Pathological Interaction of Locks with Transactional Memory

Haris Volos, Neelam Goyal and Michael M. Swift

University of Wisconsin–Madison

{hvolos,neelam,swift,}@cs.wisc.edu

Abstract

Transactional memory (TM) promises to simplify multi-threaded programming. Transactions provide mutual exclusion without the possibility of deadlock and the need to assign locks to data structures. To date, most investigations of transactional memory have looked at *purely* transactional systems that do not interact with legacy code using locks. Unfortunately, the reality of software engineering is that such interaction is likely.

We investigate the interaction of transactional memory implementations and lock-based code. We identify and discuss five pathologies that arise with different systems when a lock is accessed both within and outside a transaction: Blocking, Deadlock, Livelock, Early Release, and Invisible Locking. To address these pathologies we designed and implemented *transaction-safe locks* (*TxLocks*) by modifying the existing lock implementation of the OpenSolaris C Library and extending the conflict resolution policy of a hardware transactional memory system.

1. Introduction

Transactional memory (TM) [7] promises to simplify multi-threaded programming by removing the need to assign locks to data. However, it is likely that transactional memory must co-exist with lock-based code for the foreseeable future. While TM is most helpful to applications written from scratch, it may also simplify existing programs written using locks. In both cases, transactional code may need to invoke existing, lock-based code that has not or cannot (because it is available only in binary form) be converted to transactions. Furthermore, in existing programs, it may be useful to convert only key data structures or functions to transactions, leaving parts of the code to use locks.

When converting lock-based code to transactions, we ran across many cases where transactional code naturally acquired locks that were also acquired by non-transactional code. This happened for two reasons: (1) calls into application subsystems that were not yet transactionalized and (2) calls into standard libraries. An example of the first case occurs in the BIND DNS Server [8, 9]. BIND is composed of several subsystems that make extensive use of locks. We converted the red-black tree structure, which stores the individual name records, to transactions. This structure is frequently accessed during queries and updates and is a good fit for TM. Transactionalizing this subsystem while keeping the rest of the subsystems unmodified is sufficient to improve performance through optimistic concurrency. However, on some paths the red-black tree invokes shared logging code, which uses locks. As a result, the locks in the logging code are acquired within transactions.

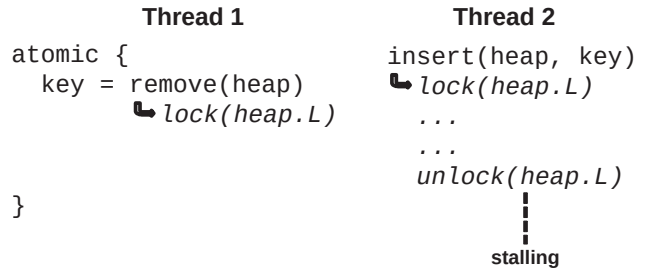

Figure 1. Interaction between lock-based and transactional code can bring the system in an unrecoverable state.

An example where transactions may interact with locks in library code occurs with the *ld.so* dynamic linker used in UNIX platforms. Dynamic linking is performed on the first call to a library function, and must use locks to prevent multiple threads from simultaneously linking a library. If a transaction makes the first call to a dynamically linked function, then the linker's locks will be acquired within a transaction.

These examples demonstrate that transactions and locks may interact when legacy code is used in transactional programs. To date, though, most investigations of transactional memory have looked at *purely* transactional systems that have no interaction with code using locks. Unfortunately, our personal experience with our own LogTM [13] system reveals that non-trivial *pathological behavior* may arise.

Figure 1 presents an example of pathological behavior when transactions and locks interact. In the example, a heap data structure provided by a library is invoked from a transaction (Thread 1) and from non-transactional code (Thread 2). When executed on the LogTM hardware TM system [13], deadlock can occur when transactional and non-transactional threads compete for a lock. Thread 2 acquires the lock non-transactionally. When Thread 1 attempts to acquire the lock within a transaction, it finds the lock in use and spins reading the lock variable. However, *the act of reading the lock variable adds it to the transaction*, preventing other threads from changing it until Thread 1's transaction completes. When Thread 2 tries to release the lock, LogTM prevents it from writing the lock variable, and thereby releas-

Pathology	Version Management		Conflict Detection		Strong /Weak	Systems
	Eager	Lazy	Eager	Lazy		
Blocking	yes	yes	yes	yes	strong	LogTM/LogTM-SE, Bulk [5]
Deadlock	yes	no	yes	no	strong	LogTM/LogTM-SE, OneTM [2]
Livelock	yes	yes	yes	yes	strong	Bulk, LTM [1]
Early Release	yes	no	yes	yes	weak	McRT-STM [17]
Invisible Locking	no	yes	yes	yes	weak	TL2 [6]

Table 1. Summary of the lock pathologies together with TM systems affected by these pathologies.

ing the lock, in order to preserve the transaction's isolation. Thus, the program deadlocks because Thread 1 cannot complete its transaction until Thread 2 releases the lock. Thread 2 cannot release the lock until Thread 1's transaction completes.

While this pathology has been presented in the context of LogTM, it applies to other systems that block non-transactional code behind transactions, such as OneTM [2]. Investigating further, we found that similar pathological behaviors arise in other TM systems as well. Specifically, we have identified five pathologies that arise when transactions interact with locks: Blocking, Deadlock, Livelock, Early Release, and Invisible Locking. These pathologies occur across the spectrum of TM system designs, including both hardware and software systems. Table 1 summarizes where the pathologies occur and shows which proposed TM systems they affect. We describe the pathologies in detail in Section 3. These problems prevent current TM systems from interacting with locks.

Driven by these pathologies we have designed and implemented *transaction-safe locks* (*TxLocks*) that interact gracefully with transactions and eliminate the pathologies. The TxLocks' implementation entails modifications to the existing lock code in OpenSolaris' C Library and extensions to the conflict resolution policy of the TM system in use, in our case LogTM-SE [21]. In contrast to other proposals integrating locks and transactions [16], TxLocks do not depend on hardware support and work for a variety of TM systems. The design and implementation of TxLocks is presented in Section 4.

We measure the additional cost of our TxLocks' implementation over traditional locks in Section 5 and finally draw conclusions in Section 6.

2. Background

In this section we provide background material on transactional memory and locking that is critical to understanding their interaction.

2.1 Transactional Memory

A transactional memory (TM) system allows the programmer to mark code regions as transactions. The TM system is responsible for ensuring that the code executes atomically (to completion) and in isolation (without intermediate state visible to others). TM systems, both hardware and software, can be characterized along four dimensions: *version management*, *conflict detection*, *conflict resolution* and *atomicity strength*.

Version management handles the simultaneous storage of both newly written values (for commit) and old values (for abort). *Eager version management* stores new values in place and old values elsewhere (e.g., a software log). *Lazy version management* leaves old values in place and stores new values elsewhere.

Conflict detection checks for conflicts between concurrent transactions. *Eager conflict detection* checks for conflicts on each memory request, while *lazy conflict detection* defers the check until commit. Eager version management requires eager conflict detection to prevent other transactions from reading intermediate states.

Conflict resolution takes action when a conflict is detected. For eager conflict detection, resolution centers on the requester, while for lazy conflict detection it centers on the committer. The resolution policy can stall the requester (committer), abort the requester (committer), or abort the others. The choice of policy can have a large impact on performance [4, 18].

Atomicity strength reasons about the relationship between transactions and non-transactional code. *Strongly atomic* systems execute transactions atomically with respect to both other transactions *and non-transactional code*. In essence, these systems implicitly treat each instruction appearing outside a transaction as a singleton transaction. *Weakly atomic* systems execute transactions atomically only with respect to other transactions, i.e. their execution may be interleaved with non-transactional code [3]. While weak atomicity leads to several programming pitfalls [19], most software transactional memory systems employ it to boost performance.

2.2 Locks

Locks provide mutual exclusion semantics for synchronized access to shared data. A thread that attempts but fails to acquire a lock has two options: *spin* or *block*.

With *spin locks*, the thread loops reading and writing the lock variable with atomic instructions such as `Compare-and-Swap` or `Test-and-Set`. Spinning allows fast acquisi-

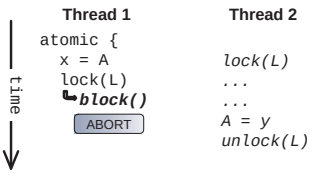

Figure 2. A case of the *blocking* pathology where transactional Thread 1 is aborted instead of blocking on lock L. It spins, retrying continuously instead of waiting in the kernel.

tion of the lock when it is released but it consumes processor cycles while busy waiting.

Blocking locks place a thread on a sleep queue while the lock is held and wake it when the lock is released. The thread sleeps in the kernel while waiting, which allows other threads to use the processor. The queue can be implemented in the kernel or in user space. Blocking has the benefit of freeing the processor for other work but requires a system call and context switches to sleep and wake up.

Adaptive locks combine the benefits of the two schemes by first spinning for a while and reverting to blocking when that fails. This can occur after a fixed number of tries, when the spinning thread is preempted, or when the thread holding the lock is suspended.

3. Classification of Pathological Behavior

When transactional code acquires a lock, the interaction of the transaction with non-transactional code contending for the same lock can lead to abnormal execution behaviors, which we call *pathologies*. We consider these execution behaviors as abnormal because they result from the execution of otherwise correct code that makes proper use of locks. We have identified five behavior pathologies where the TM system's behavior may lead to deadlock, livelock, or loss of mutual exclusion. This set has been useful in understanding TM systems but may not be exhaustive.

We now discuss each pathology in detail, describe which systems they affect, and describe with sample code the scenario under which they arise.

3.1 Blocking

Systems affected

Version Management	Any
Conflict Detection	Any
Conflict Resolution	Any
Atomicity	Strong

Description With blocking and adaptive locks a transaction may eventually block-wait in the kernel after failing to acquire a lock. On any system, this requires that transactions can be *virtualized* to survive context switching [15, 20]. If the TM system does not support suspending threads in a transaction and instead aborts the transaction, then the lock becomes a spin lock. After aborting, the transaction will continuously re-execute up to the point of acquiring the lock rather than relinquish the CPU. Even if the TM system backs off by stalling before retrying a transaction, other threads are still prevented from running on that processor.

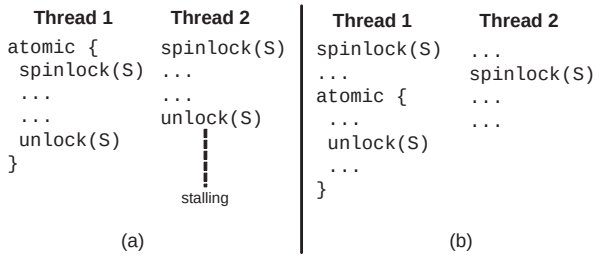

Figure 3. A case of Deadlock in an eager conflict detection and eager version management strongly atomic system (a) and a case of Livelock in a lazy conflict detection and lazy version management strongly atomic system (b).

If the TM system supports virtualization, as in software TM systems (STMs), then a problem may arise if the blocked transaction must be aborted to resolve a deadlock. Existing TM systems may not be able to abort suspended threads safely.

Figure 2 shows an example program that experiences the blocking pathology. In this example, transactional Thread 1 fails to acquire lock L, which is already acquired by the non-transactional Thread 2. With a blocking lock, Thread 1 will try to wait in the kernel. However, in this example the TM system does not support virtualization and instead aborts the transaction, causing it to restart and immediately try to acquire the lock.

3.2 Deadlock

Systems affected

Version Management	Eager
Conflict Detection	Eager
Conflict Resolution	Requester Stalls
Atomicity	Strong

Description In some TM systems, merely *attempting* to acquire a lock within a transaction can prevent another thread from releasing the lock and lead to deadlock. TM systems with eager version management and strong atomicity must stall non-transactional threads until conflicting transactions commit or abort to prevent them from reading intermediate states of the transaction. This policy can lead to a deadlock when a transactional and a non-transactional thread concurrently access a lock. This deadlock occurs both on the lock variable itself and when a non-transactional thread conflicts with any variable that a waiting transaction previously accessed.

Figure 3(a) illustrates such a deadlock. Non-transactional Thread 2 owns lock S when transactional Thread 1 tries but fails to acquire it. Although the lock acquisition fails,

the transaction still brings the memory location of the lock into its read and/or write set because of the memory access done by the atomic operation.[1] Thus, Thread 1 prevents subsequent access by any other thread, including access to release the lock, until its transaction commits or aborts. This effectively results in two owners holding the lock at different levels of abstraction. The non-transactional thread logically owns the lock while the transactional thread owns the memory containing the lock variable. These two owners prevent each other from modifying the lock's value, thereby preventing progress and leading to deadlock. Furthermore, this deadlock is undetectable by TM systems that detect only memory-level dependencies.

3.3 Livelock

Systems affected

Version Management	Any
Conflict Detection	Any
Conflict Resolution	Committer Wins
Atomicity	Strong

Description In some TM systems, spinning by a non-transactional thread can prevent a transactional thread from releasing a lock and lead to livelock. Systems using committer-wins conflict resolution and strong atomicity, so that non-transactional code always causes conflicting transactions to abort, can have this pathology.

Figure 3(b) presents sample code that experiences this pathology. Transactional Thread 1 acquires and releases lock S in separate transactions. However, when non-transactional Thread 2 spins trying to acquire lock S, it repeatedly causes the transaction that should release the lock to abort. The transaction continuously attempts to release the lock but is always aborted by the non-transactional thread's spinning before it can commit. This leads to livelock that will only be resolved when Thread 2 is eventually preempted. Note that while locks may be released in any order without ill effects, but reordering lock releases and transaction commits cause a pathology.

3.4 Early Release

Systems affected

Version Management	Eager
Conflict Detection	Any
Conflict Resolution	Any
Atomicity	Weak

Description Weakly atomic systems with eager version management allow non-transactional code to access uncommitted data. These systems may lose mutual exclusion when a transaction aborts after releasing a lock. Three problems may arise. First, the TM system will restore old values to data protected by the lock without waiting to reacquire the lock, corrupting the data if another thread has since acquired the lock. Second, the TM system may restore the lock to its unlocked state, even though it is owned by another thread. Third, if the lock was acquired before beginning the transaction, the transaction will restart assuming it owns the lock, even if another thread has since acquired it. In all cases, the system violates the mutual exclusion provided by the lock.

[1] *Test-And-Set* always writes a memory location bringing it into the transaction's write set. *Compare-And-Swap* always reads a memory location bringing it into the transaction's read set and just brings the memory location into the write set when it writes it on success.

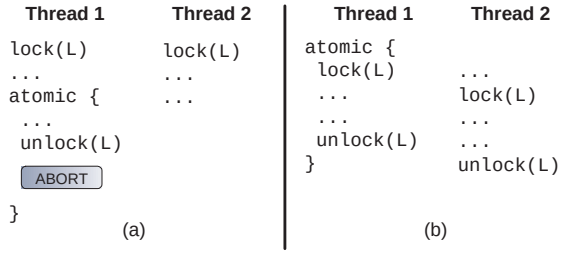

Figure 4. (a) A case of Early Release in weakly atomic eager version management system and (b) a case of Invisible Locking in weakly atomic lazy version management system. Both pathological cases break mutual exclusion enforced by lock L.

Figure 4(a) shows sample code that is affected by this pathology. Thread 1's release of lock L in a transaction is immediately visible to other threads. As a result non-transactional Thread 2 can acquire the lock before the transaction commits. This operation is unsafe if the transaction aborts and restarts. In such a case both threads incorrectly think that they exclusively have the lock and consequently the mutual exclusion semantics provided by the lock are lost.

3.5 Invisible Locking

Systems affected

Version Management	Lazy
Conflict Detection	Any
Conflict Resolution	Any
Atomicity	Weak

Description Systems with weak atomicity that buffer writes until commit may acquire locks invisibly: the acquisition of the lock and update of data are buffered until the transaction commits. As a result, concurrent non-transactional threads may also acquire the lock, unaware that it is held by a transaction, and access the data it protects. If the transaction commits while another thread holds the lock, the transaction will both update the data and potentially release the lock, violating the mutual exclusion provided by the lock.

Figure 4(b) shows sample code that experiences this pathology. Thread 1 acquires lock L inside a transaction. Because of lazy version management, the acquisition is invisible until the transaction commits. Non-transactional Thread 2 finds the lock free and successfully acquires it without con-

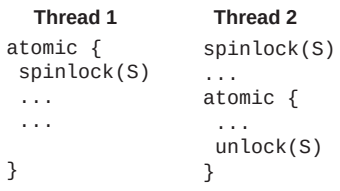

Figure 5. A case that can either deadlock or livelock depending on the conflict resolution used.

flicting because of weak atomicity. Both threads believe that they exclusively have the lock and get into the critical section protected with lock L, violating mutual exclusion.

3.6 Discussion

The Deadlock and Livelock pathologies are dual to each other in the sense that solving the one pathology may give rise to the other. Deadlock appears when the conflict resolution policy blocks requestors in favor of the owner of a transactional line. However, if the conflict resolution policy instead aborts the owner then the system can get into a livelock. Figure 5 illustrates sample code that can either deadlock or livelock depending on the conflict resolution used. Stalling the unlock request results in deadlock but aborting the transaction that encloses the unlock operation results in livelock.

Weak atomicity raises problems with locks because the additional semantics of locking are not preserved by the TM system: they are not properly acquired or released during commit or abort. This leads to the Early Release and Invisible Locking pathologies. Prior study of weak atomicity anomalies [19] may be sufficient to explain why code that accesses a lock both within and outside a transaction is problematic. However, we find value in identifying these two pathologies because it enables construction of locks (described in the following section) that allow transactions to syncronize access to data shared with non-transactional code.

Strong atomicity leads to pathologies when the dependence of one thread on another through a lock is not visible to the TM system. Thus, the TM system may stall or abort the thread holding the lock, blocking forward progress.

4. Transaction-Safe Locks

In this section we propose *transaction safe locks* (TxLocks), which cooperate with the transactional memory system to solve the pathologies discussed in Section 3. Our lock design is driven by two practical goals:

1. *Fast-Path*: The overhead of TxLocks on the common-case uncontended path should be minimal.
2. *Starvation-Freedom*: When locks and transactions conflict, all threads should eventually make progress.

The first goal requires that little additional code, and no additional synchronization, is encountered by non-transactional threads. The second goal requires that conflict resolution eventually allows all threads to make progress.

We first present an abstract design of TxLocks that may be applied to locks in any system. Later, in Section 4.2, we describe in detail an implementation of TxLocks for LogTM-SE system running on Solaris.

4.1 Design

Transaction-safe locks extend locks in four directions to prevent pathologies from occurring. The design makes three demands from the TM system.

1. A transaction must be able to specify code to execute when it commits (*commit actions*) and aborts (*compensating actions*) [12, 14, 22]. These actions enable locks to defer locking operations until commit and perform undo operations on abort.
2. A transaction must be able to *escape* into non-transactional code without terminating the transaction [14, 22]. These escape actions allow modifications of the lock variable outside transactional control so that they are immediately visible to all threads.
3. It must be possible to modify the conflict resolution policy to perform additional tasks when a transaction conflicts with a lock.

We next present the design of TxLocks by separately describing the four design elements and how they fix the pathologies. We separate design elements by the pathology they prevent, so a system that does not suffer from all pathologies does not need to implement the entire design.

4.1.1 Non-transactional Lock Operations

In order to solve the Invisible Locking pathology, we use escape actions to remove atomic operations that modify the lock variable from transactional control. Memory accessed inside an escape action is not added to the transaction's read and write sets, so modifications of the lock variable are immediately visible to the other threads. Because the lock is modified outside transaction control, the change must be explicitly reversed. The lock implementation registers a compensating action to release the lock if the transaction aborts.

This change also prevents the Livelock and some cases of the Deadlock pathology. Previously, a transaction that reads a lock variable could prevent another thread from unlocking a lock, leading to deadlock or livelock. If the transaction accesses the lock variable only in an escape action then it no longer prevents other threads from releasing the lock.

4.1.2 Deferred Unlock

A side effect of removing locks from transactional control is that strongly atomic systems become vulnerable to

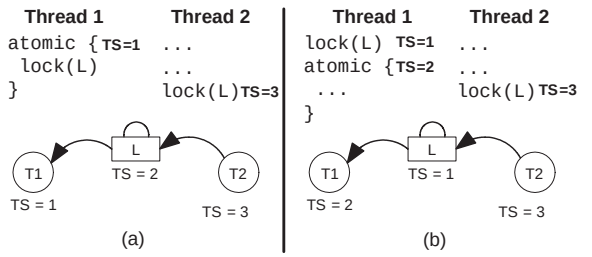

Figure 6. In (a) the thread acquires the lock inside a transaction so the logical dependency can be broken by aborting the transaction. In (b), however, the thread acquires the lock outside the transaction, so abort does not break the logical dependency.

the Early Release pathology, which previously impacted only weakly atomic systems. To prevent this pathology, the TxLocks implementation does not release a lock immediately when called within a transaction. Instead, the unlock code registers a commit action to release the lock after the transaction commits. This ensures that the lock is not released until there is no possibility of rolling back changes to the data it protects. While other threads cannot acquire the lock before the transaction commits, TxLocks allow the transaction to reacquire the lock if needed to avoid deadlocking with itself. Note that deferring release may decrease concurrency because locks are held until the transaction commits.

4.1.3 Lock-aware Conflict Resolution

The conflict resolution policies implemented in TM systems, such as committer-wins and requester-stalls, do not consider the effects of locks. As a result, they may select the wrong victim (the transaction to stall or abort) and cause the Deadlock and Livelock pathologies. We assume a software *contention manager* exists that can implement sophisticated policies, such as deadlock detection.

To prevent these two pathologies, TxLocks provide additional information to the contention manager when locks and transactions conflict. Specifically, locks provide information about when the lock was acquired and which thread owns the lock. Information about the lock owner is necessary to build a dependency graph that includes both transactions and locks.

In addition, both locks and transactions have timestamps indicating when they were acquired. This enables a contention manager to identify whether a lock is nested in a transaction or a transaction is nested in a lock. Differentiating between these two cases is important because aborting a transaction will release only the locks that are nested in that transaction. If the lock timestamp is newer than the transaction's timestamp, the transaction had begun before the lock was acquired and aborting the transaction will release the lock. However, if the lock is earlier, then aborting the transaction will not release the lock.

Figure 6 illustrates the two cases. On the left, the timestamp of the lock owner (Thread 1) is older than the lock's timestamp. Thus, aborting Thread 1's transaction will release the lock. When the lock timestamp is older, shown on the right, aborting the transaction will not release the lock and resolve a deadlock.

In addition, the contention manager for TxLocks uses both transactional dependencies (provided by the TM system) and lock dependencies (provided by the lock owner fields) to detect deadlock that arise due to interactions of locks and transactions. When such a deadlock occurs, the contention manager aborts the youngest transaction that can break the deadlock.

4.1.4 Block/Wake-up Protocols

The Blocking pathology occurs for two reasons. If the TM system does not support suspending transactional threads, blocking leads to spinning. If the TM system can suspend transactional threads, problems occur if it cannot abort waiting threads. In both case, the TM system must cooperate with the lock code through a block/wake-up protocol to achieve a graceful wake-up of the blocked transaction in the case of abort or when the lock can be acquired. We have identified two possible protocols: *Abort/Block* and *Block/Abort*.

For TM systems that cannot suspend a thread within a transaction, the *Abort/Block* protocol aborts the transaction and then suspends the thread, placing it on the lock's sleep queue. When the lock is available, the lock code wakes up the thread and restarts the transaction. The thread re-executes the transaction and hopefully acquires the lock without blocking.

While this protocol seems feasible, it suffers from several drawbacks:

- If the transaction acquires multiple locks, it may never acquire all locks without blocking, leading to starvation.
- If the transaction follows a different path upon restarting, it may need a different lock. This could lead to starvation.
- Under contention, the transaction may starve because it does not acquire the lock immediately after resuming; it must first execute from the start of the transaction up to the call to acquire the lock.

However, this protocol may be the best option for systems that cannot suspend threads in a transaction.

An alternative is the *Block/Abort* protocol in which transactions block on a lock and only abort on requests from the contention manager. The transaction must notify the TM system of which lock it is waiting for before it blocks. This allows the contention manager to resolve deadlocks that arise due to interactions of locks and transactions. When a blocked thread resumes, it checks whether it was aborted, and if so removes itself from the lock's sleep queue, waking

the next waiting thread. Otherwise the thread acquires the lock and continues with the transaction.

Both protocols allow a transaction to safely acquire a blocking lock without devolving to spinning. TxLocks implement the Block/Abort protocol because our platform supports suspending transactions.

4.2 Implementation in OpenSolaris

We implemented a working version of TxLocks by extending the adaptive mutex lock implementation of OpenSolaris' C Library with the four components presented in Section 4.1. We chose the adaptive mutex lock since this is the one most often used in our workloads. As a base TM system we use LogTM-SE, which is an HTM providing strong atomicity, using eager version management, eager conflict detection, and requester-stalls conflict resolution. We have chosen LogTM-SE over the original LogTM system because LogTM-SE supports virtualizing transactions and compensating/commit actions.

LogTM-SE records a timestamp in hardware when a transaction begins. This timestamp, taken from a loosely-synchronized cycle counter such as the `stick` (system tick) in SPARC systems, is sent on all coherence requests and is used for conservative deadlock detection: transactions stalled on a conflict that receive a conflicting request from an older transaction detect a possible deadlock and invoke the contention manager.

Figure 7 shows pseudocode for top-level TxLocks' `lock` and `unlock` functions. Shaded areas highlight addition of new code to standard OpenSolaris adaptive mutexes. The lock routines have four sets of additions, corresponding to the four design elements with each set marked in the figure using one of the lowercase letters *a*-*d*.

First, all lock code executes as an escape action, outside of transactional control (set a: lines 2, 13, 24, 27, 30, 39). This prevents the Invisible Locking, Livelock, and some cases of the Deadlock pathology. To ensure locks are released on abort, the lock function registers a compensating action to release the lock in this case (set a: line 21).

Second, the unlock function defers releasing the lock when called within a transaction (set b: line 34). This prevents the Early Release pathology by ensuring that locks are held until the transaction commits. To allow the same transaction to reacquire the lock without deadlocking with itself we rely on the fact that OpenSolaris's adaptive mutexes can be acquired recursively (lines 7, 28).

Third, the lock implementation stores information about when the lock was acquired for lock-aware conflict resolution (set c: line 18). We added a `timestamp` field to the mutex structure, in which the lock code stores LogTM-SE's transaction timestamp after acquiring the lock. For non-transactional code, this is the current value of the `stick` counter. Conflict resolution also requires the lock owner, which is already stored by OpenSolaris mutex locks (line 4). With this information, the contention manager can detect

```
1.   void lock(txlock_t* mp) {
2.a    BEGIN_ESCAPE;
3.     if (set_lock_byte(&mp->txlock_lockw) == 0) {
4.       mp->mutex_owner = self;
5.       goto lock_acquired;
6.     }
7.     if (mp->mutex_owner == self) {
8.       mp->rcount++;
9.       goto lock_acquired_noTS;
10.    }
11.    if (txlock_trylock_adaptive(mp) != 0) {
12.d     if (txlock_lock_queue(self, mp) == ABORT) {
13.a       END_ESCAPE;
14.d       ABORT_TRANSACTION;
15.      }
16.    }
17.  lock_acquired:
18.c   mp->timestamp = xact_timestamp();
19.  lock_acquired_noTS:
20.    if (in_xact()) {
21.a     register_compensating_action(
22.          txlock_unlock_impl, mp);
23.    }
24.a   END_ESCAPE;
25.  }
```

```
26.  void unlock(txlock_t* mp) {
27.a   BEGIN_ESCAPE;
28.      if (mp->mutex_owner == self) {
29.        mp->rcount--;
30.a       END_ESCAPE;
31.        return;
32.      }
33.      if (in_xact()) {
34.b       register_commit_action(
35.            txlock_unlock_impl, mp);
36.      } else {
37.        txlock_unlock_impl(mp);
38.      }
39.a   END_ESCAPE;
40.  }
```

Figure 7. Pseudocode for TxLocks' lock and unlock functions. The shaded areas are new code, and the letters distinguish code changes described in section 4.2.

deadlock from the full dependency graph of threads, including both lock-based dependencies and transaction dependencies, and decide how to resolve deadlocks. This resolves the Deadlock pathology, because the contention manager can ensure that a transaction waiting for lock does not prevent the lock holder from executing.

Fourth, the lock implementation adds code to support the block/abort protocol (set d: line 12). The lock routine checks whether it was aborted while waiting, and if so exits the escape action and aborts the current transaction. Additional code for blocking is in the `txlock_lock_queue` function (not shown for brevity). Before suspending the thread, this function calls the contention manager to check for deadlocks. After returning from the kernel, this code checks to see if it has been aborted. If so, it returns ABORT to the lock function, which aborts the running transaction.

When a thread conflicts with a transaction that is blocked in the kernel waiting for a lock, it cannot resolve the conflict by stalling. Therefore, LogTM-SE traps into the con-

tention manager when a thread conflicts with a suspended transaction [21]. On such a conflict, the contention manager suspends the thread and enqueues it behind the thread with which it conflicts. When a thread conflicts with multiple (reader) transactions, the thread waits on the thread with the youngest transaction, which is likely to end its transaction last. We implement blocking behind a transaction with user-level sleep queues, similar to Zilles et al. [22].

4.3 Summary

TxLocks resolve the five locking pathologies by moving locking code outside transactions (with escape actions) and by invoking a software contention manager to detect deadlocks and resolve conflicts. In the uncontended non-transactional case, the only additional code is to store timestamp value (the calls to begin and end escape actions are treated as no-ops). In the transactional case, the transaction must register compensating and commit actions as well.

TxLocks bear a strong resemblance to cxspinlocks in TxLinux [16]. The primary difference of the two lock primitives is the programmer interface. With cxspinlocks, programmers must specify a lock (which could be a single global lock) for all critical sections, and the cxspinlock mechanism executes these regions with optimistic concurrency when possible. However, specifying a lock introduces extra coherence traffic, which is unnecessary in the case when optimistic regions do not compete with exclusive ones. In addition, cxspinlocks requires special hardware, such as the `xtest` and `xcas` instructions, making the technique only applicable to HTMs.

In contrast, TxLocks allow the programmer to program using transactions that can acquire locks if needed. Although programmers cannot achieve optimistic concurrency for the regions protected by the lock, they can reap the productivity benefits of using transactions while still interacting with lock-based code. In addition, TxLocks require no special hardware and can therefore be used in a range of TM systems, including both hardware and software systems.

5. Preliminary Performance

We are currently developing workloads that require locks and transactions to interact, and so we do not have complete results on how TxLocks perform. However, we have experimented with them in the BIND DNS server. Txlocks allow the standard C runtime memory allocator to execute correctly when called from a transaction. Prior to the development of TxLocks, locks in the allocator caused deadlock and allocator had to be pulled completely into an escape action to execute correctly.

We measure the cost of TxLocks on the LogTM-SE simulator [21], which uses Wisconsin GEMS [11]. It models a 32-processor Sparc chip-multiprocessor with a single-issue in-order pipeline and memory latencies similar to the Sun T1 (Niagara) processor [10]. This simulator is not cycle-accurate for short code sequences, but can measure approximate performance differences.

Lock	Cycles
Solaris adaptive mutex	61
TxLock non-transactional	89
TxLock transactional	185

Table 2. Cycles to acquire a lock.

As we do not have a workload to explore the full performance of TxLocks, we measure the overhead of supporting transactions in the uncontended case. The `Lock-stress` program repeatedly acquires a native OpenSolaris adaptive mutex outside a transaction, a TxLock outside a transaction, or a TxLock inside a transaction. As there is only a single thread, there is no contention. We measure the average number of cycles to acquire the lock.

Table 2 shows the results of our experiments. In the non-transactional case, TxLocks add 45% to the native lock case. This is due to the extra instructions for entering escape actions and saving a timestamp. In the transactional case, TxLocks add 300% to the native lock case. The additional time is spent registering a compensating action to release the lock on abort, which takes 79 cycles on LogTM-SE. This cost could be further optimized to reduce locking overhead.

These results demonstrate that TxLocks add little to the cost of locking for non-transactional code. The overhead for acquiring TxLocks in transactions is higher, but provides a substantial benefit by allowing transactions and lock to interact correctly.

6. Conclusion

Many existing transactional memory systems do not interact well with locks. We found five pathologies that may arise, depending on the TM system design, when transactions acquire locks: Blocking, Deadlock, Livelock, Early Release, and Invisible Locking. The pathologies under strong atomicity occur for two reasons. First, transaction conflict resolution is unaware of locks, and may abort the only transaction able to release a lock and allow further progress. Second, lock variables may be locked both at the memory level, by the TM system, and at a logical level, by the lock itself. If these two levels conflict then deadlock or livelock can arise. In weakly atomic systems, pathologies arise because the system does not respect lock semantics during abort and commit.

To address these problems, we designed TxLocks, which prevent these pathologies with four techniques: non-transactional lock operations, deferred release, lock-aware conflict resolution, and a block/abort protocol. In testing, we found that TxLocks add little performance overhead to the common case of uncontended locking yet provide the benefit of enabling transactions and locks to interact safely.

Acknowledgements

This work is supported in part by NSF grant CNS-0720565.

References

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *HPCA 11*, Feb. 2005.

[2] C. Blundell, J. Devietti, E. C. Lewis, and M. M. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *ISCA 34*, June 2007.

[3] C. Blundell, E. C. Lewis, and M. M. Martin. Deconstructing Transactional Semantics: The Subtleties of Atomicity. In *Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, June 2005.

[4] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance Pathologies in Hardware Transactional Memory. *IEEE Micro*, 28(1), Jan/Feb 2008.

[5] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *ISCA 33*, June 2006.

[6] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, Sept. 2006.

[7] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[8] Internet Systems Consortium. BIND (Berkeley Internet Name Domain). http://www.isc.org.

[9] T. Jinmei and P. Vixie. Implementation and Evaluation of Moderate Parallelism in the BIND9 DNS Server. In *Proceedings of the 2006 Usenix Annual Technical Conference*, June 2006.

[10] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, Mar/Apr 2005.

[11] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, pages 92–99, Sept. 2005.

[12] A. McDonald, J. Chung, B. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural Semantics for Practical Transactional Memory. In *ISCA 33*, June 2006.

[13] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-Based Transactional Memory. In *HPCA 12*, Feb. 2006.

[14] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting Nested Transactional Memory in LogTM. In *ASPLOS 12*, Oct. 2006.

[15] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *ISCA 32*, June 2005.

[16] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, A. Bhandari, and E. Witchel. TxLinux: Using and Managing Hardware Transactional Memory in an Operating System. In *SOSP 21*, Oct. 2007.

[17] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: a High Performance Software Transactional Memory System for a Multi-Core Runtime. In *PPOPP 13*, Mar. 2006.

[18] W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *PODC 24*, July 2005.

[19] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing Isolation and Ordering in STM. In *Proceedings of the SIGPLAN 2007 Conference on Programming Language Design and Implementation*, June 2007.

[20] M. M. Swift, H. Volos, N. Goyal, L. Yen, M. D. Hill, and D. A. Wood. OS Support for Virtualizing Hardware Transactional Memory. In *TRANSACT 3*, Feb. 2008.

[21] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *HPCA 13*, Feb. 2007.

[22] C. Zilles and L. Baugh. Extending Hardware Transactional Memory to Support Non-busy Waiting and Non-transactional Actions. In *TRANSACT 1*, June 2006.