and Scott [78], adapted to an HTM framework. Because hardware transactions do not block in our model (they can execute, restart, or stall, but cannot wait on a queue), certain features required adaptation. The interaction of suspended transactions (via **xpush**) and contention management is discussed in Section 3.2.5.

3.1.4 Backoff

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 the absence of an explicit notification mechanism, the decision for how long to wait is heuristic. The MetaTM model supports using different backoff strategies, whose impact on workloads is measured.

Previous work has focused on exponential backoff strategies. The following list summarizes the backoff policies explored by MetaTM [78].

- **Exponential** – Exponential Backoff is implemented by choosing a random seed between 1 and 10. The number of times the conflicting transaction has backed off is raised to the power of 2, and multiplied by the seed to determine the number of cycles the conflicting transaction should wait before restarting.
- **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.

3.1.5 Device initiated memory operations

Memory operations initiated by devices (rather than by instructions) are not part of any transactional context in MetaTM. For instance, when the TLB reads from the page table, the read is not entered into the current transaction's working set. TxLinux does not change the kernel's protocol for maintaining TLB coherence. When a processor takes an interrupt in kernel mode, it stores state on the kernel stack. Such stores cannot be transactional because the trap architecture is not transactional, and no facility exists to re-raise an interrupt on a transaction restart.

3.2 Interrupts and transactions

The x86 trap architecture and stack discipline create challenges for the interaction between interrupt handling and transactions. The problems posed by the x86 trap architecture are similar to those posed by other modern processors; these problems are not adequately addressed in existing HTM proposals. This section presents a microarchitectural design for the interaction of interrupts and transactions that adds minimal hardware complexity while maintaining ease of use and efficiency for transactions. Other recent work [26] showed that solutions that do not abort active transactions to handle interrupts provide better system performance, which validates some of our initial assumptions.

This section provides background on interrupt handling, as well as how existing HTM systems deal with interrupts. It then covers the motivation for how MetaTM and TxLinux deal with interrupts. The mechanism for interrupt handling in TxLinux is covered in Section 3.2.3, and the implications for contention management in Section 3.2.5. The next section (Section 3.3), will explore issues related to the interaction of stack memory and transactions, which arise in part because of the proposed interrupt-handling strategy.

3.2.1 Interrupt handling background

One primary function of the operating system is to respond to interrupts, which are asynchronous requests from devices or from the kernel itself. Interrupt handlers in Linux are split into two halves. The top-half interrupt handler runs in response to a device interrupt that signals the completion of some work, e.g., the read of a disk block. While top-half interrupt handlers are executing, they disable all interrupts at equal and lower priorities to ensure forward progress. To keep system response latency low, top-half interrupt handlers have relatively short execution paths, pushing as much work as possible into a deferred function, or bottom half. Linux checks for and runs deferred functions, which can be long, in several places. The exact taxonomy of deferred functions in Linux is complex, but deferred functions run asynchronously with respect to system calls, just like device interrupt handlers. The operating system does a significant amount of work at the interrupt level, including memory allocation and synchronized access to kernel data structures.

Linux interrupt handlers are a prime candidate for the programming simplicity of transactions, provided the transactional hardware can provide equivalent performance to the fine-grain locking on which they currently rely.

3.2.2 Interrupts in existing HTM systems

Most previous work on HTM systems (see related work in Chapter 9.6 for details) assume that processor interrupts can be treated in the same way as context switches, and thus be aborted. This arises from several assumptions about the interaction of interrupts and transactions. These works assume that transactions are short and that interrupts are infrequent enough to rarely occur during a transaction. As a result, efficiently dealing with interrupted transactions is unnecessary. They assume that interrupted transactions can be aborted and restarted, or their state can be virtualized using mechanisms similar to those for surviving context switches. Other systems have made assumptions about the flexibility of interrupt routing to different processors. We first review how MetaTM handles interrupts, and the factors which such a design takes into account compared with existing approaches.

3.2.3 Interrupt handling in TxLinux

Consistent with the assumptions that interrupts are frequent, that transactions will grow in length, and that interrupt routing is less flexible than considered in other systems, MetaTM is designed to handle interrupts without necessarily aborting the current transaction. In TxLinux, interrupt handlers use the **xpush** and **xpop** primitives in order to suspend any current transaction when an interrupt arrives.

Interrupt handlers in TxLinux start with **xpush** to suspend the currently running transaction. This allows the interrupt handler to start new, independent transactions, if necessary. The interrupt return path ends with an **xpop** instruction. There is no nesting relationship between the suspended transaction and the interrupt handler. Multiple (nested) interrupts can result in multiple suspended transactions.

3.2.4 Factors influencing interrupt handling strategy

This section presents some of the factors that affect and influence the design of interrupt handling in an HTM system. These include transaction length, interrupt frequencies, and interrupt routing limitations.

Transaction length

One of the main advantages of transactional memory programming is reduced programmer complexity due to an overall reduction in possible system states. Coarse-grained locks provide the same benefit, but at a performance cost. The majority of the benchmarks used for research have focused on converting existing critical sections to transactions. Those critical sections were defined in the context of pessimistic concurrency control primitives, and thus were kept short for performance reasons. Because these short critical sections can be quite complex, future code that attempts to capitalize on the programming advantages of TM will likely produce transactions that are larger than those seen today.

Interrupt frequency

Our data shows much higher interrupts rates than e.g., Chung et al. [26] who assume that I/O interrupts arrive every 100,000 cycles. For the MAB benchmark, which is meant to simulate a software development workload (see Section 4.2.1 for the full description), an interrupt occurs every 24,511 non-idle cycles. The average transaction length for TxLinux running MAB is 896 cycles. If the average transaction size grows to 7,000 cycles (a modest 35 cache misses), then 31.2% of transactions will be interrupted.

Interrupt routing limitations

Most interrupts should be handled on a particular processor. The most common source of interrupts are page faults and the local advanced programmable interrupt controller (APIC) timer interrupt. Page faults should be handled locally because they cause a synchronous processor fault. The local APIC timer interrupt must be handled locally for the OS to provide preemptive multitasking. The third largest source of interrupts on TxLinux are interprocessor interrupts, which also must be handled by the local CPU for which they are intended. For the MAB workload, 96% of interrupts are page faults, 2.5% are local timer interrupts (TxLinux is configured with high resolution timers), 0.5% are inter-processor interrupts and 0.4% are device interrupts that can be handled by any processor.

Chung et al. [26] propose routing interrupts to the CPU best able to deal with them, though TxLinux must process 99% of its interrupts on the CPU on which they arrive. Even if interrupt routing were possible, it is unclear how the best CPU is determined. While CPUs in XTM are always executing transactions, CPUs might or might not be executing a transaction in other HTM models like LogTM and MetaTM. A hardware mechanism that indicates which CPU is currently not executing a transaction would require global communication and could add significant latency to the interrupt handling process. The best interrupt routing strategy is also unclear: it may be better for system throughput to route an interrupt to a processor that is executing a kernel-mode transaction rather than to a processor that is executing user-mode code that is not in a transaction.

3.2.5 Interrupts and contention management

Timestamp-based contention management has been a common default for HTM systems [66,67] because it is simple to implement in hardware and it guarantees forward progress. However, in the presence of interrupts and multiple active transactions on the same processor, timestamp-based contention management will cause livelock. Consider a transaction A, which runs and is subsequently suspended via an **xpush** when an interrupt arrives. A second transaction B, started by an interrupt handler conflicts with A. Because it is more recent, a timestamp-based policy dictates that B will lose the conflict. If B restarts, it will continue restarting indefinitely because A is suspended. This problem applies to any contention management policy where a suspended transaction will continue to win over a current transaction. Consequently, suspended transactions *require* modification of basic hardware contention management policies to favor the newest transaction when transactions conflict on the same processor.

3.3 Stack memory and transactions

Some previous work has assumed that stack memory is not shared between threads and has therefore excluded stack memory from the working sets of transactions [31]. However, stack memory is shared between threads in the Linux kernel (and in many other OS kernels). For example, the `set_pio_mode` function in the IDE (hard disk) driver adds a stack-allocated request structure to the request queue, and waits for notification that the request is completed. The structure is filled in by whichever thread is running on the CPU when the I/O completion interrupt arrives.

On the x86 architecture, Linux threads share their kernel stack with interrupt handlers. The sharing of kernel stack addresses requires stack addresses to be part of transaction working sets to ensure isolation. Interrupt handlers will overwrite stack addresses and corrupt their values if stack addresses are not included in the transaction working set. Even when stack addresses are included in transaction working sets, there is a correctness problems (Section 3.3.2) and a performance problem (Section 3.3.3).

3.3.1 Transactions that span activation frames

Many proposals to expose transactions at the language level [4, 14, 15] rely on an `atomic` declaration. Such a declaration requires transactions to begin and end in the same activation frame. Supporting independent **xbegin** and **xend** instructions complicates this model because calls to **xbegin** and **xend** can occur in different stack frames. Linux heavily relies on procedures that do some work, grab a lock, and later release it in a different function. To minimize the software work required to add transactions to Linux, MetaTM does not require **xbegin** and **xend** to be called in the same activation frame.

A simplified version of TxLinux code is depicted in Figure 3.1, where the kernel starts a transaction in one activation frame (`atomic_dec_and_xbegin`, where

```
int atomic_dec_and_xbegin(atomic_t *atomic,
                          spinlock_t *lock) {
   int ret_code = 0;
   xbegin;    /* was spin_lock(lock) */
   if (atomic_dec_and_test(atomic)) {
       ret_code = 1;
   } else {
       xend;  /* was spin_unlock(lock) */
   }
   return ret_code;
}
void dput(struct dentry *dentry) {
   if (atomic_dec_and_xbegin(&dentry->d_count,
                             &dcache_lock)) {
      d_free(dentry);      /* calls call_rcu */
      xend;/* was spin_unlock(&dcache_lock); */
   }
}
```

Figure 3.1: A simplified and slightly modified version of code from TxLinux to release a directory cache entry.

the pre-transaction code would grab a spinlock) and ends it in another (`dput`). Requiring that stack memory be part of a transaction and that transactions be able to span activation frames introduces two issues with interrupt handling: a correctness problem and a performance issue.

3.3.2 Live stack overwrite problem

Figure 3.2(A) shows the stack memory where TxLinux starts in the function `dput` (t0). It calls `atomic_dec_and_xbegin`, sets the value of local variable `ret_code` with a non-transactional store, and then starts a transaction (t1). The code then returns to `dput` (t2). While in `dput`, the CPU on which the kernel thread is executing gets interrupted (u3). The x86 trap architecture specifies that interrupts that do not cause a protection switch (e.g., an interrupt when the processor is already in kernel mode) use the current value of ESP[1]. Therefore the processor (non-transactionally) saves

[1]ESP is the stack register in the x86 architecture, which keep track of the current top of the stack.

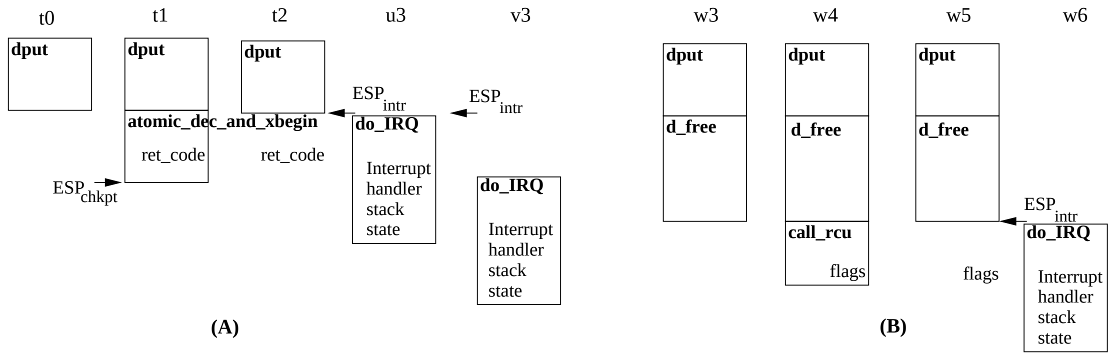


Figure 3.2: The figure shows an animation of process stack memory across time. The steps are labeled with a letter and a number: the numbers indicate temporal order and the letters indicate different timelines, so t0,t1,t2,v3 is one sequence of events and t0,t1,t2,w3,w4,w5,w6 is another. Each box is an activation frame, and in the upper left of each frame in bold is the name of the procedure. Within the frame, and right justified, are the names of local variables, such as `ret_code`. Section (A) depicts a live stack overwrite problem. In steps t0–t2, a transaction starts in one function (`atomic_dec_and_xbegin`) which then returns and then an interrupt arrives. Two alternatives are shown for the interrupt at times u3 and v3. Section (B) depicts a transactional dead stack problem. It picks up after t2 with an alternate timeline starting at step w3. Here `dput` calls `d_free` which calls `call_rcu` which returns and then an interrupt arrives. The handler conflicts with the `flags` variable, even though the variable is dead. ESP_{chkpt} is the value of the stack pointer when the transaction starts and the register values are checkpointed. ESP_{intr} is the value of the stack pointer when an interrupt arrives. Stacks grow down, toward lower numbered addresses.

registers at the point labeled ESP_{intr}, the stack value when the interrupt arrives. The interrupt handler executes an **xpush** which suspends the current transaction, and then the interrupt handler can non-transactionally store local variables on the stack, including overwriting the value of `ret_code`.

When the interrupt handler finishes, it **xpop**s back into the transaction that it interrupted. If the transaction needs to restart, the stack pointer is reset to the checkpointed value, ESP_{chkpt}, which is the value of ESP when the transaction began. The transaction began in a stack frame where `ret_code` is a live variable. However the value of `ret_code` has been overwritten by non-transactional stores in the interrupt handler. `atomic_dec_and_xbegin` also updated `ret_code` non-transactionally, so the re-execution of the transaction is incorrect. The value of a live stack-allocated variable has changed. This situation is referred to as the live stack overwrite problem.

Several factors contribute to the live stack overwrite problem:

1. Calls to **xbegin** and **xend** occur in different stack frames.
2. The x86 trap architecture reuses the current stack on an interrupt that does not change privilege level.
3. A transaction that is suspended (due to an interrupt) can restart.

To eliminate live stack overwrites, a simple change to the trap architecture of the x86 is proposed—if a transaction is active during an interrupt, and the value of ESP_{intr} (the ESP value at the time of the interrupt) is larger than the ESP_{chkpt} (the ESP value at the start of the transaction), then start the interrupt handler stack frame at ESP_{chkpt}. The processor writes the value of ESP_{intr} on the stack, because the x86 specifies that the processor save several registers to the stack on an interrupt (that does not change privilege level) including ESP. But the processor writes ESP_{intr} at the location of ESP_{chkpt} in Figure 3.2. A ESP_{chkpt} value "protects" all stack values above it by not allowing interrupt handlers to write into the region of the stack that is active when the transaction begins. This process is depicted in v3, which follows step t2.

This change is straightforward for the most comprehensive register checkpoint design in the literature [5]. In that design, checkpointed registers are not returned to the free register pool until the **xend** instruction graduates. The processor compares the current ESP with the last checkpointed ESP, and if the latter were a lower address, it copies the content of ESP_{chkpt} to ESP before saving registers and starting the interrupt handler. When the interrupt handler returns, ESP is restored to ESP_{intr} (the value stored on the stack), not the last checkpointed ESP.

3.3.3 Transactional dead stack problem

Conflicts on stack addresses can cause performance problems in addition to the live overwrite correctness problems already discussed. Dead portions of the stack e.g., from a completed procedure call, remain in a transaction's working set, and these addresses can cause spurious conflicts. If transaction A is active when an interrupt arrives, and an interrupt handler suspends A, and then uses the same stack memory of the thread that started transaction A. The handler can conflict with dead stack frame addresses that are still in A's transaction set. Because the stack memory was no longer in use, yet remained in the transaction's working set, it caused an unnecessary restart. This problem still occurs even with the above fix for live stack overwrites.

Figure 3.2(B) shows a case where the interrupt handler needlessly interferes with a suspended transaction. TxLinux starts in `dput` (t0), and calls `atomic_dec_-and_xbegin`, where it starts a transaction (t1). The code returns to `dput` (t2). `dput` calls `d_free` (w3) which calls `call_rcu`, which has a local variable called `flags` (w4). All of these function calls are within the scope of the current transaction, so all writes to the stack frame are part of the transaction's write set. `call_rcu` returns (w5), and an interrupt arrives (w6). ESP_{intr} is at a lower address than ESP_{chkpt}, so this is not a potential live stack overwrite. However, the interrupt handler will overwrite stack locations written during the activation of `call_rcu` (e.g., `flags`). These writes will cause the interrupt handler to conflict with the suspended transaction, even though the suspended transaction no longer cares about the stack state from that activation frame.

A new mechanism is proposed, *stack-based early release*, to avoid such false conflicts on stack memory. Early release [43, 80] is the explicit removal of memory addresses from a transaction's working set before that transaction completes. During an active transaction, any time the stack pointer (ESP) is incremented, if the new value of ESP is on a different cache line from the old ESP, then the processor releases the old cache line(s) from the transaction set. This works on the x86 (and is specific to it) because the stack pointer is constantly adjusted, only via ESP, to delimit the live range of the stack. On the x86, the stack pointer is not related to the frame pointer (if the compiler allocates a frame pointer), and the hardware is allowed to write the address contained in ESP at any time because an interrupt might arrive at any time.

In Figure 3.2(B), when the processor returns from `call_rcu`, it will release the line that has the `flags` variable. While procedure returns might release several lines, almost every other ESP incrementing instruction (e.g., pop) will release at most one cache line. Because `flags` is early released, the interrupt handler will not conflict with it, even if it writes that stack address. Stack-based early release is a performance optimization for MetaTM.

Chapter 4

TxLinux Evaluation

This chapter describes the modifications made to the Linux operating system kernel to run on the MetaTM model described in the previous chapter. It then evaluates the performance of this modified operating system.

4.1 Modifying Linux to use HTM

This section describes the modifications made to the Linux kernel version 2.6.16.1 to support transactions. A natural approach to converting an existing code-base to use transactions is to focus on replacing existing synchronization primitives. Much of the current literature on hardware transactional memory assumes a simple programming model for lock-based code that does not capture the diversity of synchronization primitives used in Linux. Linux supports spinlocks, reader/writer spinlocks, atomic instructions, sequence locks (seqlocks), semaphores, reader/writer semaphores, completions, mutexes, read-copy-update (RCU), and futexes[1]. Each primitive has a different bias, such as favoring readers or writers when contention exists. To create TxLinux, the following subset of those primitives were modified:

[1]Futex (short for fast-userspace-mutex) are a Linux synchronization construct, added to the 2.6 kernel. [22, 28]

- **Spinlocks.** These are the the most popular primitive in the Linux kernel by static count—over 2,000 call sites. They are intended for short critical sections where the caller spins (polls) while waiting for the critical section. TxLinux has substituted transactions for spinlocks, reader/writer spinlocks and variants which disable interrupts or soft-irqs. Conversion of spinlocks to transactions is straightforward: lock acquires and releases map to transaction begins and ends respectively.
- **Atomic instructions.** Atomic instructions guarantee that a single read-modify-write operation will be atomically committed or aborted to memory. They are safely subsumed by transactions , i.e. if a processor starts a transaction and then issues an atomic operation, that operation simply becomes part of the current transaction.
- **Seqlocks.** Sequence locks (seqlocks) are reader/writer locks that are an all-software analog to transactions. Seqlocks prioritize writers. Readers store a counter value at the start of a critical region, and writers increment the counter. A reader rereads the counter at the end of the critical region and if the value has changed, the reader re-executes its code. Regions protected by seqlock loops are protected by a transaction in TxLinux.
- **Read-copy-update.** Read-copy-update (RCU) [6] data structures avoid reader locks for a restricted class of data structures. RCU protects dynamically allocated data structures that are accessed by pointers. The implementation of RCU uses spinlocks, and these are converted to use transactions in TxLinux. TxLinux maintains the behavior that dynamically allocated memory used in RCU data structures is only freed when the kernel guarantees there are no more pointers to it.

There are significant barriers to converting Linux to use transactions [71], so the approach taken to converting Linux is incremental. Guided by profiling data, the most contended locks in the kernel were selected for transactionalization. The following subsystems were transactionalized: the slab memory allocator [12], the filesystem directory cache, filesystem translation of path names (which used a seqlock), the RCU internal spinlock, mapping addresses to pages data structures, memory mapping sections into address spaces, IP routing, and socket locking and portions of the zone allocator.

4.2 Evaluation

Linux and TxLinux versions 2.6.16.1 were run on the Simics machine simulator version 3.0.17. For these experiments, Simics models an 8-processor SMP machine using the x86 architecture. For simplicity an IPC of 1 instruction per cycle was chosen. The memory hierarchy has two levels of cache per processor, with split L1 instruction and data caches and a unified L2 cache. The caches contain both transactional and non-transactional data. Level 1 caches are each 16 KB with 4-way associativity, 64-byte cache lines, 1-cycle cache hit and a 16-cycle cache miss penalty. The L2 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 main memory is a single shared 1GB. For this study the conflict detection granularity in MetaTM is configured to be at the byte level, which is somewhat idealized, but Linux has optimized its memory layout to avoid false sharing on SMPs.

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

This evaluation of TxLinux is based on execution-based simulation. Execution based simulation, though resource intensive, is important because small changes

Name	**Description**
counter	A high contention shared counter micro-benchmark. The benchmark consists of 8 kernel threads (one per CPU) incrementing a single shared counter in a tight loop with no think time (like [5] and unlike [42, 56, 66]). Each thread performs a fixed number of increments, with synchronization enforced with spinlocks in Linux, and transactions in TxLinux.
pmake	Runs make -j 8 to compile the smallest 8 source files in the libFLAC 1.1.2 source tree and link them.
netcat	Send a stream of data over TCP. One instance per CPU.
MAB	Evaluates file system performance by simulating a software development workload. Runs 16 instances of the first four phases of the Modified Andrew Benchmark (no compile phase) in parallel.
configure	Run 8 parallel instances of the configure script for teTeX, one for each processor.
find	Run 8 instances of the "find" command to print the contents of a 78MB directory consisting of 29 directories with 968 files. The file contents are searched for a text string that is not found.

Table 4.1: Benchmarks used to evaluate TxLinux on 8 CPUs.

to thread event timing create larger-scale changes in workloads [3]. For example, the added latency from a transaction backoff can affect the order in which threads are scheduled. Execution-based simulation allows the timing of events to feed back into execution, where trace-based studies (such as [5, 17, 32]) simply count the number of events in the thread schedule that occurred when the trace was taken.

	counter	pmake	netcat	MAB	config	find
Linux	11.68s	0.66s	11.20s	2.48s	5.04s	0.93s
TxLinux	6.42s	0.67s	11.12s	2.47s	5.09s	0.93s
U/S/I Pct.	0/91/9	27/13/60	1/54/45	22/57/21	36/43/21	43/50/7

Table 4.2: Linux v. TxLinux system time (in seconds). Also shown is the division of total benchmark time into percentage of user/system/idle time.

		counter	pmake	netcat	MAB	configure	find
Linux	L1	5.90 %	4.78 %	18.09 %	12.85 %	9.68 %	21.09 %
	L2	40.64 %	0.42 %	2.49 %	1.07 %	1.03 %	4.20 %
TxLinux	L1	0.17 %	4.80 %	18.10 %	12.82 %	9.68 %	21.01 %
	L2	0.47 %	0.42 %	2.47 %	1.03 %	1.02 %	4.15 %

Table 4.3: Linux v. TxLinux cache miss rates.

4.2.1 Workloads and microbenchmarks

TxLinux was evaluated on a number of application benchmarks. The complete suite of benchmarks is listed in Table 4.1. The counter microbenchmark is different than the rest, in that the transactions it creates are defined by the micro-benchmark. The rest of the benchmarks are non-transactional user programs that run atop the Linux and TxLinux kernels. Thus, the transactions that they create are those due to TxLinux. This difference should be kept in mind as the characteristics of transactions are presented below.

4.2.2 TxLinux performance

Tables 4.2 and 4.3 show execution time and cache miss rates across all benchmarks for unmodified Linux and TxLinux. The execution times reported are only the system CPU time because only the kernel has been converted to use transactions. The user code is identical in the Linux and TxLinux experiments. To give an indication of the overall benchmark execution time, Table 4.2 also shows the breakdown of total benchmark time into user, system, and idle time, for the Linux kernel. In both Linux and TxLinux, the benchmarks touch roughly the same amount of data with the same locality; data cache miss rates do not change appreciably. The performance of the two systems is comparable, with the exception of the counter micro-benchmark, which sees a notable performance gain because the elimination of the lock variable saves over half of the bus traffic for each iteration of the loop.

	counter	pmake	netcat	MAB	configure	find
Total Transactions	12,003,505	382,657	339,265	2,166,631	3,021,123	225,832
Transaction Rate (Tx/Sec)	1,371,359	32,486	16,635	449,322	182,072	121,808
Transaction Restarts	3,357,578	10,336	10,970	36,698	65,742	25,774
Transaction Restart Pct.	21.9 %	2.6 %	3.1 %	1.7 %	2.1 %	10.2 %
Unique Tx Restarts	1,594	3,134	3,414	11,856	23,229	5,211
Unique Tx Restart Pct.	0.01 %	0.81 %	1.00 %	0.54 %	0.76 %	2.30 %
Pct. Tx In Interrupts, etc.	99 %	60 %	16 %	46 %	60 %	11 %
Pct. Tx In System Calls	1 %	40 %	84 %	54 %	40 %	89 %
Live stack overwrites	8,755	49	4	273	523	4
Interrupted Transactions	56,793	104	33	1,175	1,057	39

Table 4.4: TxLinux transaction statistics. Total transactions, transactions created per second, and restart measurements for 8 cpus with TxLinux.

Table 4.4 shows the basic characteristics of the transactions in TxLinux. The number of transactions created and the creation rate are notably higher than most reported in the literature. For instance, one recent study of the SPLASH-2 benchmarks [16] reported less than 1,000 transactions for every benchmark. The data shows that the rate of restarts is low, which is consonant with other published data [56]. Relatively low restart rates are to be expected for TxLinux because TxLinux is a conversion of Linux spinlocks, and significant effort has been directed to reducing the amount of data protected by any individual lock acquire.

The tables distinguish between two types of restarts: *unique* and *non-unique* restarts. *Non-unique* restarts count each unsuccessful attempt at completing a transaction. *Unique* restarts measure how many transactions restarted at least once. For example, if a thread starts a transaction which restarts 10 times before completing, it will count as 10 non-unique restarts, but as only 1 unique restart. With this distinction in mind, “Total Transactions” is defined to be unique transactions, which insulates it from the effects of contention management, and makes it closer to being a property of the program under test. We define “Transaction restarts” to be non-unique restarts. This is why “Tx Restarts” can exceed “Total Transactions”. Total non-unique transactions can be computed from the data presented. To calculate the restart rate we use: $\frac{TxRestarts}{TxRestarts+TotalTx}$.

The find benchmark shows the highest amount of contention, excepting the counter micro-benchmark. There are several dozen functions that create transactions, but approximately 80% of the transactions in find are started in two functions in the filesystem code (`find_get_page` and `do_lookup`). These transactions, however, have low contention, causing only 178 restarts. 88% of restarts are caused by two other functions (`get_page_from_freelist` and `free_pages_bulk`), which create only 5% of the transactions.

4.2.3 Stack memory and transactions

Also shown in Table 4.4 is the number of live stack overwrites. While the absolute number is low relative to the number of transactions, each instance represents a case where without MetaTM’s new architectural mechanisms, an interrupt handler would corrupt the stack of a kernel thread in a way that could compromise correctness.

The table also shows the number of interrupted transactions. The number is low because many of the spinlocks that TxLinux converts to transactions also disable interrupts. However, it is the longer transactions that are more likely to be interrupted and will lose more work from being restarted because of an interrupt.

Stack-based early release prevents 390 transaction conflicts in MAB, 14 in find and 4 in pmake. These numbers are small because TxLinux only uses **xpush** and **xpop** in interrupt handlers, and most transactions are short, without many intervening function calls. As transactions get longer and/or **xpush** and **xpop** are used in more general programming contexts, stack-based early release will become more important. The work required to release the stack cachelines is not large—MAB releases 48.2 million stack bytes while pmake releases 2.8, and both run for billions of cycles.

4.2.4 Transaction working sets

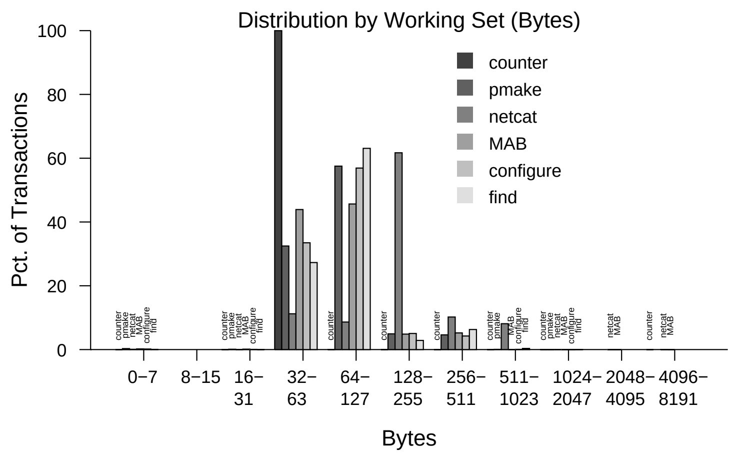


Figure 4.1: Transaction distribution by number of unique memory (byte) locations read or written.

Figures 4.1, 4.2, 4.3, and 4.4 present information about the size of transactions in TxLinux. Figures 4.1 and 4.3 show the distribution of transactions according to the number of unique byte-addressed locations they involve, while Figures 4.2 and 4.4 are in terms of unique cache line blocks involved, with block size fixed at 64 bytes.

Figures 4.1 and 4.2 show the total memory locations read or written by a transaction. If a location is both read and written, it is only counted once; if multiple reads or writes occur, they are only counted once as well. This is the traditional definition of working set, and indicates the net work of a transaction. With the exception of counter, where the working set size is tiny by design, the

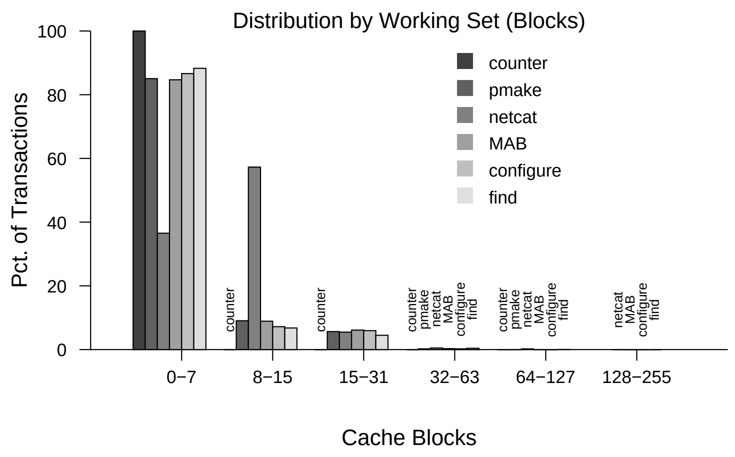


Figure 4.2: Transaction distribution by number of 64-byte cache blocks read or written.

benchmarks show that most, but not all, transactions in TxLinux are small. Most transactions touch fewer than 8 cache blocks. For the netcat benchmark, however, most transactions touch between 8 and 16 cache blocks, and 9% of transactions touch more than 16 cache blocks. One interesting statistic is the average number of memory addresses touched per cache block. For netcat, the average transaction touches about 128 bytes; about half are between 64-128, and half are between 128-256. The average number of blocks touched is 8. Since cache blocks are 64 byte in size, only one fourth of the data in cache blocks touched by the transaction is actually used.

Figures 4.3 and 4.4 focus on the memory written by a transaction, counting multiple writes to the same location only once. These measurements provide information not found in the traditional definition of working set. For HTM implementations, the amount of data written by a transaction plays a pivotal role in system performance. Specifically, data version management is entirely focused on the write-set of a transaction. For eager version management implementations, extra hardware is usually provided to buffer the old version of the data. In lazy version

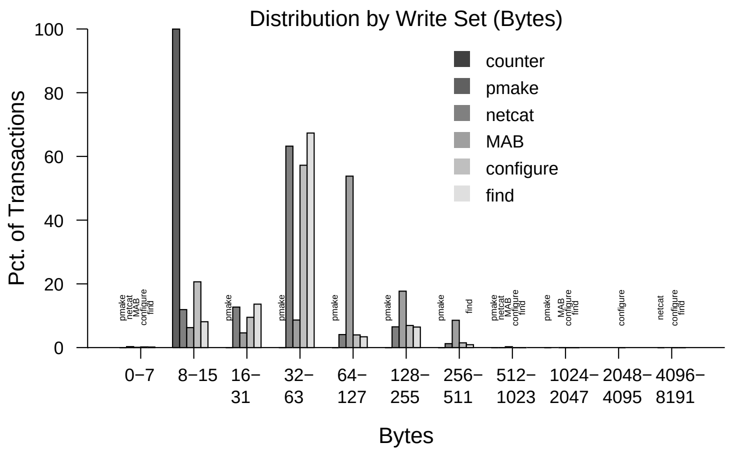


Figure 4.3: Transaction distribution by number of unique memory (byte) locations written.

management, the size of the write-set is what determines the amount of data that must be transferred at commit time.

LogTM [56], which uses eager version management, presents results of transactionalizing SPLASH-2, and showed that a 16-block write buffer would cover almost all transactions, except 5% of those in Barnes and 0.4% of Radiosity, whereas a 64-entry buffer would be sufficient to cover all large transactions. Across the benchmarks, TxLinux is similar to the largest of the Splash-2 benchmarks with a 16-block write buffer sufficient for all but 2.5%–3.5% of transactions. The largest transactions were in the 128-256 block range.

4.2.5 Commit and abort penalties

Figures 4.5 and 4.6 show normalized execution times for variable commit and abort penalties in TxLinux. Different HTM proposals create different penalties at restart and commit time, e.g., software handlers [53, 57] are functions that run when a transaction commits or restarts. In LogTM and MetaTM, the restart handler copies data from the log back to memory.

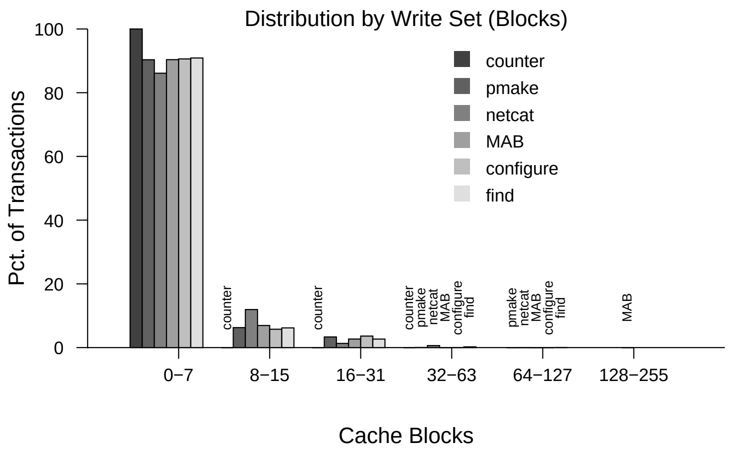


Figure 4.4: Transaction distribution by number of 64-byte cache blocks written.

The results for abort penalties reveal some subtle interplay between contention management and abort penalties: abort penalties can behave very similarly to explicit backoff, thereby reducing contention. As the abort penalty increases, performance does not necessarily decrease, as seen in netcat and find.

Commit penalties (Figure 4.5) have an obvious, negative impact on performance. While a moderate amount of work at commit time (i.e. 100 cycles) does not perceivably change system performance, counter and MAB slowed down by 20% at a commit penalty of 1,000 cycles, and all benchmarks significantly slowed down at 10,000 cycles. These effects will become more pronounced with more transactions.

4.2.6 Backoff Policy

Table 4.5 shows execution time across the benchmarks, with four different backoff policies. The data shows two of the policies are undesirable. The **random** policy performs poorly on pmake, MAB, and configure benchmarks, compared with the other policies, and the "none" policy (where a processor does not wait at all before restarting) performs poorly in the pmake and configure benchmarks. Linear and

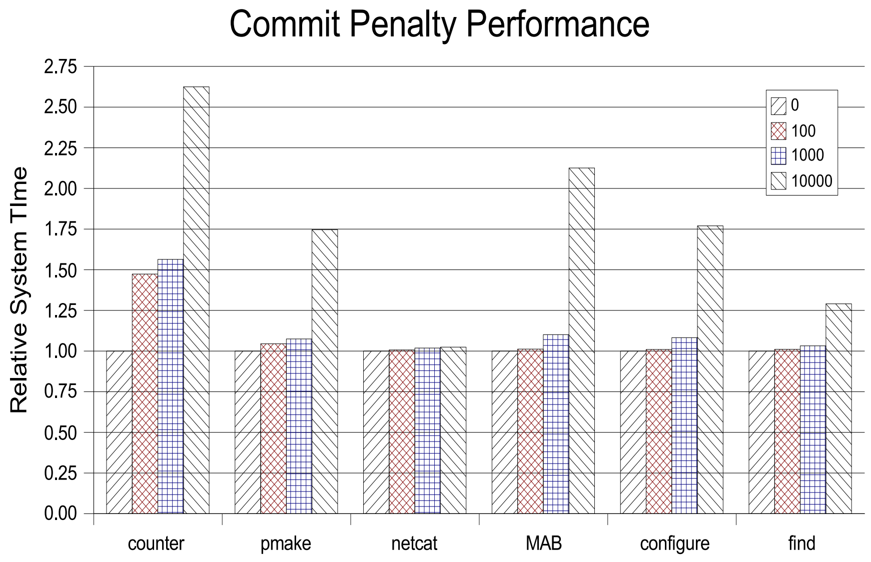


Figure 4.5: Relative system time for all benchmarks, varying the commit penalty among 0, 100, 1000 and 10,000 cycles.

	counter	pmake	netcat	MAB	configure	find
exponential	x	0.67 s	11.18 s	2.48 s	5.10 s	0.93 s
linear	6.42 s	0.67 s	11.12 s	2.47 s	5.09 s	0.93 s
none	6.24 s	0.80 s	11.34 s	2.47 s	8.56 s	47.28 s
random	6.36 s	0.84 s	11.20 s	2.46 s	11.05 s	x

Table 4.5: Backoff Policy effect on TxLinux system time (seconds). Cells with “x” represent data that was not available due to technical problems.

exponential backoff behave reasonably. The mean backoff cycles for the linear policy is between 75–1,189 cycles, depending on the benchmark. The means, however, do not show the whole picture, as most transactions that back off only stall for a very small number of cycles, while a much smaller set have much longer delays. This could explain why under low contention, there is no large difference between linear and exponential, since the difference is not pronounced when transactions restart only a few times.

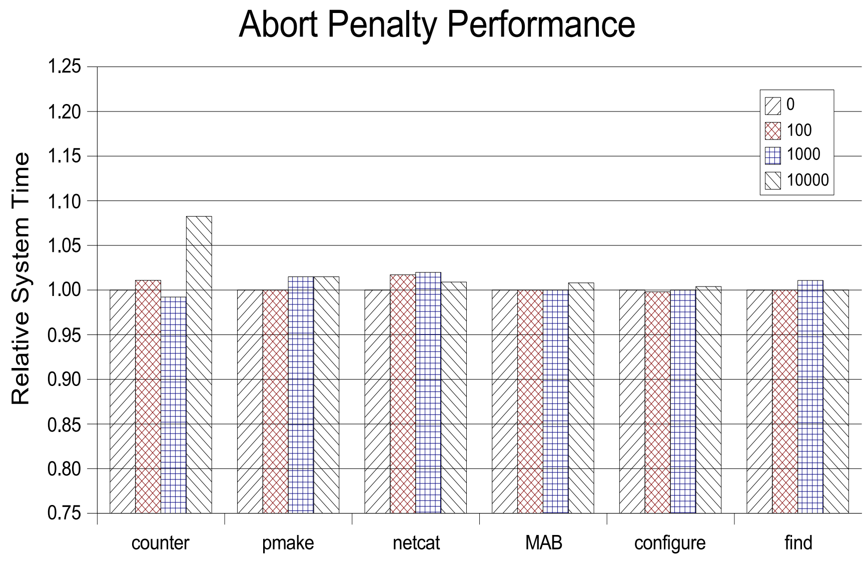


Figure 4.6: Relative system time for all benchmarks, varying the abort penalty among 0, 100, 1000 and 10,000 cycles.

Chapter 5

Dependence-Aware Transactions

Transactional memory provides the abstraction of atomic, isolated execution of critical regions. By *atomic*, it is meant that if a memory transaction fails for any reason its effects are discarded: either all of its updates become globally visible, or none of them do. By *isolated*, it is meant that no memory transaction sees the partial effects of any other transaction: uncommitted or speculative state is private to a transaction. Transactions are also *linearizable*: each transaction appears to take effect instantaneously at some point between when it starts and when it finishes [44].

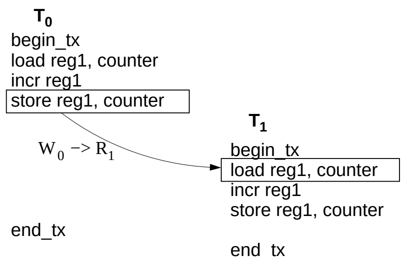
A transactional *conflict* occurs when one transaction writes data that is read or written by another transaction. When the ordering of all conflicting memory accesses is identical to a serial execution order of all transactions, the execution is called *conflict-serializable* [29].

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* [29]. Even TM implementations that do not use locks [33, 56], including both eager and lazy systems, only provide concurrency equivalent to two-phase locking. Any data read or written by one transaction has an implicit lock on it that conflicts with any attempt to write the same data.

This chapter proposes *dependence-awareness*, a transactional memory implementation technique that ensures conflict serializability. *Dependence-aware transactional memory* (DATM) manages conflicts by making transactions aware of dependences, and in some cases, by forwarding data values between uncommitted transactions. The key insight of DATM is that using conflict serializability as the system's safety property increases concurrency relative to using two-phase locking. Dependence-awareness allows two conflicting transactions that are conflict-serializable to both commit safely, thereby increasing concurrency and making 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.

Most previous proposals to help TM deal with write-shared data 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. 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. Because dependence-awareness admits concurrency where current designs cannot, it provides good system performance without burdening programmers with exotic new programming issues.

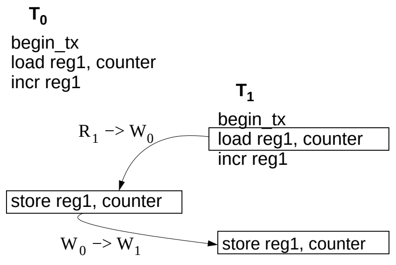
This chapter introduces the dependence-aware transactions model, first by way of an intuitive example in Section 5.1, followed by a more detailed description in Section 5.2. Section 5.3 provides an overview of the formal model and its properties.



```
begin_tx
load reg1, counter
incr reg1
store reg1, counter
end_tx
```

```
begin_tx
load reg1, counter
incr reg1
store reg1, counter
end  tx
```

(a) both transactions commit



```
begin_tx
load reg1, counter
incr reg1
store reg1, counter
```

```
begin_tx
load reg1, counter
incr reg1
store reg1, counter
```

(b) circular dependence

Figure 5.1: Two transactions increment the same counter, illustrating (a) a successful commit using dependences with data forwarding, and (b) an abort due to circular dependences.

5.1 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 work.

5.1.1 Shared counter example

This chapter adopts standard notation for data dependences, for example, W$\rightarrow$R means a memory cell was written by one transaction and then the same cell was 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 thesis a “memory cell” is a cache line unless otherwise stated.

Consider the shared counter shown in Figure 5.1(a). Assume that two differ-

```
 T0                   T1                  |  T0                   T1                  |  T0                   T1
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 5.2: 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.

ent 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 5.1(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 5.1(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[1]

5.1.2 Accepting more interleavings

Figure 5.2 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.

5.1.3 Comparison with other conflict resolution strategies

Figure 5.3 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 5.3 shows eager conflict detection (done at the time of the memory reference) [56] and lazy conflict detection (done at commit time) [33]. Eager conflict detection with restart (Figure 5.3b) causes $T2$ to restart on the conflict, and $T2$ conflicts again. Eager conflict detection with stall-on-conflict (Figure 5.3c) causes

[1]The choice of which transaction to restart, in addition to taking into account the usual contention manager factors such as transaction age and so on, can also take into account new factors such as how many other transactions have dependencies and would thus need to be restarted as well.

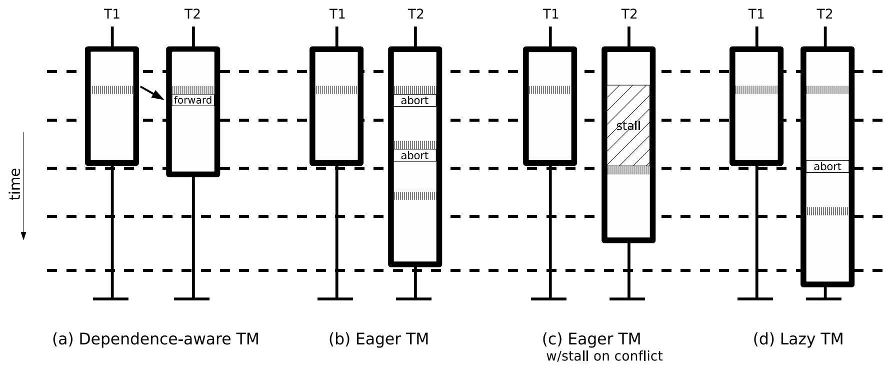


Figure 5.3: Two transactions that conflict while incrementing a shared counter. Part (a) shows the dependence-aware implementation, while Parts(b-d) show conventional HTM techniques. Assume that transaction $T1$ always commits first. The shared counter accesses are the shaded regions within each transaction.

$T2$ 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 strategies are committed safely by DATM.

5.2 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.

5.2.1 Dependence types

Table 5.1 shows a summary of dependence types and their properties. The notation W→R denotes a read after write (RAW) dependence—one transaction reads a memory cell 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	**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 restart if either: **a)** T_0 does. **b)** T_0 overwrites forwarded data with new value.

Table 5.1: Summary of dependence types and their properties.

dependence, then transaction A must commit before B.

The system tracks all dependences at the level of memory cells 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 memory cell when the dependence is created. The system records that the memory cell 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 memory cell for the source transaction. So the destination transaction must restart if the source transaction overwrites the data it forwarded. Table 5.1 lists the cases when restarts are necessary.

Dependences are created per memory cell on first access to the cell. Subsequent accesses to the same object do not affect dependence structure For example, if T_0 writes a memory cell that T_1 then writes, and then T_1 reads the memory cell 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.

5.2.2 Multiple dependences

Multiple dependences arise when two transactions conflict on more than one memory cell. Each memory cell 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 memory cell 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 memory cell, 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 memory cell. The latest writer provides the most up to date version of the memory cell. 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.

5.2.3 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.

There are several ways to handle cycles, or an implementation can avoid them. One 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, 78]. 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.

5.2.4 Exceptions and inconsistent data

Because W→R dependences forward uncommitted data, a transaction can read invalid or inconsistent data . Zombie transactions (those that will never commit) can enter infinite loops, write to incorrect addresses, read from incorrect addresses, jump to incorrect addresses, and fail program assertions. Some STM systems allow zombie transactions and have mechanisms to deal with them [20]. DATM's control over data forwarding makes containing zombies easy.

With dependence-aware transactions, infinite loops are resolved by runtime support. When transaction A enters an infinite loop (for example if a loop present in the original program fails to terminate due to inconsistent state relating to its termination conditions), it must have read inconsistent data from a transaction B that will not successfully commit. When transaction B restarts, the runtime restarts transaction A. If B is also in an infinite loop because of a dependence on A, the runtime system periodically polls for circular dependences and restarts both transactions. In a managed runtime (such as Java), the VM can propagate the restart. In C, the runtime system sends a signal.

The runtime buffers data written during a transaction, which prevents zombie transactions from corrupting the program's data structures, and from causing spurious exceptions due to stores to incorrect addresses. Zombie transactions can load from incorrect or invalid addresses, causing incorrect control flow or spurious exceptions. When any transaction that has read forwarded data throws an exception, the transaction is restarted in no-forward mode. Otherwise, it will be restarted when the source of the inconsistent data restarts. A managed runtime can detect exceptions directly, while an unmanaged environment can use signal handlers.

Jumping to a loaded address in a transaction that has read forwarded data causes the runtime to restart the transaction in no-forward mode. Program assertions must be integrated with the STM runtime. Any failed assertion in a transaction that has read forwarded data is restarted in no-forward mode.

5.2.5 Cascading aborts

Cascading aborts happen 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, cascading 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.

Cascading aborts have been unattractive in the database systems context due to the expense of rolling back transactions. Rolling back database transactions can be expensive, and the failure of a single transaction could trigger an arbitrary amount of rolling-back and restarting. For memory transactions, however, tracking dependencies and rolling back transactions is inherently cheaper (there are no disk accesses). DATM already tracks dependencies so it know exactly which dependencies to abort. Memory transactions are shorter than database transactions and are therefore less likely to have long dependence chains.

Most important, however, cascading aborts are not observed to be problem in DATM prototypes. There may well be applications vulnerable to cascading aborts, and dependence-aware transactions may not be effective for them. In this case they can be disabled by the system to improve performance.

5.3 Properties of model

The DATM model has been formalized and proven both safe and more concurrent than current mechanisms that enforce serializability. The complete proof is provided in PPOPP paper [73], this section will simply sketch the main approach and results.

The formal model is adapted from Lynch et al. [51], where the computation is modeled as a *history*, that is, a sequence of instantaneous *events*. Each event is either a read, write, commit or abort action.

One can think of a concurrency control mechanism as an automaton that accepts concurrent histories. First, we show that DATM is an automaton that accepts histories that are serializable. Specifically, we prove that if h is a history accepted by DATM then the set of committed transactions in the history have the following three properties:

Failure-Free None of the committed transactions fail.

Serial Steps of distinct transactions are not interleaved.

Legal Each value read from a variable is the value most recently written.

Second, we prove that DATM accepts *all* conflict serializable schedules. Contrast this with other TM systems that accept only a subset of the schedules because they use two-phase locking.

Some care is needed when interpreting this claim. In practice, an implementation will abort a transaction only if it detects, or suspects, a deadlock resulting from a cyclic dependency. Our automaton accepts all conflict-serializable histories, but an actual implementation may reject some as a result of imprecise deadlock detection (for example, premature timeouts).

Any history has an associated *serialization graph.* Each node is labeled with a committed transaction, and there is a directed edge from T_0 to T_1 if first T_0 and then T_1 apply conflicting operations (at least one writing) to the same object. A history is conflict-serializable if and only if the associated serialization graph is acyclic [29].

As a part of the proof we define two sets:

***earlier*(T):** The set of transactions that must commit before T can commit. This tracks the write-read dependence.

***notLater*(T):** The set of transactions that must commit or abort before T can commit. This tracks the read-write and write-write dependence.

DATM enforces the following commit rule—An active transaction T may commit only if all transactions in *earlier*(T) are committed, and all transactions in *notLater*(T) have committed or aborted. We show that this condition is both necessary and sufficient for ensuring that the serialization graph is acyclic. It follows that if a history is conflict serializable then the automaton will allow all transactions to commit.

This second claim is an important contribution of this paper. It quantifies the nature of concurrency that is available to TM systems to exploit. In essence

DATM is the first provable TM system that has the property of accepting all conflict serializable schedules.

Chapter 6

DSTM: Design and Evaluation

This chapter presents the design and evaluation of an STM system that uses dependence-awareness. Dependence-aware software transactional memory uses techniques from TL2 [21] that are modified to support dependences and data forwarding. This chapter focuses on the C language version (DATM-C), which is word-based. A C++ version (DASTM), which is object-based was also developed and is detailed elsewhere [69]. The implementations were evaluated on high contention workloads—a shared counter micro-benchmark, three programs from the STAMP benchmark suite [55], and STMBench7 [30]. The results show that some STAMP benchmarks benefit from managing dependences and forwarding data between uncommitted transactions.

6.1 Design

This section introduces the prototype implementation of dependence-aware software transactional memory (DASTM). It presents the key data structures and the basic steps transactions follow. Finally, it discusses some of the more interesting optimizations.

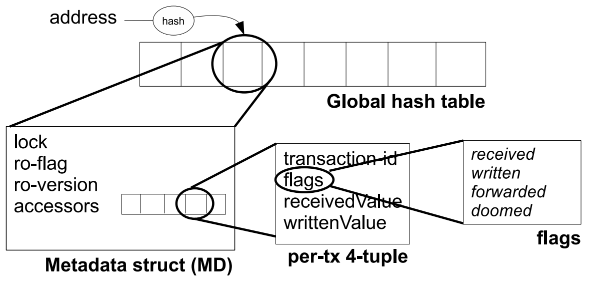


Figure 6.1: Key data structures in DASTM.

6.1.1 Data structures

Each thread maintains transaction-specific information in thread-local storage. Each transaction has a read-set and a write-set, implemented as linked lists with bloom filters to reduce list searches (like TL2 [21]). There is a single shared *global-clock* vector. Each transaction also has a *wait-vector* to manage dependences.

The primary shared data structure is a global hashtable that contains the system metadata, shown in Figure 6.1. Active transactions hash memory addresses to look up memory metadata structures (MDs) in the hashtable. Each address requires a unique MD, so hashtable collisions are resolved using a linked list of entries. DASTM uses the same addressing interface as the STAMP TL2 implementation, where load and store addresses are to 4-byte, aligned data units. Each MD contains the following fields.

- *lock*
- *ro-flag*
- *ro-version*
- *accessors*, a sequence of 4-tuples, each comprised of:

  [**transaction-id, flags, receivedValue, writtenValue**]

For efficiency, all addresses that hash to the same value share the same *lock* *ro-flag* and *ro-version*. The *lock* is a recursive spinlock that protects access to the MD structure. The *ro-flag* and the *ro-version* enable an optimization for memory locations that are only read during a transaction (see Section 6.1.3).

The core of the MD structure is the *accessors* list, an ordered sequence of 4-tuples. Each tuple has a transaction-id that identifies the transaction accessing this memory location. The flags field contains four bits, Received, Written, Forwarded, and Doomed. The first three bits indicate whether the tuple has received, written, or forwarded a value. The Doomed flag indicates that the transaction accessing the address will have to abort. The receivedValue field holds the memory value retrieved from memory or the forwarded value from another in-progress transaction. The writtenValue field records updates to the memory location made by the transaction.

6.1.2 Basic transaction execution

The following steps summarize transaction execution. Transactions end either in commit or abort (where aborted transactions restart).

1. **Transaction initialization.** Transactions begin by clearing the thread-local read and write sets. As described below, they obtain a transaction-id and initialize their *wait-vector* to all zeros.

2. **Transactional accesses.** Memory reads and writes add the address to the transaction's thread-local read or write set (respectively). They then look up and create, if necessary, the MD structure corresponding to the memory address in the global hashtable. The lock protecting the MD structure is held for the duration of servicing the memory operation. If this access is the first access to the address by the transaction, a new 4-tuple is appended to the accessors sequence in the MD.

   a **Reads.** If this is the first access to the memory location, the value is read either from an active transaction or from memory, and is then stored in the recievedValue field. Forwarding happens by having the transaction scan the accessors list backwards from the end for previous tuples. If it finds a tuple that does not have the Doomed flag set and has its Written flag set, the transaction copies the writtenValue, sets the Forwarded flag, and sets the Received flag in the receiving transaction's accessor tuple. If no such tuple exists, then the transaction initializes the receivedValue directly from the memory. The value returned for the read operation is the value in the receivedValue field, or, if the written flag is set, the value in the writtenValue field.

b **Writes.** The Written flag in the MD accessor flags field is turned on and the writtenValue field is updated with the new value being stored. If the memory address is previously read but not written (Written flag is not set), then it turns on the Doomed flag of all tuples that are later in the sequence.

3. **Transaction commit.**

a **Resolve dependences.** Wait until all dependences are resolved, i.e., all transactions this one depends on $(earlier(T) \cup notLater(T))$ must commit or abort (see Section 6.1.3 for details).

b **Write-set locking.** Acquire and hold the MD structure locks for all addresses in the write-set. If any write-set tuples for this transaction have the Doomed flag set, release all the locks and abort.

c **Read-set validation.** Validate the read-set by ensuring that none of the MD accessor tuples for this transaction have the Doomed flag set. The MD structure is locked only for the duration of the check. If the validation fails, the transaction releases all held locks and aborts. The read set does not need to be locked for the duration of commit because any subsequent writer will form a dependence and wait for this transaction to commit.

d **Write-back.** Write back the value of each element in the write-set to main memory, and release the MD lock. If the Forwarded flag of the transaction's tuple is set, the transaction dooms any dependent transaction that has received a stale value for this memory address. The transaction scans the dependents, dooming any entry that has its Received flag set if the receivedValue is different from the committing transaction's writtenValue. This check terminates at the end of the sequence, or at a non-doomed tuple that does not have the Received flag set.

The start of write-back is the transaction's linearization point [44]—any transaction that starts write-back will successfully commit, with any contending transaction serializing afterwards.

4. **Transaction abort.** A transaction that aborts must ensure that all transactions dependent on it also abort. For all addresses in the write-set with the Forwarded flag set, the transaction sets the Doomed flag for all subsequent accesses by transactions that have the Received flag set. The transaction stops at the first non-doomed tuple that has the Written flag set and the Received flag clear. Each MD structure is locked only for the time it takes to perform this check.

5. **Cleanup.** Both Commit and Abort complete by removing all tuples that correspond to the transaction's read and write set from the corresponding MD structures. The MD structures themselves (if dynamically allocated) may be freed if the tuple-sequence has become empty.

6.1.3 Design details

This section describes a few of the important optimizations and design choices made in the prototype.

Resolving dependences

A transaction must wait until all transactions on which it depends complete (commit or abort). After they commit, it can proceed (past step 3a) and continue its attempt to commit. A transaction's dependences are implicitly encoded by its tuple's position in the MD accessors sequence—the transactions of tuples preceding it in the sequence are the ones it may depend on. One strategy for resolving dependences is to iterate through each address in the working set, checking if all preceding transactions in the MD accessor list that have the Written flag set have completed. Recall that when a transaction completes, it removes its tuple from the MD accessor.

The prototype implements a more efficient strategy for resolving dependences that uses vector clocks. The runtime has a *global-clock* (GC) that tracks the number of transactions completed by each processor. Each transaction has a *wait-vector* that is used to summarize the transactions it has to wait on. When a transaction starts on a processor p, it reads GC[p] (the pth entry in the vector clock) and keeps track of V = GC[p]+1. This scalar (V) represents the value which the transaction will write into the *global-clock* when it completes, and is communicated to other processors that wish to take a dependence on this transaction. This communication occurs when a transaction accesses any memory address, it updates its *wait-vector* with the V values of all transactions preceding it in the accessor tuple. The V value is present in each tuple, as the transaction-id field encodes both p and V. When a transaction completes (whether commit or abort), it writes V to GC[p], after dooming its write-set. Dependence resolution is thus reduced to each transaction waiting for *global-clock* to be greater-or-equal to its *wait-vector*.

Optimizing read data

Most transactions read more data than they write. Some addresses are only read during a transaction and never written. For such transactions, the basic algorithm can impose a heavy performance penalty in the steps required to process a read (2a) and to validate the read-set at commit time (3c). For data that is only read during a transaction, we would like to avoid any MD structure locking , vector-clock management, tuple management, and so on.

The approach initially assumes that all data accessed by a transaction is read-only—as indicated by the *ro-flag* field in the MD. The transaction reads the desired data from main memory, and saves the *ro-version* value in its read-set. The validation phase (3c) for such memory locations consists of ensuring that for each address the *ro-version* has not changed in the MD structure, and that the *ro-flag* still indicates that the address is in read-only mode.

If a transaction stores to a memory location (i.e. uses the MD structure), the *ro-flag* is cleared. Any transaction that previously read the location while the *ro-flag* was set will abort during validation if it sees that the flag has been turned off. With the flag off, reads are processed as in 2a. The runtime can decide to transition an MD back to read-only mode by resetting the *ro-flag* and increasing the *ro-version*. Increasing the *ro-version* ensures that any outstanding transaction that reads the location in read-only mode will abort during validation. The runtime might turn on all *ro-flag*s if there are no active transactions, or might turn them on every N transactions.

Deadlock management

Deadlock can arise in the commit protocol, steps 3a-c, for a variety of reasons. First, cyclical dependences in the DASTM model result in two transactions waiting for each other to commit, and thus both stay in step 3a indefinitely. Second, no specific ordering on lock acquires is imposed (exacerbated by the fact that we acquire write-set locks before read-set), so transactions can deadlock in steps 3b or 3c. Third, since the implementation uses a single *lock* to protect multiple MD structures that hash to the same bucket, on rare occasions false conflicts can cause deadlocks.

Deadlocks are handled using timeout, similar to TL2. Other approaches are possible, including implementations that avoid deadlocks (e.g. by restricting dependence creation to guarantee acyclic dependences or imposing lock order), or which use more sophisticated deadlock detection techniques like Dreadlocks [39].

Design tradeoffs

The STM design does not implement every feature of the dependence-aware model. Transaction dependences are always created relative to the most recently written value of the object. With this policy, the dependence graph is always a chain, and new dependences are appended to the end. A given transaction will forward only a single value, even if the address is written multiple times. That single value can be forwarded to multiple transactions. Preliminary data indicated that these optimizations would generate little performance and add complexity.

6.2 Evaluation

Experiments for DASTM were conducted on a Sun server using the UltraSparc T1 (Niagara) processor. This processor contains eight multi-threaded cores with four contexts per core, for a total of 32 total hardware contexts. The machine runs the 64-bit Linux 2.6.24-19 operating system. Counter is used as a micro-benchmark to study how DASTM performs in the presence of hot-spots. Performance results are also reported for three representative STAMP 0.9.8 benchmarks—vacation, labyrinth and ssca2. DASTM is compared with unmodified TL2 [20] on each of these benchmarks. The TL2 statistics obtained differ from those reported by Minh

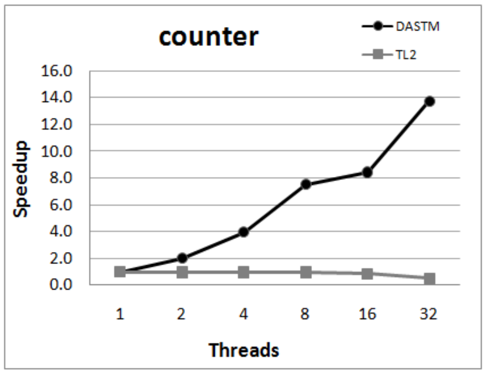


Figure 6.2: Speedup (higher is better) seen in DASTM and TL2 on the counter benchmark.

et al. [13] because their results are from a simulator, while those presented here are from real hardware. The TL2 code distributed with the STAMP suite was used. Each benchmark's threads are appropriately pinned to individual processors (using thread affinity) to avoid OS scheduling anomalies. Tthe average of three benchmark runs are reported.

6.2.1 counter

Writing shared data within a transaction generally leads to hot-spots that result in poor performance of an STM. Figure 6.2 shows the effect of updating a shared counter for a total of 100,000 times using a variable number of threads. Each increment transaction also contains a fixed amount of think time (5,000 iterations of a local loop), to simulate work on private data. TL2 does not scale at all, revealing the inherent lack of concurrency in two-phase locking systems. This micro-benchmark demonstrates how DASTM, in ideal conditions, can increase concurrency by allowing conflicting transactions to safely commit.

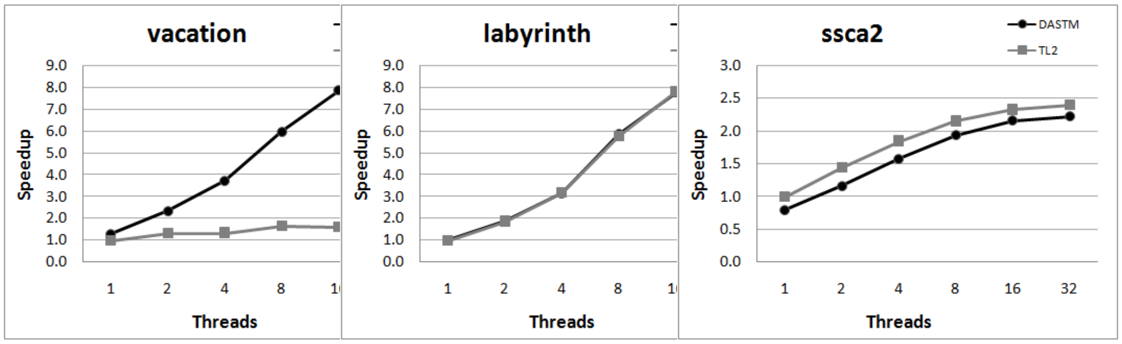


Figure 6.3: Speedup (higher is better) achieved by DASTM and TL2 compared to the single thread performance of TL2 on (A) vacation (B) labyrinth and (C) ssca2.

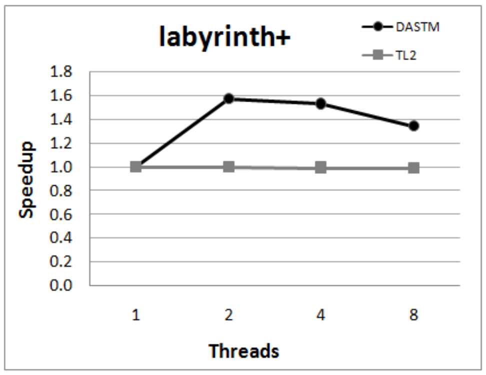


Figure 6.4: Speedup (higher is better) achieved by DASTM and TL2 compared to the single thread performance of TL2 on labyrinth+, a high contention variant of labyrinth

6.2.2 STAMP

vacation Vacation models a travel reservation system. It uses red-black trees to store data. Client tasks are performed within transactions to provide safe access to this data. The experiments execute 1,000 transactions and use the parameters "-t 1000 -n 100 -u 50". This particular configuration has very high contention and more than 86% of the benchmark time is spent within transactions. Figure 6.3 depicts how DASTM compares to TL2. DASTM outperforms TL2 by 4.86× at 16 threads.

The results show that vacation performance decreases on DASTM going from

16 to 32 threads. The abort rate doubles when going from 16 to 32 threads with almost all of vacation's aborts due to timeouts waiting for dependences to be satisfied at commit (see Table 6.1 in Section 6.2.3 below). Increasing the timeout value which decreased the abort rate and improved performance at 32 processor threads, nearly matching 16 thread performance. With more cores, deadlock detection that is more precise than simple timeouts (for example, Dreadlocks [39]) are likely to become important to sustain good performance.

labyrinth The labyrinth benchmark uses Lee's algorithm to find the shortest path between pairs of nodes in a maze [13]. The program reads and updates memory locations within data structures, such as a worklist and a grid. Most of the updates occur in long transactions. Figure 6.3 shows the results for a maze of size $256 \times 256 \times 5$, using parameters "-i random-x256-y256-z5-n256.txt". The total number of transactions is between 514 to 576 (depending on the number of threads). With the default transaction boundaries, both TL2 and DASTM are able to scale well on this benchmark. Therefore two different transaction boundaries were used: the benchmark's primary loop creates two transactions per iteration, which in a new version of the benchmark are merged into a single transaction. This is another way to transactionalize the benchmark (producing the same results), however contention is much higher. Figure 6.4 shows the results for this variant, called labyrinth+. DASTM is able to improve performance by up to approximately $1.6\times$ with additional cores, whereas TL2 is unable to improve beyond single thread performance. Neither system achieves any additional speedup as the number of hardware threads is increased above 2. Like vacation, aborts due to time out increase with more threads. In addition, overwrite aborts (shown as A_2 in Table 6.1) also increase with more threads. Experiments with increased timeout thresholds do reduce those aborts by up to 60%, but overwrite aborts remain unchanged and thus become the limiting factor for performance. Even with more sophisticated deadlock detection, labyrinth+ is

Parameter (in %)	**vacation**	**labyrinth+**	**counter**
Reduction in Exec-time	72.5	23.9	86.8
Reduction in Restarts	98.2	90.0	99.5
Abort Rate	44.3	88.8	3.3
$\mathbf{A}_1$**:**Dep. Wait Aborts	79.5	62.6	20.7
$\mathbf{A}_2$**:**Overwrite Aborts	16.2	34.0	79.3
$\mathbf{A}_3$**:**Lock Timeout Aborts	4.2	0.0	0.0
$\mathbf{D}_1$**:**Tx using R$\rightarrow$W	3.6	3.8	0.0
$\mathbf{D}_2$**:**Tx using W$\rightarrow$W	1.3	76.6	99.6
$\mathbf{D}_3$**:**Tx using W$\rightarrow$R	34.0	77.4	99.6

Table 6.1: DASTM statistics for vacation, labyrinth+ and counter at 8 threads. The first two rows are relative to the non-DASTM benchmarks.

inherently limited in the amount of concurrency that can automatically be achieved.

ssca2 The ssca2 benchmark uses a scientific computational kernel that operates on a multi-graph to produce an efficient graph structure representation using adjacency (and other auxiliary) arrays [13]. The benchmark is run using the parameters "-s17 -i1.0 -u1.0 -l3 -p3". It creates a large number of transactions: over 1.4 million, which individually are relatively small. The benchmark has very little contention, so it does not benefit much from dependence management. Figure 6.3 shows that on single-threaded runs, TL2 is roughly 20% faster than DASTM, due to overheads for managing metadata. Overheads for other benchmarks depend on a variety of factors including access to DASTM metadata, transaction contention, and the ratio of transaction computation to data accesses. TL2 achieves a peak speedup of approximately 2.4× at 32 threads, while DASTM's overhead causes its peak speedup to be slightly lower at 2.25×.

6.2.3 DASTM statistics

Table 6.1 shows the reduction in execution time and in transactional restarts moving from TL2 to DASTM at 8 threads for the three highest contention benchmarks—vacation, labyrinth+ and counter. On all of them, the number of dynamic aborts is reduced by 90% or more. These findings validate the intuition that current STMs abort more transactions than what is strictly necessary to remain safe. DASTM's reduction in aborts translates to increased performance.

Table 6.1 also gives the percentage of transactions that restart (abort rate) and a breakdown of the various types of aborts and dependences seen in these benchmarks. The *Abort rate* of the vacation benchmark shows that 44.3% of the total transaction attempts resulted in aborts. These aborts occur for three reasons: (1) timeouts while waiting for dependences to be satisfied (A_1), (2) aborts because a transaction overwrote a value that it had already forwarded (A_2), and (3) timeouts while trying to acquire locks on memory locations (A_3). The values of categories A_1–A_3 equals 100% of aborts (nearly 100% for labyrinth+, which has some explicit calls to abort). In a workload such as the counter benchmark aborts are mostly due to forwarding of values that are then overwritten, while labyrinth+ and vacation mostly abort due to timeout while waiting for dependences to resolve.

D_1, D_2 and D_3 give the number of transactions involved in R→W, W→W and R→W dependences. For example, 99.6% of the transactions in counter read values forwarded by the (W→R) dependences. The numbers in these categories do not total to 100% because a single transaction can have multiple dependence types.

Chapter 7

Coordinated Sibling Transactions

The traditional prescription for intra-transaction parallelism is a nested subtransaction model, extended to allow the nested transactions to run in parallel. This basic model works only if the parallel activity within the transaction is independent. However, the fact that the work was put into the same transaction to start with makes complete independence unlikely. Moreover, some kinds of parallelism, such as speculation, are not exploitable in this model. While this intra-transaction model is conceptually simple, it has not been widely used in a database context, or examined in much depth in the newer TM context.

This chapter proposes a new mechanism, *xfork*, a programming construct designed to make it easier for programmers to express intra-transaction concurrency. It enables programmers to leverage additional cores to increase performance, while retaining the ease of use of the TM API. Xfork is the mechanism by which programmers create **coordinated sibling transactions**. These transactions are similar to parallel closed-nested transactions, but with added coordination semantics that make them more accessible and useful to programmers [74].

Coordinated sibling transactions allow programmers to treat parallel nested transactions as a group and to specify the semantics for the group as a whole. This relationship is formalized in this model using *sibling coordination forms* OR, AND, and XOR, using a rough analogy to the Boolean functions. The utility of these forms for TM programmers, and the natural fit with TM code structure, is demonstrated.

The xfork API can be viewed as a blend of nested transactions, fork/join parallelism, distributed transaction coordination and speculation. To allow programmers to focus on their application's algorithms, and parallelism within them, the runtime should shield them from the complexities of the low-level implementation of such coordination. The xfork implementation handles the threading and concurrent execution of the different forks, and performs the necessary coordination. Programs built with the xfork API benefit because they can use extra cores to speed up individual transactional units of work.

This chapter begins with an example (Section 7.1) that motivates the xfork API and the coordinated sibling transaction model (Section 7.2).

7.1 Motivating Example

This section illustrates coordinated sibling transactions with an example taken from one of the benchmarks in our evaluation.

7.1.1 Background

Our example is the prototypical bank transfer scenario, where a sum of money is transferred between two bank accounts. The bank account records are represented by nodes in linked lists that need to be protected from concurrent accesses.

We'll assume that the two bank accounts are stored in separate lists (one for checking accounts and one for savings accounts). If each list is protected by a lock, and the transfer operation does not use fixed lock ordering, then concurrent transfer operations can result in deadlock. Deadlock occurs when each of two transfers acquires one data structure's lock and waits for the lock held by the other operation.

Transactions simplify the programmer's task because he or she need not consider the order in which data structures are accessed. Thus, the potential for programmer-visible deadlock is eliminated, as is the complexity involved in avoiding deadlocks. In the example of two data structures and two locks, lock ordering is trivial. However, when scaled to real programs, lock ordering becomes difficult, sometimes even impossible, to define and maintain.

7.1.2 Data structures

Our basic example starts with two data structures. The first, `checking-list`, holds checking account records, with one record per node. The second, `savings-list`, holds savings account records. To make the example even more flexible (and more realistic), we add a third structure that is also accessed by the transfer operation. We assume that if the amount transferred into a savings account exceeds a certain threshold, then the bank wants to make a note of the transaction. For our example, the bank may want to send the customer involved a special offer to upgrade his savings account to silver or gold status. Since the accounts involved in this list are called Notable Savers Accounts, we call this third structure the `nsa-list`. All three structures are doubly-linked lists.

7.1.3 Transactional transfer code

The pseudo-code for the basic transactional transfer operation as it might be implemented on STMs today is shown in Figure 7.1. We make a few observations:

- The account numbers are specified in the `src-accno` and `dst-accno` parameters. For clarity, we assume both accounts always exist, and omit the code to deal with missing accounts.

```
Transfer (src-list, src-accno, amount
          dst-list, dst-accno, nsa-list)
{
  atomic
  {
     DebitCredit(<params>)
     NsaCheck(<params>)
  }
}

DebitCredit (<params>)
{
  src-account = SearchList(src-list, src-accno)
  if (src-account.balance >= amount)
  {
    src-account.balance -= amount
    dst-account = SearchList(dst-list, dst-accno)
    dst-account.balance += amount
  }
}

NsaCheck (dst-list, dst-accno, nsa-list, amount)
{
  if (dst-list == savings-list && amount > 10000)
    AddNotableList (nsa-list, dst-acct, amount)
}
```

Figure 7.1: Traditional transactional code for the transfer operation

- The destination account will be credited if and only if the source account debit succeeds (i.e. there are sufficient funds).
- We want to add the accounts to the `nsa-list` even if funds were insufficient to complete the transfer because, for the bank's purposes, attempted transfers are just as important to track as actual transfers.

We omit the pseudo-code for SearchList, which is a simple linked list traversal. The AddNotableList code searches the list to see if the record for the specified account exists and, if so, notes the latest transfer amount in that record. If the account isn't already notable, it adds a new record for the account to the list. The entire transfer operation is performed within a single transaction. The code executes safely even in the presence of other concurrent transfer operations. Without transactions, the code would be more complex, due to locking issues and recovering from partial failures. The code is simple and correct, but for programmers wanting to take advantage of multi-core processing, the basic TM API provides no further easy routes for improving the latency of the transaction. The next subsection shows how sibling transactions can improve this code's performance with very little coding effort.

7.1.4 How to parallelize the transfer operation

Even though our transfer operation is trivial in size, concurrency can be improved with the appropriate tools. We can identify three potential areas for concurrency, each of which corresponds to one of the forms of sibling transactions. The forms are described in more detail in Section 7.2, but intuitively these determine under which conditions sibling transactions are allowed to commit.

- The actual debit/credit is independent of the `nsa-list`. Regardless of whether the debit/credit succeeds, we will be performing the `nsa-list` check. Thus, the `nsa-list` work can be performed in parallel. This is an example of OR-form sibling transactions. (The Boolean notation describes the relation between the success of the forks, and the success of the xfork operation). Note that there is no short-circuit evaluation in OR-form, each of the forks are executed.

- The actual transfer consists of two parts, debit and credit. They can be performed concurrently, since the credit part does not depend on data from the debit. The only restriction is that the credit part must not be allowed to complete if the debit fails (i.e. insufficient funds). This is an example of AND-form sibling transactions. There is short-circuit evaluation in this form.

- The SearchList function, which operates directly with the linked lists, may also be sped up by performing the search in multiple ways. Because we are dealing with doubly-linked lists, the search may be performed simultaneously from both ends of the list (i.e. both forward and backward traversal). This kind of parallelism is data structure and operation specific. This parallelism is an example of XOR-form sibling transactions. (Even if both succeed only one should commit).

7.1.5 Parallel transactional transfer code

Taking into account the potential concurrency identified in the previous section, we can re-implement the transfer operation to take advantage of sibling transactions. The new code is shown in Figure 7.2.

The pseudo-code uses xfork with four parameters: the first specifies the form of coordination between the sibling transactions (AND, OR XOR), the second specifies how many sibling transactions are to be created, the third specifies the procedures to be invoked within the nested transactions, and the fourth defines the

```
Transfer (<params>)
{
 atomic
 {
   xfork (OR, 2, {DebitCredit, NsaCheck},
                 {<params>, <params>}
         )
 }
}

DebitCredit (<params>)
{
   xfork (AND, 2, {AcctAccess, AcctAccess},
            {(src-list, src-accno, -1*amount),
             (dst-list, dst-accno,amount)}
         )
}

AcctAccess (list, accno, amount)
{
    account = SearchList(list, accno)
    account.balance += amount
    if (account.balance < 0)
       return Failure;
    else
       return Success
}

SearchList (list, accno)
{
    var account
    xfork (XOR, 2, {FwdSearch, BackSearch},
             {(list, accno, &account),
              (list, accno, &account)}
          )
   return account
}
```

Figure 7.2: Parallel transactional code for the transfer operation, using sibling transactions

parameters to pass to each procedure. The next section describes the API in more detail.

The changes are straightforward and localized. The transfer operation itself now simply uses an OR-form xfork to execute both the transfer operation and the NsaCheck concurrently. The SearchList procedure is rewritten to use an XOR-form xfork to search a list concurrently using forward and backward traversal (we omit the FwdSearch and BackSearch traversal code). Finally, the DebitCredit procedure now uses an AND-form xfork operation to perform both parts of the operation concurrently. A new procedure, AcctAccess, is invoked to perform both the debit and credit operations. It returns a failure code (which aborts the operation) if the balance goes below zero (i.e. insufficient funds). The AcctAccess procedure invokes SearchList, which also uses xfork. The AddNotableList function (called by NsaCheck), is unchanged, though it benefits from the modifications to SearchList.

7.1.6 Observations about the modified code

Our modified example code highlights several important aspects of sibling transactions:

- Sibling transactions are composable. A transaction can create siblings with xfork and these sibling transactions can create nested siblings themselves. Figure 7.3 illustrates the sibling transaction tree produced by the execution of the transfer operation. The six leaf nodes represent the maximum number of concurrent threads that can be involved in executing the single Transfer operation.

- Sibling transactions retain the safety property of transactions. When identifying potential areas for concurrency, we focus on the parts of the original algorithm that are independent, and thus can safely execute concurrently.

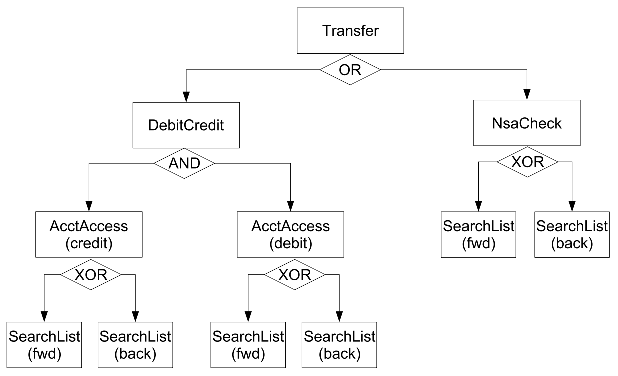


Figure 7.3: Sibling transactions created by transfer example. The diamonds represent xfork invocations, of the specified types.

However, since the transactions execute as nested subtransactions the programmer does not need to guarantee that they are completely independent of each other. If they touch common data, the fact that they are sub-transactions provides the atomicity property that protects them from concurrent accesses by other siblings (or, for that matter, from other top-level transactions).

- Of the three forms of xfork in this example, only the XOR form involves speculation. This form of speculation is novel because it uses the atomicity and rollback properties of transactions to allow the programmer to express speculative execution paths that cannot be automatically inferred.

7.2 Coordinated Sibling Transactions: Model

This section presents the xfork API and the semantics of coordinated sibling transactions.

```
bool xfork (xforkForm form,
                   int numForks,
                   xforkProcedure forkProc,
                   object data);

enum xforkForm { AND, OR, XOR };

delegate xforkResult xforkProcedure (int forkNum,
                            object data);

enum xforkResult { Success, Failure };
```

Figure 7.4: The xfork API and accompanying types. The user implements the actual sibling code in a function with the signature of xforkProcedure.

7.3 Public API

We add a single function, *xfork*, to the TM API which provides coordinated sibling transactions. The function name stands for transactional fork, and builds on intuitive notions of the fork/join pattern for expressing parallelism (and to a lesser extent the fork system call). The method signature from our C# implementation is shown in Figure 7.4. Other languages, or overloads, can provide additional syntactic sugar for the API (e.g. separate forkProcs for each fork), but the core parameters for xfork are as follows:

- *form*: the form of sibling coordination (AND, OR, XOR).
- *numForks*: the number of concurrent sibling transactions to create.
- *forkProc*: a procedure to execute inside the sibling transactions.
- *data*: (optional) user-specified data to be passed to each forkProc.
- *return-code*: the xfork function returns true on success, false otherwise.

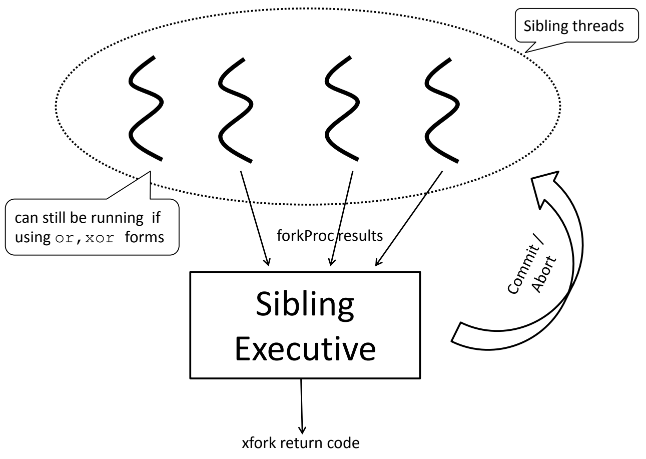


Figure 7.5: Illustration of how forkProc return codes are used.

The system will create numForks nested transactions, with implementations free to schedule them as concurrently as possible. The system starts the sibling transactions before it calls into the user-provided forkProc, passing in the optional data. This data usually contains work-partitioning information, space for output results, and so on. Fork procedures (forkProcs) are passed the forkNum parameter, to identify which sibling fork is being executed on this thread.

The forkProc returns a code indicating success or failure, which is used by the system to determine whether that transaction (and possibly its siblings) will be allowed to commit or forced to abort. Note that forkProcs do not themselves commit or abort the sibling transactions they are executing in—the commit decision is handled by the system (this process is illustrated in Figure 7.5). Sibling transactions may request a restart, as any regular transaction can, and they may also restart due to conflicts with other transactions. Re-execution after restarts will cause the forkProc to be invoked again by the system.

7.4 Sibling forms

The form parameter specifies the coordination semantics between sibling transactions. We focus on the three core forms used by our benchmarks (though our system allows custom user-defined forms). The semantics of each form is a function of the return codes of the forkProcs:

- AND: All sibling transactions must succeed, or none succeed.
- XOR: Only one sibling transaction must succeed.
- OR: Sibling transactions succeed or fail independently.

The coordination happens in a (conceptual) pre-commit phase for the sibling group. The xfork operation decides the outcome for each sibling transaction based on the form semantics. If the semantics are satisfied, then the siblings commit as allowed by their form type (e.g. in the AND form all siblings commit, in the XOR form only one could commit, and in the OR form all successful siblings commit).

All sibling transactions need not complete execution before a decision to commit is made. An XOR-form call completes as soon as a single sibling return success and finishes commit. The system need not wait for other siblings to complete, and ensures that they will abort (if it has scheduled them). In an AND-form call, if a single sibling returns a failure code, the parent fails.

These forms emulate a wide range of coordination behaviors, ranging from the traditional distributed-transactions semantics of the AND form, to the traditional independent closed-nested transactions of the OR form, to the speculative nature of the XOR form. However, the XOR form can also be used in a non-speculative manner, such as when a work item is known to be in one of a set of data structures (or buckets). In that case, parts of data structures can be searched in parallel, but as soon as one transaction finds the target, the XOR semantics terminate the other siblings.

7.5 xfork return code

The xfork API returns true if siblings succeed according to the specified form's semantics, and false if they do not. For example, the XOR form will return false if none of the forks return success, while the AND form will return false if any fork returns failure. When xfork returns false, it guarantees to the caller that *no siblings will commit* (i.e. system state is the same as before the xfork call).

If the xfork operation completes successfully, it returns true. The read and write sets of the successful siblings are merged into the parent transaction following the usual rules of closed-nesting. For the OR form, an overload of the API can return status flag which indicates which siblings (forks) completed successfully, and which failed.

Edge cases exist where the semantics associated with a true or false return code cannot be provided. The implementation detects these cases before returning a result to the user, and handles them by forcing the parent transaction to restart. This case arises due to the inability to *guarantee* an atomic commit of a set of siblings, even after xfork successfully completed the prepare phase of the entire set of transactions. The final decision rests with the CLR's transaction manager (discussed below), and possibly other resource managers. By relying on higher-level failure-handling (aborting the parent transaction), we guarantee that the transaction is always executing in a well-defined program state. This is one of the key reasons why xfork can only be called from within a transaction.

7.6 Sibling conflicts

Programmers using the AND form should ensure that the sibling transactions have independent write sets (xfork returns an error and aborts all siblings if this is violated). They may conflict with non-sibling transactions, but it is a programmer error if they conflict with each other. It may be possible to run such conflicting siblings serially, but until we have more experience with AND-form siblings, it is unclear whether this situation is one that should be supported. Conflicts among siblings in the OR or XOR form are allowed, and will cause one of the siblings to restart. If the conflict occurs with a sibling that has executed successfully (but hasn't committed, since the entire xfork is still active by definition), then such a conflict will cause the non-completed sibling to behave as if it has returned a failure error code. The thread stops attempting to re-execute.

Chapter 8

Sibling STM: Design and Evaluation

This chapter examines issues related to building a TM system on the Common Language Runtime, and presents the architecture of SSTM, one such system (Section 8.1). This is followed by the design of a software system that supports the coordinated sibling model (Section 8.2) and an STM implementation that supports parallel closed-nesting (Section 8.3). Lastly, it evaluates the ability of the prototype to scale the performance of several benchmarks (Section 8.4).

8.1 SSTM: Architecture

This section describes the architecture of *Sibling STM* (SSTM). SSTM is our prototype implementation of an STM that supports the xfork API and coordinated sibling transactions. It is built on top of the .NET Common Language Runtime. This section provides an overview of how we use and extend the CLR and integrate with its transactional libraries. This section also provides an overview of the two primary components of SSTM: the coordinated sibling executive (SibEx) and the nesting-aware transactional object store (TxStore). The two components are mostly independent[1] and the detailed design of each will be presented in Section 8.2 (SibEx) and Section 8.3 (TxStore).

8.1.1 Common Language Runtime

The Common Language Runtime (CLR) is Microsoft's managed execution environment that underlies several commercial and research languages. The CLR has several features that make it an attractive environment for our prototype work. These include significant transactional and threading infrastructures, located in the System.Transactions and System.Threading class libraries. Our prototype can be used by any CLR language (as long as we stay within the bounds of the CLI specification), including C# and VB.Net.

Our design is somewhat constrained because we do not have access to the CLR source code and we did not want to make changes to the C# language. Thus, we built on top of the CLR, exploiting as many extensibility points as possible while working within the C# language. The ability to change the language or the internals of the execution engine would result in even tighter integration of sibling transactions.

8.1.2 CLR transaction model terminology

SSTM works within the database-derived model defined by the System.Transactions library, a model different from that used by most STM implementations. Therefore, some of the terminology may be unfamiliar in the TM context. Usually an STM system provides a programming interface (such as the atomic keyword) to use transactions, and a runtime implementation that manages the transactions and memory.

[1]The single minor exception will be noted in Section 8.2.4.

Our database-derived model separates several of the system's components. The *transaction manager* (TM) is responsible for tracking active transactions, qua transactions, and coordinating their commit/abort. The TM is not directly involved in the execution of operations inside any given transaction. One or more *resource managers* (RM) manage the actual resources that may be transactionally modified, as well as export the operations that make these modifications. The TM and RM must be coordinated. This coordination is handled through the *two-phase commit* (2PC) protocol, where RMs vote on whether a transaction should commit, and are informed of the outcome of the transaction as determined by the TM. The first time that a transaction uses an RM, the RM must *enlist* in the transaction, meaning that the RM contacts the TM to participate in the 2PC protocol.

SSTM uses this model, with the actual transactional memory being represented by the TxStore RM. Moreover, the coordination between sibling transactions (performed by the SibEx component) does not require modification of the TM, but can be implemented as just another RM involved in the transaction. Unlike existing STMs, which usually provide their own set of TM APIs and do not support additional resource managers, SSTM must cope with the loss of absolute control over, for example, the process involved in starting and completing transactions.

8.1.3 System.Transactions interfaces

In the CLR, the TM is provided by the System.Transactions library, while the user must implement the RMs. SSTM's two main components (TxStore and SibEx) are resource managers, and thus must interact with the main RM interface, an interface responsible for enlistments. An enlistment is made using the current transaction object's EnlistVolatile method (Transaction.Current is a thread-local variable that holds the current transaction object, if any). An enlisting RM must implement an interface that is used as part of transaction completion. The IEnlistmentNotification interface (defined in System.Transactions) is shown in Figure 8.1. The interface reflects the familiar two-phase protocol commonly used by distributed transaction managers[2].

System.Transactions also includes the interface that is used by the programmer to create transactions. Transactions are created through the TransactionScope mechanism. This mechanism is similar to the familiar atomic block with a few exceptions, notably that transactions do not automatically retry when they abort. Transaction commit or abort is ultimately determined by the TM and all the RMs together, not by any individual RM. This framework has the benefit of being more general and extensibile as compared to existing STM designs.

Finally, the System.Transactions framework does not directly support nested transactions. We therefore define our own convention for managing closed-nested transactions. We define a property, *parents* using the CLR's thread-local storage facility. The parents property holds a list of parent Transactions, which are propagated appropriately across threads, and maintained as the nesting level changes. This property complements the ambient Transaction.Current property (making up for the absence of a Transaction.Parent). Our SSTM components are aware of this extension; it is public and usable by future nesting-aware RMs as well.

8.1.4 SibEx

The SibEx component implements the xfork API and coordinates the execution of sibling transactions. It may be used without TxStore, provided that the programmer has some other (nesting-aware) resource manager to use. The primary tasks of the SibEx component include the following:

[2]System.Transactions also provides an ISinglePhaseNotification interface, which allows enlisters to avoid the prepare phase when they are the only resource manager in the transaction. However, our prototype does not currently implement this optimization. Since our two components are neither integrated with the CLR nor with each other, they would appear as two separate RMs and thus render the optimization unusable.

```
interface IEnlistmentNotification
{
  void Commit (Enlistment enlistment);
  void InDoubt (Enlistment enlistment);
  void Prepare (PreparingEnlistment
                preparingEnlistment);
  void Rollback(Enlistment enlistment);
}
```

Figure 8.1: The System.Transactions.IEnlistmentNotification interface, implemented by both TxStore and SibEx.

- Scheduling the work for each fork to execute on some thread.
- Creating the sibling nested transactions.
- Invoking the fork procedure to execute within the proper context.
- Enforcing the semantics of each sibling form, which may include re-trying aborted transactions, killing unnecessary transactions, or coordinating the commit of multiple sibling transactions.
- Stalling the parent transaction/thread that calls xfork until the sibling transactions complete.

The implementation of each step is described in Section 8.2.

8.1.5 TxStore

The TxStore component provides basic software transactional memory functionality with support for parallel nested transactions. We built this component because the CLR does not provide built-in support for transactional memory. TxStore can be used even without the xfork API provided by the SibEx component. However, without the xfork API, coordinated sibling transactions are not available to the programmer. We configure some of our experiments to not use the xfork API as a baseline to measure the additional benefits of sibling transactions.

TxStore differs from previous STM designs due to requirements from the runtime environment and SibEx. The key differences are:

- TxStore is a generic store. TxStore operates at the granularity of objects, not memory bytes. The objects themselves may be integers, booleans (or any other built-in primitive types), *as well as* user-defined classes and structs.
- TxStore is a resource manager (from the System.Transactions point of view).
- TxStore supports true closed-nested transactions, as well as concurrent access by nested transactions of the same parent.

The detailed design of TxStore is presented in Section 8.3.

8.2 SibEx: Design and Implementation

The *Sibling Executive* (SibEx) is the component responsible for executing and coordinating sibling transactions. It contains the logic that transforms regular concurrent nested transactions into sibling transactions with the desired semantics. The SibEx assumes that the system supports concurrent nested transactions. As noted in the previous section, we were able to add this behavior to the CLR, and both SibEx and the TxStore are aware of our extensions. In Section 8.1.4, we gave a high-level view of the tasks that the SibEx must perform to execute an xfork request. This section discusses how each of those tasks is carried out.

8.2.1 Scheduling work

Design The first task is to schedule execution of different forks across a set of threads. The parent thread (which calls xfork) can also execute work items.

Implementation Our prototype uses the CLR's built in ThreadPool, which maintains a set of worker threads managed according to workload. Xfork packs the relevant state for each fork (forkProc, forkNum, data, SibExCoordinator, and the parent transaction) into a context object, and requests execution on a thread using ThreadPool's QueueUserWorkItem method. This is done once for each fork. We rely on the ThreadPool to determine the optimal number of threads to use, given hardware resources and the workloads of the application and system. The ThreadPool calls back into SibEx using a provided callback function, which was passed in to QueueUserWorkItem.

8.2.2 Creating sibling transactions

Design Once a fork is executed on a separate thread, the system must create a sibling transaction before invoking the forkProc. The sibling transaction must be a child of the parent thread's transaction.

Implementation The SibEx callback (invoked by the ThreadPool) creates a transaction using the TransactionScope mechanism. At a high level, this process is similar to the atomic keyword in existing STM proposals. The created transaction looks like a regular top-level transaction to the system. However, we ensure that the parent transaction, passed through the context object, is made available to resource managers by adding it in our thread-local parents list.

8.2.3 Invoking the fork procedure

Design The forkProc is then invoked within the context of the sibling transaction. The forkProc terminates either with a return code (success/failure) or an exception. In general, exceptions will be treated as an abort, and are functionally equivalent to the forkProc returning false. Depending on the type of sibling transaction, aborting a sibling transaction does not necessarily result in an xfork operation failure.

Implementation In the CLR, function pointers are represented by objects of type System.Delegate. The SibEx callback synchronously invokes the user-provided delegates, which execute within the transactional context. We define two new exceptions, TransactionRetryException and TransactionDoomedException, which have the obvious semantics when thrown by system components or the forkProc.

8.2.4 Enforcing the semantics of each sibling form

We previously defined the semantics of each of the sibling transaction forms (AND, OR, XOR). Since each form requires different logic, we implement three subclasses of an abstract class, SibExCoordinator, to perform the coordination specific to each of the forms. The SibExCoordinator class is used by the xfork method and the various threads executing the sibling transactions. The appropriate type of SibEx-Coordinator is created for each instance of xfork invocation, and passed to all the thread-pool threads as part of the work-item context object. Our design allows custom policies to be defined (as other implementations of the abstract class), which is important for enabling future research. We discuss the internals of the three provided SibExCoordinator implementations and how they interact with executing sibling transactions.

SibExCoordinator-OR

Design The simplest of the SibExCoordinator implementations is the one that handles OR-form invocations of xfork. This form is most similar to regular parallel nested subtransactions, the only notable difference being that the xfork implicitly performs a join before returning control to the parent transaction. The sibling transactions are independent, with each fork completing (commit or abort) without influencing the outcome of other transactions. The SibExCoordinator is informed by the fork threads whenever forkProc execution begins, ends, or throws an exception.

The SibExCoordinator-OR attempts to retry any aborts caused by conflicts. Finally, when all forks have completed execution, the parent is signalled that the fork is over. A bitmask of the succeeding forks can be returned from the xfork for the programmer's use.

Implementation The SibExCoodinator-OR must retry transactions that abort due to conflicts. Here we encounter an important distinction between System.Transactions and traditional STM transactions: the former do not automatically retry. CLR transactions do not retry because they are designed in the database-style, not the TM style. The SibExCoordinators decide whether to re-execute the forkProc in a new transaction based on the result of the previous transaction execution. The transaction executions results in a commit (with success or failure return codes), an abort, or an exception.

To know whether a transaction aborted or committed, the SibExCoordinator must subscribe to the outcome-notification using the Transaction.TransactionCompleted event of each sibling transaction created. Once the SibExCoordinator determines whether a given fork has committed or aborted, it decides if a restart is appropriate. Since restarted transactions manifest as entirely new transactions, the SibExCoordinator is given a chance to subscribe to their outcome notification before forkProc execution begins.

SibExCoordinator-AND

Design The SibExCoordinator-AND has one key difference from the SibExCoordinator-OR: it must influence, not simply learn, the outcome of each transaction. The semantics of the AND form mandate that none of the sibling transactions commit unless they all commit. They are similar in some respects to a traditional fork/join barrier.

Note that the xfork API reports failures to the parent thread only when it can guarantee that *none* of the transactions have committed. Execution never returns to the caller in a state where some forks have committed while others have aborted. Thus, the parent transaction always knows exactly what the program state is after the xfork and can continue execution appropriately.

Implementation The SibExCoordinator-AND enlists in every sibling transaction before the forkProc is called. By enlisting, it participates in the two-phase commit process, and thus can influence the outcome of each transaction. It does not allow any transaction to commit unless they all reach the prepare phase. If this occurs, then xfork returns a successful prepare to all of the transactions. If any fork fails, xfork aborts all sibling transactions.

The possibility remains that one or more of the transactions will still abort after being prepared by SibEx (due to decisions of other resource or transaction managers). As mentioned previously, in situations where the implementation is unable to meet the atomicity guarantee of the xfork API, it forces a restart of the calling (parent) transaction.

SibExCoordinator-XOR

Design The SibExCoordinator-XOR is similar to the AND coordinator in that it must influence the outcome of a transaction. It has two additional requirements. First, only one of the sibling transactions is allowed to commit. Second, if any transaction commits, then any other sibling forks (and transactions) still executing are useless and no longer needed, and should be aborted/terminated. This termination can be accomplished by actually tearing down the executing threads, or by some other mechanism to preempt the execution of the forkProcs on those threads.

We maintain a state machine to coordinate the states of the different sibling transactions. If no transaction has yet committed, the coordinator only allows one transaction to proceed past the prepare phase, stalling any others that reach that phase. If the outstanding prepare request commits, then all other prepare requests are aborted and all future transactions must abort. However, if the outstanding prepare does not commit, then another prepare request is allowed to attempt a commit.

Implementation Terminating unneeded forks and aborting their associated sibling transactions involves two steps. First, the actual transactions are aborted by failing all prepare requests that arrive after a sibling has committed. This is accomplished by virtue of the SibEx coordinator being enlisted in the transaction (and thus having a vote). As for terminating the execution of the fork, it was impracticable to use the Thread.Abort API to actually terminate the threads. Not only is the API discouraged (and very slow), but the threads we use are owned by the ThreadPool (and thread-replacement policies introduce additional delays for subsequent work items). Instead, we terminate the forkProc's execution, without destroying the actual thread-pool thread.

This is done by having the coordinator call into the resource-manager (the TxStore), and informing it of the set of transactions it knows it will abort (i.e. those that are doomed). The TxStore then throws a TransactionDoomed exception when a doomed transaction attempts to perform an operation on it. This exception is ultimately caught and swallowed by the SibEx callback executing on the thread-pool thread. This method allows quick cleanup (assuming that the siblings will frequently be invoking the TxStore). An implementation of sibling transactions that is more integrated in the CLR runtime environment would not have to make this assumption, as it would be able to hijack control of any thread at more opportunities.

8.2.5 Stalling the parent transaction

Design The xfork call must know when it is safe to return control to the calling thread (parent transaction). As discussed above, the SibExCoordinator managing an invocation of xfork knows when it is safe to return, as well as the operation's result. This event does not necessarily mean that all sibling transactions have finished execution on the other threads, only that the necessary committed ones have completed. Other transactions in the XOR form may still be executing, but ultimately do not affect the execution of the parent transaction because the coordinator ensures that they will abort.

Implementation Our parent thread waits on a ManualResetEvent provided by the SibExCoordinator, which is set when it is safe to return to the parent transaction.

8.3 TxStore: Design

The TxStore is a nesting-aware transactional object store used by our prototype. This section describes its API, high-level design and key data structures and operations.

8.3.1 API

The TxStore exposes a public API, which may be used directly by the programmer, or, when tightly integrated with a language or runtime, may be indirectly accessed on the programmer's behalf. The API, shown in Figure 8.2, has transactional semantics and is similar to a dynamically-addressed dictionary. Objects that are written must implement the System.ICloneable interface (if they are not value-copied primitives), which allows the TxStore to make a clone for storage and to clone objects in response to read requests. Cloning objects ensures that active transactions do not directly manipulate the stored object, but rather a clone in their working set.

Unlike most previous STMs, which operate at the word or class level, TxStore is a generic (i.e. based on System.Object) store that can be used for both. TxStore leverages the CLR's ability to treat any variable as an object (including

```
class TxStore
{
  object Read(int address);
  void Write(int address, object obj);
  int Allocate()
  void Free(int address)
}
```

Figure 8.2: TxStore API

primitive types, which also derive from System.Objects), relying on the transparent boxing and unboxing[3]. of CLR objects. TxStore does not define the granularity of the objects in the programmer-visible method. The language designer decides the mapping of programmer-visible classes to TxStore-managed objects, and the TxStore implementation handles both. For example, an object with five member fields can be stored as one object within the TxStore, or each member of the object can be stored as a separate object within the TxStore. The finer granularity can be used to reduce conflicts, such as when different members are updated concurrently by different transactions. Our prototype assumes that the entire object is treated as a single unit.

8.3.2 Design

The TxStore design can be seen as an extension of TL2 [21], adapted to support parallel nested transactions. As with TL2, the design accepts serializable executions (equivalent to 2-phase locking) with conflict detection based on time-stamps. TL2 maintains a single write timestamp per transaction, and only maintains read versions for items in the working set. However, TxStore maintains write versions for each item in the working set. The relation of main memory to TL2 transactions is similar

[3]Value types, such as integers, which are usually stored on the stack, are automatically transferred to heap memory managed by the garbage collector, if needed, and vice versa. This process is called boxing and unboxing, and is transparent to the programmer.

to that of parent transaction working sets to parallel nested transactions. TxStore's use of per-item write versions allows parallel nested transactions to modify different subsets of a single parent's working set.

TxStore uses lazy conflict detection, and the read set is validated at commit time against a parent transaction's copy or the committed item. Isolation is provided by locking the write set at prepare time, and holding these locks until commit or abort. Deadlocks cannot arise if the TxStore is being used alone, as conceptually all locks are acquired in a known order. However, when used with coordinated sibling transactions, deadlocks can arise in certain cases. We deal with any deadlocks by using a simple timeout mechanism. All internal data structures (including working sets) need to be protected against concurrent access by children transactions. Our implementation uses locking for simplicity (though we anticipate hardware assistance could be particularly helpful in this area).

8.3.3 Data structures

Internally, the TxStore maintains two maps. The first is a map of committed objects (StoredObjects), indexed by address (see Figure 8.3). The second map is of active transactions that are accessing the TxStore. Each transaction, whether top-level or nested, is tracked using an internal RMTransaction object. The nesting relationships between transactions is tracked by the TxStore, using a pointer to the parent RMTransaction. Beyond the status field, the most critical member of RMTransaction objects is the working-set, a hashtable of AccessedObjects indexed by address. Figure 8.3 shows the fields of the AccessedObject class.

8.3.4 TxStore operations

We present the high-level logic for a transaction performing a read and write operation (assuming a previously created object is being accessed). We also show the

```
class StoredObject
{
  object realObj;
  int versionNum;
  bool deleted;
}
class AccessedObject
{
 object realObj;
 int readVersionNum;
 int changeVersionNum;
 bool readFlag;
 bool writtenFlag;
 bool addedFlag;
 bool deletedFlag;
 StoredObject lockedObject;
}
```

Figure 8.3: TxStore internal data structure. The StoredObject is for committed objects, while the AccessedObject is maintained by active transactions in their working sets.

steps performed on transaction commit and abort. We omit the allocate/free APIs, as well as the edge cases where the data structures may concurrently be accessed while being allocated or freed. These cases are all available in the source code, which will be made publicly available.

TxStore.Read:

1. Enlist in the transaction if necessary, creating the RMTransaction internal object.

2. Look in the RMTransaction.working-set for the address. If found, then return a clone of AccessedObject.realObj.

3. If not found, search the chain of parent RMTransactions for the first working-set that contains the requested address. If found, propagate it down to all working sets in the chain, including that of the current transaction, while

setting the readFlag for each AccessedObject created. Each newly created AccessedObject's readVersionNum field is copied from the changeVersionNum of the AccessedObject found at the top of the chain. To protect against races from other concurrent children, each RMTransaction object has a lock that is acquired while accessing/modifying its working set.

4. If no parent has accessed the object, look up the address in the StoredObjects map and copy the realObj into the current RMTransaction's working set. The StoredObjects map is synchronized for the duration of this lookup (currently using a lock though finer synchronization is possible). We set the readFlag for the created AccessedObject, and its readVersionNum is copied from the StoredObject's versionNum.

TxStore.Write:

1. Enlist in the transaction if necessary, creating RMTransaction internal object.
2. If the RMTransaction working-set does not contain an AccessedObject for the requested address, create one.
3. Write the object into the existing or created AccessedObject.realObj field, and set the writeFlag. A newly allocated version-number is written into the changeVersionNum field.

RMTransaction.Prepare:

1. Lock StoredObject map.
2. For each AccessedObject in the RMTransaction's working-set:
   (a) Lock the StoredObject, and set the AccessedObject's lockedObject field to point to the StoredObject.

   (b) If the AccessedObject readFlag is set, ensure that the readVersion is equal to the nearest parent's AccessedObject (changeVersionNum, otherwise readVersionNum), or the StoredObject's versionNum. If not, release all locks, and fail the Prepare request.

3. Unlock the StoredObject map.

RMTransaction.Rollback:

1. For each AccessedObject in the RMTransaction's working set, release the lockedObject if it is not-null. This is done without taking the StoredObject map lock.

RMTransaction.Commit:

1. Lock the RMTransaction object to prevent any races from concurrent children.

2. For each AccessedObject in the RMTransaction's working-set:

   (a) If the transaction is a top-level one and the AccessedObject writeFlag is set, copy the realObj and the changeVersionNum to the StoredObject's realObj and versionNum fields.

   (b) If the transaction is nested and the parent does not have an AccessedObject for the same address, transfer the entire AccessedObject to the parent's working set.

   (c) If the transaction is nested and the parent has a corresponding AccessedObject, merge the current AccessedObject with the parent's AccessedObject. If the transaction has only read (readFlag set), then no merging needs to occur because the prepare phase ensured that the version number of the child is identical to the parent regardless of whether the parent has read or written. If the transaction has written (writeFlag set), the changeVersionNum and realObj are propagated to the parent, whose writeFlag is also set.

(d) Unlock the StoredObject, which is in the AccessedObject's lockedObject field.

3. Unlock the RMTransaction object

8.4 Evaluation

This section describes our evaluation of the SSTM prototype, the benchmarks used, and the speedup results compared to our base-line STM configuration.

8.4.1 Software and hardware setup

SSTM is implemented on top of the Microsoft .NET Framework version 2.0.50727. The system consists of 2,345 lines of C# code, including about 600 lines of benchmark code. We make extensive use of the base class libraries (collection classes, etc.) and synchronization primitives (WaitEvent, Monitor, etc.), and have not bothered with many of the usual optimizations (e.g. reader-writer locks, specialized data structures, avoiding kernel calls caused by wait-objects, etc.). Our experiments are run on an Intel Core2 Quad CPU running at 2.66 Ghz, with 4GB of RAM. The OS is Microsoft Windows Vista Ultimate (32-bit), with SP1.

8.4.2 Benchmarks

We evaluate SSTM with three benchmarks, each corresponding to one of the forms of sibling transactions. They allow us to evaluate xfork overhead compared to benefit. Each benchmark is written with the serial transactional code (the base-line version), and then modified to use xfork to produce 2 and 4-core SSTM versions.

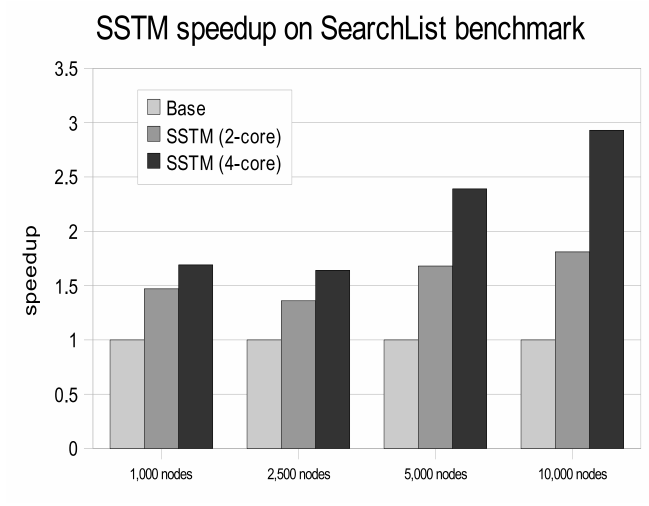


Figure 8.4: Speedup of SSTM on the SearchList benchmark, for different length lists.

SearchList

The SearchList benchmark has a transaction that finds a random element within a doubly linked list. We compare the base version, which performs a forward traversal of a linked list, with the xfork version. The xfork version uses the XOR form to search the list. As soon as one of the siblings has found the node, the others are destroyed. The 2-core version performs both forward and reverse traversals, while the 4-core version performs two additional traversals starting from the middle of the linked list (maintained by the structure).

We vary the number of nodes in the list from 1,000 to 10,000 nodes. The results are shown in Figure 8.4. Sibling transactions are able to provide a significant speedup over the regular linked list traversal code, with minimal programmer effort.

Transfer

The transfer benchmark exercises the AND form of sibling transactions. It is a debit-credit operation that is a simplified form of the transfer example presented

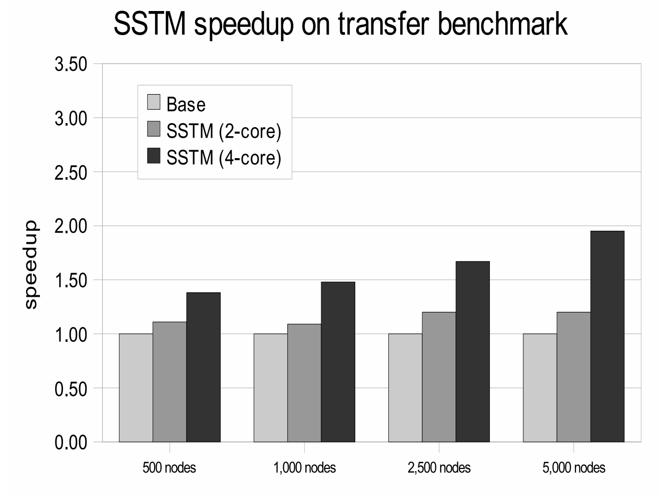


Figure 8.5: Speedup of SSTM on the Transfer benchmark, for different length lists.

earlier in this paper (only two lists are involved, not three). The accounts exist in two separate lists, and accounts are randomly selected. We incorporate a fixed think time to model the work involved when the appropriate account nodes are actually found. We compare the base transaction code, where the debit is done followed by the credit, with an xfork version, where the debit and credit are done concurrently. The debit fork returns a failure code if the balance is insufficient. The 2-core benchmark uses regular linked list traversal in each of the credit and debit forks. The 4-core benchmark adds a speculative search of each list, in effect nesting an invocation of the SearchList benchmark. The results are shown in Figure 8.5. The speedups are less pronounced compared to SearchList because of the additional overhead of AND coordination (which involves enlistments) compared to OR. The 4-core results demonstrate that multiple levels of sibling transactions are practical, and further improves performance in the presence of the additional cores.

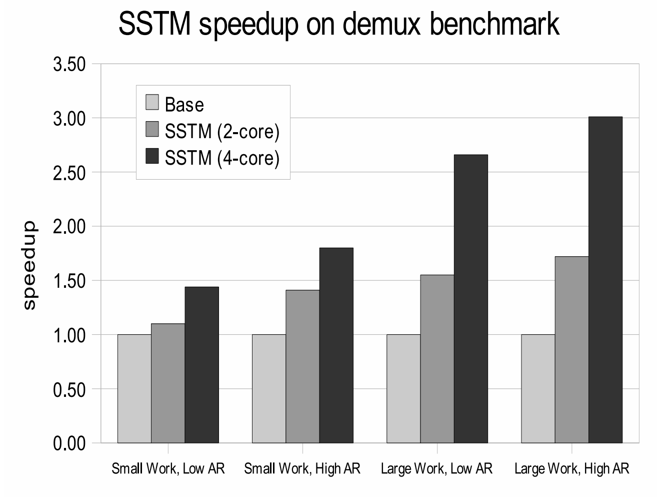


Figure 8.6: Speedup of SSTM on the demux benchmark. Low AR models no conflicts, while High AR models 12% conflict rate. Small Work uses a think-time of 10kC, while Large Work uses a think-time of 50kC.

Demux

The demux benchmark involves a transaction that dequeues a work item, processes it, and then attempts to enqueue two or more subsequent work items to further queues. All operations are done within a single transaction so that any failures leave the system in a consistent state. Enqueue operations can sometimes fail, for example when conflicts occur with other top-level transactions, or if the queue is temporarily full. The benchmark models this by failing enqueues at a specified probability. We also model the work done before the transaction enqueues by a variable think time (measured in kilo-Cycles). We execute 20,000 transactions over the duration of the run. We modified the benchmark by performing an xfork that performs the enqueue operations using the OR form, in both 2 and 4 core versions.

The results are shown in Figure 8.6 as speedups of the SSTM version to the baseline code. The SSTM versions are able to speedup the baseline version on 2 cores, and scales noticeably on 4 cores. Note that as the abort rate increases, the relative speedup of SSTM over the baseline also increases. This increase is a benefit of nested transactions in general, since less work is wasted when an enqueue operation fails, since only that action (done within a sibling transaction) is retried, not the work done by the parent transaction. The user of xfork automatically gets the benefits of nested transactions. By the same token, increasing the amount of work (think time) in the parent transaction leads to better scaling.

8.4.3 Benchmark results summary

These benchmarks show that for a very small amount of programming effort, we can make sequential transactions run even faster by leveraging additional cores while retaining the benefits of transactions. We demonstrate a speedup of up to 1.87× (on two cores) and 3.12× (on four cores) on the demux benchmark. On the more substantial transfer benchmark (with nested xforks), we see a speedup of 1.2× (on two cores) and 1.95× (on four cores). These results, obtained on our unoptimized C# implementation, are encouraging, and show that further research and experience with xfork in software and hardware is warranted, so as to understand better its strengths and limitations.

Chapter 9

Related work

9.1 Transactional Memory

In this section, we discuss the most relevant related work from the literature. Larus and Rajwar provide a thorough reference on TM research through the beginning of summer 2006 [49].

Optimistic synchronization [38,45] and optimistic database concurrency control [48] motivated early HTM designs [42]. Rajwar and Goodman explored transactional [45,67] execution of critical sections (Speculative Lock Elision), which along with technology trends, sparked a renewal of interest in HTM.

HTM Several designs for transactional memory systems have been proposed: MetaTM most closely resembles Moore et. al's LogTM [56], both in terms of its semantics and its impact on cache coherence protocols. Both MetaTM and LogTM require less modification to cache coherence protocols than TCC [32], which replaces traditional cache coherence protocols with transactions. Transactional stores in LogTM update memory values in place, with previous values logged to virtual memory. Eager conflict detection ensures that two transactions do not write to the same memory area. Because LogTM stores new values in place, commits are inexpensive, but transaction aborts require reading from a log, and are handled in software.

Like LogTM, Unbounded Transactional Memory [5] stores updated values in place, retaining overwritten values in a log. In theory, UTM allows transactions of arbitrarily large size that are able to survive paging, processor migration, and context switches. UTM has not been implemented, even by its designers. LTM [5], which is based on UTM, attempts to simplify transactional memory implementation by restricting the size and durability of transactions. LTM stores updated memory locations in the cache, overflowing to a hash table in main memory. Ananian et. al [5] evaluate the possible behavior of UTM in the Linux kernel. However, their evaluation is based on replaying memory and synchronization traces collected from a non-transactional kernel. The additional requirements imposed on transactional memory by an operating system kernel are not considered.

TCC [32] buffers transactional writes to a private cache. At commit time, values are written through to the L2 cache and conflicts are detected (lazy conflict detection). Because conflict detection occurs before values are updated in main memory, restarting a transaction is inexpensive. Lazy conflict detection, however, may lead to wasted work. Recent work [26] has added virtualization support to TCC.

Virtual Transactional Memory [68] augments cache-based transactional memory with in-memory data structures in order to allow transactions to overflow the cache and survive context switches. VTM writes updated values to memory on transaction commit, and performs eager conflict detection.

Recent HTM work has investigated semantics and new architectural designs [16, 27, 53], language-level support for HTM [14] and transactional resource virtualization [10, 26, 68, 83].

STM Software transactional memory (STM) systems [79] are an active area of research in the synchronization and programming language communities. Software transactional memory systems does not require hardware support.

While the programming interfaces for HTM and STM systems are generally isomorphic, a persistent drawback of STM systems is the relative performance. Recent advances have been made in making them more efficient [1,34,36,41,43,52].

Recent proposals have combined elements of both, resulting in, hybrids [18, 47] and hardware-accelerated STM [25] that attempt to get the performance of hardware systems without the limitations of hardware or the need to virtualize, thus preserving the advantages of both approaches.

9.2 TM Isolation.

Dependence-aware transactions detect conflicts in a way that is neither eager nor lazy [56], but rather combine strengths of both approaches. The constraints on commit order imposed by dependences have a lazy flavor, though most lazy version management systems have a first-to-commit arbitration policy, which is absent with dependences. Since multiple transactions that write the same memory cell cannot update the cell in place, DATM version management is lazy.

Other transactional memory designs and implementations have also observed that modifying the safety conditions for transactions can allow a system to extract more concurrency from workloads. Along with DATM [69,72], TSTM [7] identifies that using conflict serializability as a correctness criteria, rather than two-phase locking, benefits transactional memory systems by allowing more concurrency. TSTM is based on timestamp ordering, and does not accept every conflict serializable schedule as DATM does (e.g. those that involve forwarding). In particular, their implementation would not allow concurrent updates to a shared counter.

SI-STM uses snapshot isolation, a weaker isolation level than conflict-serializability [76].

SI-STM shares some of the performance goals of DATM, trying to get conflicting transactions to commit. It also shares some implementation techniques with DATM, namely preserving multiple versions of the same memory byte. Snapshot isolation is more difficult for the programmer to reason about than conflict serializability and is applicable to fewer situations than DATM.

The developers of CS-STM [77] (which utilizes a new consistency criterion that the authors call z-linearizable), also consider a variant of that algorithm which maintains full serializability. This variant (called S-STM) is only briefly described as using timestamps and vector clocks and having to maintains partial precedence graphs. The authors state that the runtime overhead of managing their intricate data structures can be prohibitive, especially for smaller transactions, though performance data is not reported.

9.3 TM programming model extensions

Several proposed extensions to the TM programming model can be used to achieve higher performance, including privatization [81], early release [80], escape actions [83], open and closed nesting [61, 63], Galois classes [46], transactional boosting [40] and abstract nested transactions [35].

These techniques all fundamentally affect the programming model, increase programmer effort, and increase program complexity as 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.

Privatization [81], and its complement (publication) are programming technique that allows programmers to carefully manage when data is shared and accessible by other transactions, versus being private. They bridge the conceptual gap between per-CPU data structures and shared data structures. Early release [80] allows a programmer to drop transactional isolation on given memory locations. The programmer must be correct in his judgment that isolation is not needed on those locations, or the program will no longer be correct. Having code in an escape action access the same data as a paused transaction can cause semantic anomalies [59]. Open nesting [60] trades physical isolation for logical isolation, with the programmer guaranteeing correctness. All of these techniques require more programmer effort than DASTM.

These proposals often are a good fit in limited applications, and usually require the programmer to be very careful to avoid subtle consistency and isolation problems. By contrast, coordinated sibling transactions are more general, straightforward to use and retain the full protection of closed-nested transactions.

Galois classes [46] and transactional boosting [40] allow the programmer to provide inverse operations for the concurrent data structures. These techniques are orthogonal to dependence-awareness, and can be used to complement them. They have the potential to eliminate structural conflicts in many situations, though programmers may have varying success providing inverses for different data structures (e.g., k-d tree inserts are not straightforward to handle), and defining commutativity relationships between all operations. Similarly, abstract nested transactions (ANTs) [35] attempt to reduce the performance effect of benign conflicts, but require the programmer identify the regions of code that are likely to be victims of such conflicts. The system then ensures that ANTs are re-executed appropriately if they do experience a conflict. ANTs differ from closed nesting in when the re-executing can occur, specifically ANT re-execution can be delayed until the top-level transaction attempts to commit.

Language extensions that are related to sibling transactions include orElse [37] and t_for [31] constructs, the latter focusing on the well-known issue of loop parallelization. Fortress [4] allows (independent) parallel nested atomic blocks, as does XCilk [2].

9.4 Databases

Spheres of control Transaction dependences also have roots in early database research. Spheres of control [19], in the context of a static hierarchy of abstract data types, introduced the notion of dynamic spheres created around actions accessing shared data. Transaction dependences are at one level a refinement and formalization of the general notion of spheres of control in a way that can be implemented in the context of transactional memory.

Multi-version concurrency Time-domain addressing [75] (also called multi-version concurrency control (MVCC)) tracks multiple versions of objects modified concurrently, along with time information that allows different transactions to access different versions of the object. DATM implementations have to deal with some of the same issues that arise in MVCC systems. However, write-shared data are known to degrade MVCC performance, while DATM is designed to scale in their presence. Also, the techniques DATM employs to achieve conflict-serializability (notably, the different types of dependences and the forwarding of uncommitted data) are not found in MVCC systems.

Custom isolation Modern databases deal with hotspots as well, and the concurrency control mechanism (locking) makes it difficult to scale certain scenarios without special support. The high-contention counter arises in the context of generating sequential, unique identifiers. Support includes data definition language changes (e.g. marking a field as *autoincrement*, so new rows are assigned unique values), as well as extensions (such as Sequence objects [65]) which operate outside the scope of the transaction, but therefore have weaker semantics.

Concurrency control A standard taxonomy for database concurrency control mechanisms distinguishes between pessimistic and optimistic concurrency control.

Optimistic concurrency control mechanisms can further be classified as either timestamp-based, or validation-based. Within this taxonomy, DATM can be viewed as an efficient implementation of a SGT-based (Serialization Graph Testing-based) certifying scheduler [9]. It is designed to permit *recoverable* schedules, a superset of ACA (Avoid Cascading Aborts) schedules. It does not build up the actual serialization graph, since the dependences on the shared objects provide sufficient information to provide the necessary constraints.

9.5 Interrupts in existing HTM systems

Much existing work on HTM systems [5, 42, 56, 66–68] makes several assumptions about the interaction of interrupts and transactions. These works assume that transactions are short and that interrupts are infrequent enough to rarely occur during a transaction. As a result, efficiently dealing with interrupted transactions is unnecessary. They assume that interrupted transactions can be aborted and restarted, or their state can be virtualized using mechanisms similar to those for surviving context switches.

Nested LogTM [57] and Zilles [83] allow transactional escape actions that allow the current transactional context to be paused to deal with interrupts. However, both of these systems do not allow a thread with a paused transaction to create a new transaction. A design goal of MetaTM is to enable transactions in interrupt handlers and data in Table 4.4 show anywhere from 11–60% of transactions in TxLinux come from interrupt handlers.

XTM [26] makes significant assumptions about the flexibility of interrupt handling. When an interrupt happens in XTM, the interrupt controller calls into the OS scheduler on a selected core.

The scheduler runs inside of an open nested transaction so it does not affect any ongoing transaction. If the interrupt is not critical, it is handled after the current transaction completes. Otherwise, the current transaction is aborted. If this method leads to a long transaction being repeatedly aborted, the transaction is virtualized, so that further interrupts do not affect it.

9.6 Multiple active transactions

Concurrent with our proposal of multiple active transactions using **xpush** [70, 71], other work introduced the **xact_pause** primitive [83] to pause transactions. This semantics of these two primitives are quite different. **Xact_pause**, which was designed for use cases where weaker forms of atomicity are traded for performance, would not be usable in the interrupt-handling setting of TxLinux. The **xpush** instruction truly suspends, or pauses, the active transaction, and allows code (such as interrupt handlers) to execute either non-transactional operations, or to start new, completely independent transactions. Multiple calls to **xpush** are allowed. **Xact_pause**, on the other hand, does not allow new transactions to be started during the pause. Moreover, the non-transactional code is not actually independent from the paused transaction—instructions executing during the pause can access uncommitted transaction state. The target use cases for **xact_pause** require communicating values between a transaction and the non-transactional code. In this sense the transaction is not really *paused*; the programmer is switching from memory-cells to higher level units of resource management, which is also evidenced by their use of higher-level compensating actions. Escape actions [57] provide weaker still isolation, and have been proposed as a way to deal with system calls and interrupts in operating systems, as well as low-level debugging scenarios. As with **xact_pause**, code executing within an escape action cannot start new a transaction. Conflict detection and version management are bypassed, and escape actions are allowed to register commit and compensating actions to release isolation when the enclosing transaction commits.

Nested LogTM [57] has a problem analogous to the transactional dead stack problem—nested transactions can modify the same stack address via aliasing (causing an O1 violation). Nested LogTM allows transactions to start and end in different stack frames. The published material [57] contains little detail and suggests using the bottom of the page to delimit the bottom of the stack. This heuristic will not work for the Linux kernel, and many user programs, which have stacks that are larger than a page.

9.7 Nested transactions

Nested transactions Nested transactions were introduced long before transactional memory [8, 62], and were used in early distributed systems such as Argus [50] and Camelot [24]. Nested transactions are only one type of advanced transaction model. Others are surveyed by Gray [29] (ch. 4) and Weikum [82]. Application-specific transaction models tend to be even more complex (and less general). These are surveyed by Elmagarmid [23] and include cooperating transactions, used by some CAD applications, which interact to pass ownership of resources.

Transactional memory nesting Transactional memory systems initially did not implement any real nesting, making do with a flat transaction model. More recently, however, there have been several proposals that incorporate true nesting into transactional memory, both in hardware [53,58] and software [35,64]. Recent TM systems have looked at linear (non-parallel) closed-nesting [58,60], but found its performance lacking [58].

Parallel nested transactions Parallel nesting is as old as nesting itself, and has also been proposed recently in the TM context [2]. However, Agrawal's model treats each nested transaction as completely independent of its siblings. We believe this model insufficiently expresses some of the intuitive sources of parallelism, and should be complemented with siblings that can affect each other's outcomes.

Chapter 10

Conclusion

Considering the current state of the art, almost every transaction model, except for flat transactions and distributed transactions, can be considered exotic.

... However, with applications getting bigger, more integrated, and thus more complex, there are numerous types of processing that are not well-served by the simplicity of flat transactions.

- Jim Gray and Andreas Reuter, *Transaction Processing* [29], p.219, 221

As the above quote indicates, it is challenging, but necessary, to move beyond flat transactions. This thesis has examined, in the new context of transactional memory, how transaction models can be modified to increase concurrency between and within transactions. It introduced the following models:

- **Suspended transactions:** Allow multiple concurrent transactions on the same processor
- **Dependence-aware transactions:** Convert conflicts into commits, by accepting more schedules

- **Coordinated sibling transactions:** Makes intra-transaction parallelism a commodity

Several TM systems that use these models were implemented and evaluated, either as hardware simulations and software systems. Several new benchmarks were developed as part of this thesis, the most significant of which is the TxLinux (a variant of Linux), the first OS to use TM for synchronization, and one of the largest existing TM workloads.

10.1 MetaTM and TxLinux

Previously, the design decisions necessary for implementing hardware transactional memory have only been evaluated in the context of micro- and application benchmarks. The dependence of operating systems on extreme concurrency and the complex synchronization necessary to achieve such concurrency, however, makes them an ideal workload for evaluating such synchronization primitives. Hardware transactional memory has the potential to greatly simplify operating system synchronization while retaining a high degree of concurrency.

Operating systems also represent a unique workload for transactional memory due to their position as the arbiters between computer hardware and software. We have shown that asynchronous events such as interrupts require special consideration when designing transactional memory hardware.

We have examined several aspects of hardware transactional memory implementation and policy in the context of an operating system workload. We find that some backoff on contention is important and both linear and exponential backoff work well. Transaction restart penalties can have similar performance effects to explicit backoff policies.

10.2 Dependence-aware TM

Dependence-aware transactions increase throughput by enabling concurrent execution of transactions that would otherwise conflict due to updating shared data structures. This paper formalizes the DATM model and presents the design, and evaluation of a prototype implementation of the first dependence-aware software transactional memory system. Experimental results from our prototype confirms the potential performance benefits of dependence-aware transactional memory as compared to traditional TM 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.

10.3 Coordinated sibling transaction

The xfork API allows programmers to easily express inherent concurrency within their atomic sections, while retaining the simplicity of transactions. Coordinated sibling transactions is introduced as a model that can make parallel nested transactions a commodity. We presented the design of our prototype system built on the CLR. The evaluation shows the efficacy and potential of this model.

Bibliography

[1] A.-R. Adl-Tabatabai, B. Lewis, V. Menon, B. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI*, Jun 2006.

[2] Kunal Agrawal, Jeremy T. Fineman, and Jim Sukha. Nested parallelism in transactional memory. In *PPoPP*, 2008.

[3] A. Alameldeen and D. Wood. Variability in architectural simulations of multi-threaded workloads. In *HPCA*, 2003.

[4] E. Allen, D. Chase, J. Hllett, V. Luchango, J. Maessen, S. Ryu, and S. Tobin-Hochstadt. *The fortress Language specification 1.0*, 2007.

[5] C. Anaian, K. Asanovic, B. Kuszmaul, C. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA*, 2005.

[6] A. Arcangeli, M. Cao, P. McKenney, and D. Sarma. Using read-copy-update techniques for system V IPC in the Linux 2.5 kernel. In *USENIX Annual Technical Conference, FREENIX Track*, pages 297–309, 2003.

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

[8] C. Beeri, P. Bernstein, N. Goodman, M. Lai, and D. Shasha. A concurrency control theory for nested transactions. In *PODS*, 1983.

[9] Philip Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, 1987.

[10] C. Blundell, J. Devietti, E. C. Lewis, and M. Martin. Making the fast case common and the uncommon case simple in unbounded tm. In *ISCA*, 2007.

[11] C. Blundell, E.C. Lewis, and M. Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *Proceedings of the Fourth Workshop on Duplicating, Deconstructing, and Debunking*, Jun 2005.

[12] J. Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer*, 1994.

[13] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *IEEE International Symposium on Workload Characterization (IISWC)*, Sep 2008.

[14] B.D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. Cao Minh, C. Kozyrakis, and K. Olukotun. The Atomos transactional programming language. In *PLDI*, Jun 2006.

[15] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th OOPSLA*, 2005.

[16] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. v.Biesbrouck, G. Pokam, B. Calder, and O. Colavin. Unbounded page-based transactional memory. In *ASPLOS*, 2006.

[17] J. Chung, H. Chafi, C. Cao Minh, A. McDonald, B. Carlstrom, C. Kozyrakis, and K. Olukotun. The common case transactional behavior of multithreaded programs. In *HPCA-12*. Feb 2006.

[18] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS-XI*, 2006.

[19] Charles T. Davies. Data processing spheres of control. *IBM Systems Journal*, 17(2), 1978.

[20] D. Dice and N. Shavit. What really makes transactions faster? In *ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Jun 2006.

[21] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC*, Sep 2006.

[22] Ulrich Drepper. *Futexes are tricky*, 2005.

[23] A. Elmagarmid. *Database Transactional Models for Advanced Applications.* Morgan Kaufmann, 1992.

[24] J.L. Eppinger, L.B. Mummert, and A.Z. Spector. *Camelot and Avalon: A Distributed Transaction Facility.* Morgan Kaufmann, 1991.

[25] B. Saha et al. Architectural support for software transactional memory. In *MICRO*, 2006.

[26] J.W. Chung et al. Tradeoffs in transactional memory virtualization. In *ASPLOS*, 2006.

[27] L. Yen et al. Logtm-SE: Decoupling hardware transactional memory from caches. In *HPCA*. 2007.

[28] Hubertus Franke, Rusty Russel, and Matthew Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in Linux. In *Proceedings of the Ottawa Linux Symposium*, 2002.

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

[30] R. Guerraoui, M. Kapalka, and J. Vitek. Stmbench7: A benchmark for software transactional memory. In *EuroSys*, 2007.

[31] L. Hammond, B. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency. In *ASPLOS*, October 2004.

[32] L. Hammond, V. Wong, M. Chen, B. Carlstrom, J. Davis, B. Hertzberg, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA*, June 2004.

[33] L. Hammond, V. Wong, M. Chen, B. Hertzberg, B. Carlstrom, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA*, 2004.

[34] T. Harris, M. Plesko, A. Shinnar, and D.Tarditi. Optimizing memory transactions. In *PLDI*, Jun 2006.

[35] T. Harris and S. Stipic. Abstract nested transactions. In *TRANSACT*, 2007.

[36] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA*, pages 388–402, Oct 2003.

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

[38] M. Herlihy. Wait-free synchronization. In *TOPLAS*, January 1991.

[39] M. Herlihy and E. Koskinen. Dreadlocks: Efficient deadlock detection for stm. In *TRANSACT*, Feb 2008.

[40] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP*, 2008.

[41] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *OOPSLA*, 2006.

[42] M. Herlihy and J. E. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, May 1993.

[43] Maurice Herlihy, Victor Luchangco, Mark Moir, and III William N. Scherer. Software transactional memory for dynamic-sized data structures. pages 92–101, Jul 2003.

[44] 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.

[45] T. Knight Jr. An architecture for mostly functional languages. In *ACM Conference on LISP and Functional programming*, 1988.

[46] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *PLDI*, Jun 2007.

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

[48] H.T. Kung and J.T. Robinson. On optimistic methods of concurrency control. In *ACM Transactions on Database Systems 6(2)*, 1981.

[49] J. Larus and R. Rajwar. *Transactional Memory.* Morgan & Claypool, 2006.

[50] B. Liskov. Distributed programming in argus. In *CACM 31(3)*, 1988.

[51] Nancy A. Lynch, Michael Merritt, William E. Weihl, and Alan Fekete. *Atomic Transactions*. Morgan Kaufmann, 1993.

[52] V. Marathe, M. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. Scherer III, and M. Scott. Lowering the overhead of nonblocking software transactional memory. In *TRANSACT*, 2006.

[53] A. McDonald, J. Chung, B.D. Carlstrom, Cao C.M., H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. In *ISCA*, Jun 2006.

[54] V. Menon, S. Balensiefer, T. Shpeisma, A. Tabatabai, R. Hudson, B. Saha, and A. Welc. Single global lock semantics in a weakly atomic stm. In *TRANSACT*, 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*, Jun 2007.

[56] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, , and D. A. Wood. Logtm: Log-based transactional memory. In *HPCA*, 2006.

[57] M. Moravan, J. Bobba, K. Moore, L. Yen, M. Hill, B. Liblit, M. Swift, and D. Wood. Supporting nested transactional memory in LogTM. In *ASPLOS-XII*, 2006.

[58] 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*, 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] E. Moss and T. Hosking. Nested transactional memory: Model and preliminary architecture sketches. In *SCOOL*, 2005.

[61] J. E.B. Moss. *Nested transactions*. MIT, 1985.

[62] J.E.B. Moss. Nested transactions: An approach to reliable computing. In *MIT LCS-TR-260*, 1981.

[63] Y. Ni, V. Menon, A. Tabatabai, A. Hosking, R. Hudson, J. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *PPoPP*, 2007.

[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*, 2007.

[65] Oracle Corp. *Guide to using SQL Sequence Number Generator*.

[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*. Jun 2005.

[69] H. Ramadan, C. Rossbach, O. Hofmann, and E. Witchel. Dependence-aware transactional memory. Technical Report TR-07-58, University of Texas at Austin, Computer Sciences Department, 2007.

[70] H. Ramadan, C. Rossbach, D. Porter, O. Hofmann, A. Bhandari, and E. Witchel. Metatm/txlinux: Transactional memory for an operating system. In *ISCA*, 2007.

[71] H. Ramadan, C. Rossbach, and E. Witchel. The Linux kernel: A challenging workload for transactional memory. In *Workshop on Transactional Memory Workloads*, June 2006.

[72] H. Ramadan, C. Rossbach, and E. Witchel. Dependence-aware transactions for increased concurrency. In *MICRO-41*, 2008.

[73] H. Ramadan, I. Roy, M. Herlihy, and E. Witchel. Committing conflicting transactions in an stm. In *PPOPP*, 2009.

[74] H. Ramadan and E. Witchel. The xfork in the road to coordinated sibling transactions. In *TRANSACT*, 2009.

[75] David P. Reed. Implementing atomic actions on decentralized data. *ACM TOCS*, 1(1), 1981.

[76] Torvald Riegel, Christof Fetzer, and Pascal Felber. Snapshot isolation for software transactional memory. In *TRANSACT*. 2006.

[77] Torvald Riegel, Heiko Sturzrehm, Pascal Felber, and Christof Fetzer. From causal to z-linearizable transactional memory. Technical Report RR-I-07-02.1, Universite de Neuchatel, Institut d'Informatique, February 2007.

[78] William N. Scherer III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC*, 2005.

[79] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213, Aug 1995.

[80] T. Skare and C. Kozyrakis. Early release: Friend or foe? In *Workshop on Transactional Memory Workloads*, Jun 2006.

[81] Michael Spear, Virendra Marathe, Luke Dalessandro, and Michael Scott. Privatization techniques for software transactional memory. In *PODC*, Aug 2007.

[82] G. Weikum and H. Schek. *Concepts and Applications of Multilevel Transactions and Open Nested Transactions*. Morgan Kaufmann, 1992.

[83] C. Zilles and L. Baugh. Extending hardware transactional memory. In *TRANSACT*, Jun 2006.

[84] F. Zyulkyarov, V. Gajinov, O. Unsal, A. Cristal, E. Ayguade, T. Harris, and M. Valero. Atomic quake: Using transactional memory in an interactive multiplayer game server. In *PPoPP*, 2009.

Vita

Hany Ramadan was born in Nairobi, Kenya in 1976, son of Dr. Essam S. Ramadan and Nagwa Mahmoud. He graduated from the American University in Cairo in 1996 with a B.S. in Computer Science, followed by a M.S. in the same field from the University of Minnesota in 1999. He was employed as software engineer by Microsoft Corporation in Seattle from that time until 2005, when he started the doctoral program in the Department of Computer Sciences at the University of Texas at Austin.

Permanent Address: PO Box 8351
Austin, TX 78713

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.