Applying Transactional Memory to Concurrency Bugs

Haris Volos[1], Andres Jaan Tack[2] *, Michael M. Swift[1], Shan Lu[1]

[1] Computer Sciences Department, University of Wisconsin–Madison
[2] Skype Limited

[1] {hvolos, swift, shanlu}@cs.wisc.edu, [2] andres.jaan.tack@skype.net

Abstract

Multithreaded programs often suffer from synchronization bugs such as atomicity violations and deadlocks. These bugs arise from complicated locking strategies and ad hoc synchronization methods to avoid the use of locks. A survey of the bug databases of major open-source applications shows that concurrency bugs often take multiple fix attempts, and that fixes often introduce yet more concurrency bugs. Transactional memory (TM) enables programmers to declare regions of code atomic without specifying a lock and has the potential to avoid these bugs.

Where most previous studies have focused on using TM to write new programs from scratch, we consider its utility in fixing existing programs with concurrency bugs. We therefore investigate four methods of using TM on three concurrent programs. Overall, we find that 29% of the bugs are not fixable by transactional memory, showing that TM does not address many important types of concurrency bugs. In particular, TM works poorly with extremely long critical sections and with deadlocks involving both condition variables and I/O. Conversely, we find that for 56% of the bugs, transactional memory offers demonstrable value by simplifying the reasoning behind a fix or the effort to implement a fix, and using transactions in the first place would have avoided 71% of the bugs examined. We also find that ad hoc synchronization put in place to avoid the overhead of locking can be greatly simplified with TM, but requires hardware support to perform well.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Languages, Reliability

Keywords Transactional memory, concurrent program, concurrency bug, debugging, atomicity violation, deadlock

1. Introduction

Transactional memory (TM) promises to simplify concurrent programming by reducing the burden on programmers and to improve performance or scalability through increased concurrency [21]. The simplicity benefit can be achieved by writing new programs using transactions [46] or by rewriting existing programs to use transactions [60]. However, this benefit could also be realized by applying transaction memory to problematic code in existing programs without a complete rewrite. This preserves the investment in existing code while potentially simplifying complex synchronization code.

Applying transactional memory to concurrency bugs raises three challenges. First, it focuses the effort on especially difficult problems and thus stresses TM's ability to express synchronization requirements. Second, it forces us to consider *concise* code changes, because we want to minimize the amount of code that must be rewritten throughout the program. Finally, it stresses the *integration* with existing code to avoid rewriting major portions of the program. Fortunately, extensions to TM allowing the use of locks [45, 53], condition variables [17, 22], and select system calls [39, 54] simplify this task by allowing most existing code to be moved into transactions.

We evaluate the utility of TM in complex concurrent programs by applying it as a fix for previously found concurrency bugs in three programs: the Mozilla web browser, the Apache httpd web server, and the MySQL database [30]. We explore four different approaches to using transactional memory in existing concurrent programs. To be as unobtrusive as possible, these four methods tailor the use of TM to existing lock-based synchronization code so they do not necessarily reflect how a programmer would use TM to write new programs. Two simple approaches place all conflicting code in transactions. While useful, we find this sometimes requires widespread code modifications. Two other, more sophisticated approaches compose transactions with locks and leverage the rollback mechanism of transactional memory to preempt locks causing a deadlock.

We find that transactional memory is not useful in 17 of 60 atomicity violations and deadlock bugs. This surprising result arises because many concurrency bugs are not about shared data; rather they concern synchronization, such as condition variables, or I/O, such as file or network access. However, we do find that straightforward uses of TM can fix 40 of the 60 bugs, and sophisticated uses of TM can fix 3 additional bugs and simplify the fixes of 20 of the 40 bugs. We judge that 34 out of the 43 TM-based fixes are simpler and preferable to developers' fixes. These results demonstrate that transactional memory, as proposed, is moderately useful in concurrent programs, but that it does not ameliorate enough predicaments to be a panacea. We also find that ad hoc synchronization, such as ownership flags put in place to avoid the overhead of locking, can be greatly simplified with TM, but requires hardware support to perform well enough.

2. Related Work

As the count of cores in commodity computers rises, researchers have devoted more effort to the problem of writing correct multithreaded code. Our work builds on past work investigating the

* Work done while a student at the University of Wisconsin – Madison

ASPLOS'12, March 3–7, 2012, London, England, UK.

usefulness of transactional memory, integrating transactions into existing code, and fixing concurrency bugs.

Usefulness of TM Our work explores the utility of TM as a useful tool for fixing concurrency bugs. Several recent studies have sought to quantify and explain the benefits of programming with transactions as compared to locks [37, 46]. These studies found that TM is measurably easier than fine-grained locks, and results in fewer bugs, less development time, or better performance. Compared to our work, these studies use TM for relatively simple programs with a small number of different thread types accessing a small number of data structures, and do not require synchronization around I/O. Thus they may not have encountered similarly difficult synchronization problems. Other work has questioned the utility of software TM due to its poor performance [6], despite work showing it can improve performance of existing multithreaded applications [54]. In contrast, our work targets transactional memory at solving complex synchronization problems in existing code rather than exploring the benefits of developing an entire program with transactions. Furthermore, we observe that large programs use many different synchronization mechanisms, while these studies focused on using just a single mechanism (locks or transactions).

Most similar to our work is the Atomic Quake project [60], which rewrote a game engine to use transactions. The authors sought to use transactions everywhere, and redesigned the application to better fit transactions. They found, similar to our results, that conditional synchronization remains a problem for transactions. In contrast to Atomic Quake, we use transactions as one of many synchronization tools, rather than applying it as the dominant synchronization method.

Transactional Memory Mechanisms Our work depends on high-quality transactional memory systems and seeks to apply them to existing code bases. There are many proposed and developed transactional memory mechanisms in hardware [14, 20, 34, 43] and software [12, 15, 16, 19, 21, 48]. IBM has integrated transactional memory into their Blue Gene/Q supercomputer processor [1], AMD has indicated it may implement limited transactional memory systems in hardware [3], and GCC provides extensions for using transactional memory [23]. Thus, transactional memory is reaching a maturity level where it can be used in deployed applications.

Access to non-memory resources is difficult for transactional memory systems. Past work on executing system calls within transactions has shown that system calls occur regularly within lock-based critical sections [4, 51]. Proposed mechanisms to handle system calls include making transactions inevitable [5, 50, 56] or executing them transactionally with library support [54] or with kernel transactions [39]. Our work leverages these mechanisms and shows that supporting system calls within transactions is useful even for the specific task of fixing concurrency bugs.

TM-enabled locks Our work relies on using transactions and locks cooperatively. Due to the difficulty of providing transactional semantics for system calls or I/O and the low performance of software TM systems, prior work such as TxLinux propose locks accelerated by transactions that provide the programmability and low overhead of coarse-grained locks but the concurrency of transactions [44, 45, 52]. In contrast, we seek the programmability of transactions more than their concurrency. However, transactions cannot normally coexist with the locks found in existing code without modifying the implementation of locks [53]. Fortunately, proposed transactional-memory semantics precisely define the interaction of locks and transactions and enable interoperability [2, 11, 49]. In this paper, we leverage the TxLock [53] design, which defers releasing locks until a transaction commits and automatically releases locks on abort.

Concurrency bugs While we apply transactions as a possible solution to concurrency bugs, other research has looked at automated methods to generate locking code [25] or to dynamically avoid concurrency bugs through scheduling [8, 8, 26, 31, 40, 41]. Our work focuses on the utility of transactional memory to express hard synchronization requirements, while those projects focus more specifically on fixing bugs, perhaps temporarily until a developer creates a permanent patch.

3. Applying TM to Concurrency Bugs

The goal of our work is to investigate whether transactional memory's simple interface can address challenging synchronization problems in concurrent programs. To that end, we apply TM to *concurrency bugs*, where developers either left out synchronization statements or coded them incorrectly. We do not attempt to rewrite the whole program using transactional memory for two reasons: the current performance of software TM is too low for some uses, and not all system calls can be handled transactionally. For example, state-of-the art TM systems fall back on a single global lock for difficult I/O operations, and cannot handle two-way communication.

Instead, we look at transactional memory as a tool that can be useful in solving difficult synchronization problems. Thus, our approach is to look for solutions that require a minimum of code changes and allow transactions to co-exist with existing lock-based synchronization code. To that extent, our approach to applying transactional memory to concurrency bugs does not necessarily reflect how a programmer would use transactional memory to write new programs.

3.1 Concurrency bugs

We focus on two classes of concurrency bugs that transactional memory was designed to address. In a *deadlock* (DL) the order of lock operations may lead to circular wait between threads. Deadlocks can also occur with condition variables, for example when waiting with a lock held. TM addresses deadlock by (i) removing the use of multiple locks that enable deadlock, and (ii) automatically aborting one or more waiting threads. In an *atomicity violation* (AV), code is not protected from interleaving with other threads accessing the same shared data. TM addresses atomicity violations by making it simple to declare atomicity without having to select the right lock: TM detects conflicts with all other threads, not just those holding the same locks. TM alone does not address *ordering violations*, which occur when the program requires that events in two threads happen in a certain order but does not enforce the ordering, so we do not consider these bugs.

Writing correctly synchronized code can be challenging: it may require the addition of synchronization code throughout a program when a variable becomes shared. More importantly, it requires *non-local reasoning* to understand how introducing new synchronization code in one location can affect code elsewhere. For example, adding a new lock requires considering whether it can introduce deadlock with all existing locks.

The difficulty in fixing a bug comes from the *conceptual effort* of creating the patch, which is how difficult it is to find a solution, and the *implementation effort*, the amount of code that must be changed, both of which we evaluate with transactional memory. Due to the non-local nature of concurrency bugs, the conceptual effort can be harder than the implementation effort, as it requires finding all the code regions that have contributed to the bug, and making sure that the new fix will not introduce new correctness problems (*e.g.*, no deadlock, no double-acquire of locks). In contrast, the implementation effort is largely mechanical and relates to how many code locations must change. Ideally, transactional memory will prove to have less conceptual effort, because of its simple

interface and strong semantics, as well as less implementation effort, because less code needs to be changed.

Indeed, evidence suggests that concurrency bug fixes frequently take a long time to produce and often fail to fix the problem or introduce new bugs as a result of the above challenges. Consider atomicity violations, where the logic of the fix is straightforward: hold a lock while executing critical regions of code. In studying concurrency bug fixes, we have seen that the reality of fixing these bugs is far more complex:

- In Mozilla, programmers used the wrong lock to fix an atomicity violation (Mozilla#18025), which was not discovered until diagnosing another atomicity violation four years later (Mozilla#133773).
- In an Apache bug, although the buggy code involved only one function, developers had to make code changes in another two places to declare new lock variables and initialize these locks (Apache#25520).
- Performance concerns further complicate the fixing process in a MySQL bug: programmers implemented their own conflict checking, abort, rollback, and re-execution mechanisms to fix an atomicity violation bug without using locks (MySQL#16582).

Deadlock bugs lead to similar problems. Their fixes frequently require re-ordering, adding and removing synchronization code.

- In several cases, fixing one deadlock bug introduced another deadlock bug, taking months or even a year to completely fix the problem (Mozilla#54743,#79054, #60303,#90994).
- In a Mozilla deadlock bug, the developers were so frustrated that they intentionally introduced a data-race bug in order to fix the original deadlock bug (Mozilla#123930).

These experiences demonstrate that writing correctly synchronized concurrent code is hard, and easier coding techniques would be welcomed.

3.2 Transactional Memory

Transactional memory allows a programmer to declare a block of code as an *atomic region* with the `atomic` keyword, and the TM system ensures that each atomic region executes to completion or not at all (atomicity), and that intermediate states of memory are not visible to other atomic regions (isolation). This simple interface aims to ease concurrent programming as compared to alternatives by helping the programmer to avoid correctness pitfalls.

A speculation-based implementation of atomic regions allows multiple regions to execute concurrently through memory transactions. [1] The implementation must detect and resolve conflicts between concurrent atomic regions by aborting and re-executing one of the transactions involved.

Both software and hardware implementations have been proposed. Software TM implementations instrument code to record the locations read and written and detect conflicts either at commit or while the transaction executes. As a result, software TM implementations may slow down critical sections by 3-5x [12]. Proposed hardware TM implementations, in contrast, perform these operations in hardware with less slowdown to critical-section code [20, 34]. However, feasible hardware TM implementations often bound the number of distinct memory locations that can be accessed within a transaction. Thus, they are best paired with a software TM for fallback when transactions exceed hardware limits [10, 13, 29].

[1] In this paper, we use the term *atomic region* to refer to the language level construct and the term *memory transaction* to refer to the implementation of an atomic region with a speculation-based transactional memory system.

4. Bug-Fix Methodology

We develop a set of different uses of transactional memory to solve hard concurrency problems. We present our mechanisms for fixing concurrency bugs as a set of recipes: the *ingredients* are the underlying mechanisms provided by transactional memory, while *recipes* describe how to combine the mechanisms to fix specific classes of bugs.

4.1 Transactional Memory Ingredients

Our fix recipes rely on four mechanisms, each of which we describe below. We describe possible implementations below, and defer discussion of specific mechanisms used in experiments to Section 5.1. Not all proposals for transactional memory support all these mechanisms, and this study demonstrates the added value of such support.

Atomic regions allow a programmer to declare a region of code `atomic`, and the underlying implementation ensures that it executes atomically in isolation. Thus, other atomic regions cannot view updates performed by the code region until it has executed entirely and this region cannot see changes made by other regions during its own execution. Atomic regions may be implemented through either memory transactions or locks. All transactional memory systems support atomic regions, although they may be limited in size and complexity for some hardware proposals, for example to the size of the write buffer, cache or a fixed number of cache lines [9, 42]. Lock implementations have unlimited size, and use either a global lock or perform lock inference, which (semi-)automatically assigns distinct fine-grain locks to atomic regions [7, 18, 33]. Locks remove the need for rollback and thus can easily support I/O operations.

Explicit rollback leverages the ability of transactional memory implementations to rollback partially executed transactions. It can be used for retry-style synchronization [22], akin to condition variables: a thread aborts the current transaction and suspends until a variable the transaction read has changed, at which point the thread retries the transaction. This allows a thread to wait until a condition is satisfied as a result of another thread's actions. Not all TM mechanisms support rollback, as atomic regions implemented with locks cannot roll back.

In addition to these two basic mechanisms, which have been explored in the past, we also identify two additional mechanisms that use transactions and locks cooperatively to simplify synchronization.

Preemptible resources can safely be acquired inside a memory transaction and automatically released if the transaction aborts. With this mechanism, the no-preemption requirement for deadlock can be removed if one thread is in a transaction and aborts and releases contested resources. Several proposed designs allow standard locks to be acquired and preempted within a transaction [27, 45, 53]. Reversible I/O operations extend the set of operations that can take place within a transaction to include system calls. Transactional system interfaces such as TxOS [39] or xCalls [54] may be used to enable reversible I/O. Unfortunately, some forms of I/O, such as two-way communication, cannot be made reversible with these techniques.

Atomic/lock serialization properly synchronizes accesses protected under an atomic region with accesses protected under a lock so that code protected using locks cannot see intermediate state of an atomic section that has not yet completed, and vice-versa. The *cxspinlock* construct from TxLinux [45] and speculative lock elision using transactions [38, 42, 47] provide this by beginning transactions in all critical sections. Another possible implementation is to grab a global reader/writer lock in shared mode when acquiring a lock, and exclusively when executing a transaction.

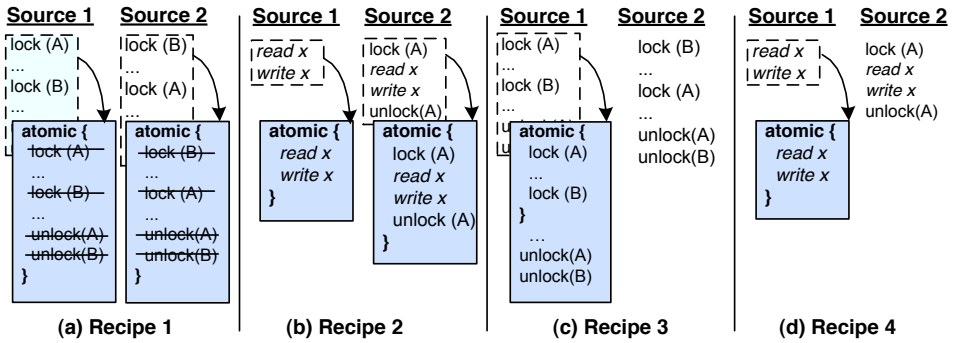

Figure 1. **Example uses of our fix recipes.**

This construct ensures that code synchronized with transactions correctly interoperates with code using locks, and therefore allows any atomicity violation to be fixed by placing the relevant code in a transaction, whether or not other code uses locks for atomicity. We stress that we evaluate this mechanism to see whether it is useful, not because it is efficient in software.

4.2 Recipes

We use two *straightforward* fix methods (recipes 1 and 2) that rely primarily on atomic regions and have broad applicability. In addition, we describe two *sophisticated* methods (recipes 3 and 4) that use additional ingredients beyond atomic regions, have limited applicability but require less effort to apply.

4.3 Simple Approaches

Recipe 1: Replacement of Deadlock-prone Locks

Remove all locks contributing to a deadlock and insert atomic regions in their places. Similarly to lock coarsening, this fixes deadlock bugs by preventing circular wait .

This approach can solve a large set of deadlocks and simplify complex locking protocols. When deadlock would occur, the transactional memory system either prevents deadlock by serializing the threads involved (similar to lock coarsening), or automatically aborts one or more transactions and allows the others to make progress. Unlike lock coarsening, a transactional solution preserves the concurrency of fine-grained locking if independent atomic regions can execute concurrently. However, the overhead of software TM may degrade performance unacceptably if transactions occur in critical-path code. In addition, this approach cannot solve deadlocks involving non-lock resources (*e.g.*, conditional variables, pipes), because transactions only protect concurrent access to memory.

Conceptually, replacing deadlock-prone locks with transactions is simple because the identity of the locks is available when a deadlock occurs. Compared to fixing the locks, the developer does not need to reason about how to change the order in which locks are acquired or whether a lock can be removed. In addition, a developer needs not reason about which code path leads to locks being acquired out of order. This fix is largely mechanical, but may be time-consuming to apply to existing code if the locks protecting data are acquired in many places.

Recipe 2: Wrap All

Wrap all conflicting code regions in atomic regions, effectively fixing atomicity violation bugs.

Atomicity violations can be fixed by placing all conflicting code regions (regions that access shared data where at least one region performs a write) in atomic regions. This ensures that the memory accesses of those regions are protected under a common synchronization mechanism. For code with completely missing synchronization all synchronization is done with transactions. When some code already uses locks, this fix can be applied by either replacing existing locks with transactions, or acquiring those locks within transactions (using a transaction-safe lock such as TxLocks [53] or cxspinlocks [45]).

Compared to locks, fixing atomicity violations with atomic regions has low conceptual effort: a programmer introduces atomic regions by placing begin/end transaction markers around a portion of code, without worrying about lock granularity for concurrency and whether the fix introduces new deadlocks. In contrast, a programmer relying on locks to solve an atomicity bug must identify the right lock to use. If such lock does not exist, the programmer must introduce a new lock, and reason where the new lock falls into the lock hierarchy to avoid deadlocks with existing locks

The effort to wrap code in atomic regions is largely mechanical, but requires identifying all code that accesses the protected data. If a lock exists and is known, the effort to use transactions is similar to using existing locks. However, if a lock does not exist, using TM is simpler, because a programmer introducing a new lock to fix the bug has to add code to manage the new lock.

4.4 Sophisticated Approaches

The preceding approaches relied on the normal application of transactional memory: place all access to a subset of shared data within transactions to reap the benefits. However, we have identified two further methods that use TM's mechanisms in different ways.

Recipe 3:(Asymmetric) Deadlock Preemption

Make at least one thread (dynamic path) involved in the deadlock preemptible by wrapping the corresponding static code regions in a single abortable atomic region (memory transaction).

This recipe fixes deadlock bugs by removing the *non-preemption* requirement for deadlock. At least one of the threads involved in the deadlock must begin a transaction before acquiring the locks involved with the deadlock. When deadlock occurs, the transaction aborts, and releases any locks it acquired before retrying. This allows at least one thread to proceed. Note that not all threads need to execute transactions: as long as one thread involved with the deadlock can abort, it can break the circular wait for all threads. Thus, this solution can be efficient, because most threads can execute unchanged with locks. Preferably, the preemptible thread should be low priority or infrequently run to minimize the performance impact of executing its critical sections within a transaction.

For this fix to work, all resources acquired transactionally must be revocable: locks must be released on abort, and I/O operations must be undone. It also requires a deadlock detector to determine when to abort the transaction. Furthermore, this recipe is unique in that transactions are used only for rollback and not isolation; locks still provide mutual exclusion between threads.

Deadlock preemption can cause livelock if the preempted thread restarts and acquires locks before the other threads finish. Exponential backoff before retry, already used for contention management in TM systems, can prevent this livelock. However, livelock can also occur if useful work performed by a transaction before the deadlock is needed to make progress, as in the work performed in a monitor before waiting on a condition variable. Thus, this recipe is not useful in such cases.

Like Recipe 1, this recipe requires identifying a deadlocked thread. Unlike Recipe 1, it also requires ensuring that all resources acquired in a transaction are revocable, making it slightly more difficult to reason about than Recipe 1, which can use inevitability mechanisms to handle complex I/O. However, this fix recipe is simple to implement, as it keeps existing locks in place and wraps them in a transaction.

Recipe 4: Wrap Unprotected

Wrap the code region intended to be executed atomically in an atomic section that is serialized with all other lock critical sections.

Recipe 2 requires a developer to wrap *all* conflicting code regions in atomic regions, even if they already use locks. This in effect duplicates any existing effort put into using locks. In contrast, this recipe approach modifies only the buggy code regions that actually contribute to the atomicity violation, and leaves unmodified the code that correctly uses locks. This method is useful for *asymmetric atomicity violations*: when most code regions properly use the intended locking discipline to correctly express their atomicity objective but some do not. While atomic regions are conceptually serialized with all other lock critical section in this recipe, atomic and lock critical sections may execute concurrently if a scalable implementation of atomic/lock serialization is available. As suggested earlier, such an implementation is possible on TM systems that support transactionally executing critical sections such as TxLinux [45] and speculative lock elision using transactions [38, 42, 47].

This approach is conceptually simpler to apply than either Recipe 2 (wrap all accesses to the shared data in transactions) or to using locks, as it requires the developer to identify only the code regions that do not correctly express their atomicity objective. Furthermore, this approach has the same implementation effort as using an existing lock, assuming it is known. Compared to Recipe 2 it effectively saves the developer from distributed code changes.

5. Effectiveness of TM on Concurrency Bugs

In this section, we evaluate our bug-fix methods in the context of previously found and fixed bugs in Mozilla, Apache httpd, and MySQL [30]. The existence of fixes allows us to compare a TM solution against the developer solution, and the log of changes allows us to gauge the complexity of the fix, such as whether it took multiple tries.

Our study has three main goals. First, we want to determine whether TM has the expressive power to solve a synchronization bug (*Can TM fix the bug?*). Second, we want to find whether TM offers value to a developer when compared against locks (*Can TM fix the bug simply?*). We answer these two questions first. Finally, we give a set of examples of showing how TM can address synchronization problems and evaluate whether the performance of the fix using TM is satisfactory (*Can TM fix the bug efficiently?*).

5.1 Recipe Ingredients: Implementation

Section 4.1 presented a high-level description of the basic mechanisms required by the fixes. We now describe the specific implementation of each mechanism used in our study. We use Intel's software transactional memory (STM) compiler and runtime framework as our main TM platform [24], as it runs on real hardware, can compile the code in many open-source projects, and automatically instruments the code placed in atomic regions with calls to the STM runtime. This greatly reduces the implementation effort needed to use transactions.

Atomic Regions. The Intel STM uses memory transactions to execute atomic regions but it reverts to a global lock [5, 50, 56], when there are calls to unsafe operations that have visible side-effects before the transaction commits, such as un-instrumented legacy code, synchronization and I/O [36]. Recent work on defining TM semantics [2, 49] captures this dual notion of execution by differentiating between two types of atomic regions: atomic transactions and relaxed transactions. Atomic transactions can only be used when there are no unsafe operations, and they appear to execute atomically in a data-race-free program. In contrast, relaxed transactions are allowed to contain unsafe operations, in which case they may appear to interleave with non-transactional operations from other threads. The implication to our recipes is that memory transactions in the form of atomic transactions may only be used when unsafe operations are replaced with their transactionally safe equivalent ones as we describe in the *preemptible resources* section below.

Explicit Rollback. The Intel STM provides an `abort` statement, which can be used to explicitly roll back a transaction. We use this statement to provide a limited version of the `retry` mechanism, by aborting and immediately retrying a deadlocking transaction. The use of this mechanism precludes the use of atomic regions that do not utilize speculation and cannot roll back. In light of the work on TM semantics, this mechanism is only safe with atomic transactions as implementations of relaxed transactions may not always utilize speculation.

Preemtible Resources. We implement two classes of preemptible resources: revocable locks and reversible I/O. We implement revocable locks with TxLocks [53]. When acquired within a transaction, these locks are held until the transaction commits and released if the transaction aborts. In addition, they detect deadlock both among locks and between locks and transactions, and will abort the transaction if deadlock occurs. We enable reversible I/O via xCalls [54]. xCalls provide a library-based implementation of transactional semantics for common system calls. The xCall library defers until commit time those system calls that can be delayed. When that is not possible, system calls are executed as part of the transaction and their side effects are reversed on abort atomically with respect to all other transactions. xCalls reverts to inevitable transactions for system calls that are not reversible and cannot be deferred. Such calls produce side effects and either have ambiguous/variable semantics (*e.g.*, `ioctl`) or require two-way communication with a non-transactional device or service. Finally, since xCalls is a library-based approach, it enables transactional semantics only for threads running in the same process. In consideration of TM semantics, preemptible resources extend the use of atomic transactions into code that otherwise would have to rely on relaxed transactions to perform unsafe operations, namely acquire locks or do I/O.

Atomic/lock Serialization. We augment both the STM's atomic regions and POSIX mutex locks with a special global reader/writer lock that provides mutual exclusion between atomic regions and lock-based critical sections. Mutex locks acquire the global lock in shared mode, while atomic regions acquire it exclusively. We note this approach prevents transactions from having any concurrency and slows down other uses of locks. This limitation however is an artifact of this implementation. As we suggested earlier in section 4.1, other more scalable approaches such as *cxspinlock* may be used.

Bug	App	All	Transactional Memory Fixes				
			Total	R1	R2	R3	R4
DL	Mozilla	13	9	8(2)	-	7(1)	-
	Apache	4	2	1(1)	-	1(1)	-
	MySQL	5	1	0	-	1(1)	-
AV	Mozilla	25	20	-	20(12)	-	8(0)
	Apache	7	5	-	5(3)	-	2(0)
	MySQL	6	6	-	6(2)	-	4(0)
	Total	60	43	9(2)	31(17)	9(3)	14(0)

Table 1. Applications and the number of concurrency bugs in each category together with a breakdown of how many bugs each recipe can help fix.

5.2 Methods

To evaluate whether TM *can* fix a bug and can *simply* fix a bug, we obtained the buggy source code from application repositories and used our recipes to apply transactions to fix the bug. We used a set of found and fixed concurrency bugs in Mozilla, Apache httpd, and MySQL [30], excluding order-violation bugs because TM does not fit naturally to such bugs. For a subset of the bugs, we implemented and tested a fix. For the remainder, which had no reproduction scenario, we examined the code, determined what library or function calls were made within the transaction and sketched how to apply transactions.

For each TM fix we implement, we ensure the bug is fixed by running either tests cases provided with the bug report or ones we wrote. Fixing a bug raises the risk of introducing a new bug. We ensure the correctness of our fixes by comparing them against the existing developer fixes and ensure that they provide the same atomicity properties. Furthermore, we ensure our changes do not break any regression tests. While this approach to correctness does not represent a formal proof, it provides us a degree of confidence comparable to the developers'.

To evaluate whether TM fixes a bug simply, we qualitatively compare our fix against the developers' final fix, in which the bug is actually resolved. We note that, in many places, developers chose to fix bugs without the use of locks or by removing portions of the code. As a result, our evaluation is not a comparison of using locks and transactions to fix a bug but rather of comparing what application developers currently do against transactions. To be as objective as possible, we rate the difficulty of each TM and developers' fix by considering the implementation and conceptual effort of the fix, and then pick the fix with the lowest difficulty. In general, logic or design changes are rated as hard; changes that require checking whether resources can be safely preempted or dropped are rated as medium or hard; changes that require adding a new lock in the lock hierarchy are rated as medium; and changes that require large scale code changes such as adding multiple atomic blocks are rated as medium or hard.

To evaluate whether TM *efficiently* fixes the bugs, we performed several case study experiments where we compare the performance of representative TM fixes to the performance of the developers' fixes using Intel's STM [24]. We ran tests on an Intel Core 2 2.5GHz quad-core based machine running CentOS Linux. We note that this is not the best performing transactional memory system, particularly given some of caveats noted above, so performance numbers should be treated as a lower bound.

5.3 Effectiveness

In this section, we discuss our findings of whether transactional memory is useful as a concurrency-bug fixing mechanism. We summarize our findings in Table 1: for each application and bug type, it lists the number of bugs that each recipe can fix. Each number in parentheses indicates how many bugs can be fixed only using the specific recipe.

App	Difficulty	Dev' fixes		Transactional Memory fixes			
		DL	AV	R1	R2	R3	R4
Mozilla	Easy	1	5	2	4	4	6
	Medium	1	6	4	13	2	2
	Hard	7	9	2	3	1	0
Apache	Easy	0	1	1	3	0	2
	Medium	0	2	0	1	0	0
	Hard	2	2	0	1	1	0
MySQL	Easy	0	1	0	2	0	4
	Medium	0	3	0	4	1	0
	Hard	1	2	0	0	0	0

Table 2. Characterization of developers' and TM fixes in terms of easy, medium, and hard difficulty. We characterize only fixes solved by both developers and TM.

Bug type	Application	Downcalls			
		CV	I/O	LongAction	Other
Deadlock	Mozilla	3	2	3	3
	Apache	1	0	0	0
	MySQL	0	0	0	0
Atomicity	Mozilla	3	3	3	0
	Apache	0	2	0	0
	MySQL	0	1	1	0
	Total	7	8	7	3

Table 3. Characterization of our TM fixes based on the number and type of function calls inside atomic blocks (*downcalls*).

Overall, we found that the TM can fix 43 out of the 60 bugs (71 percent) we investigated. As we could not reproduce all the bugs, we implemented and thoroughly tested 18 of the 43 fixes: 7 deadlock and 11 atomicity violation fixes. The *straightforward* recipes (Recipes 1 and 2) are sufficient to tackle 40 of the 43 bugs. The *sophisticated* recipes can fix 3 more bugs: Recipe 3 (deadlock preemption) can repair three deadlock bugs involving condition variables operations that could not be fixed using Recipe 1 (replace deadlock-prone locks). These results demonstrate that straightforward TM has the expressive power to capture the synchronization requirements regarding locks and the flexibility to execute all the code necessary for many concurrency bugs. Furthermore, extensions for failure atomicity (such as rolling back on failure) provide an easy "out" from complex deadlocks involving locks and condition variables. We describe one of these bugs in more detail in Section 5.4.

We judge that 34 out of the 43 TM-based fixes are simpler and preferable to developers' fixes. Table 2 lists the number of easy, medium, and hard fixes for each application based on the criteria of Section 5.2. In the next two sections, we further discuss the relative difficulty of our and the developer' fixes.

Many TM systems have limited support for *downcalls*, calls to non-transactional code in system services or other modules. Table 3 lists how many fixes have atomic blocks making direct or indirect calls to: condition variables (*CV*), I/O, long actions such as garbage collection (*LongAction*), and other library/module functions. Five fixes (all in Mozilla) required support for condition variables in transactions [17], two required a `retry`, eight required I/O [54], and seven required very long transactions encompassing millions of instructions. Thus, we found that incorporating memory transactions into a concurrent program requires supporting many actions beyond memory access within a transaction, which is similar to the experience of the AtomicQuake developers [60].

5.3.1 Deadlock Bugs

As shown in Table 1, TM can be used to fix 12 of the 22 deadlock (DL) bugs. We further break down the 12 bugs where TM can be used to evaluate whether a TM fix is simpler than the fix chosen by

the application developers. Overall, we judge that a TM-based fix is simpler than the developers' for 10 deadlock bugs.

In analyzing the results, we found it useful to classify bugs by the state they access and the location of the code. *Preemptible* bugs occur when there are no changes to unrelated shared state while holding the deadlocking locks; otherwise the bug is *non-preemptible* because state unrelated to the locks has changed and cannot be reverted. For example, calls to non-transactional functions or system calls are non-preemptible when executed with inevitable transactions. Similarly, code that acquires a lock and then returns to a caller in a different module is not preemptible, as it would require executing the caller's code within a transaction. *Single-module* deadlock bugs involved locks defined in a single module, while *multi-module* bugs involve locks from more than one module.

When TM does not work Most importantly, transactional memory in its pure atomic-region form (Recipe 1) cannot help deadlocks caused by condition-variable wait operations. Such deadlocks can only be approached using a combination of preemption and `retry` as described in Recipe 3. However, preemption and `retry` cannot help with deadlocks involving two-way communication such as in nested monitor lockouts [28]. In such a case, one thread holds a lock, and waits for a signal from a second thread that can only be sent after acquiring the lock held by the first thread. Such deadlocks may be only approached through design changes [28, 57]. This occurred for example in Mozilla#65146.

We also found deadlocks that span multiple modules were difficult to fix with TM, because of the large-scale changes required to modify multiple modules. Furthermore, in one case deadlock involved a third-party plugin where it was impossible to apply TM. As a result, we were not able to use TM to fix any bugs that spanned non-preemptible code in multiple modules, which represent 5 deadlock bugs. Nevertheless, it is interesting to note that in nearly all cases the developer's fix was to release a lock before continuing down the deadlocked path, which presents a dangerous decision often accompanied with significant conceptual effort to determine the correctness.

Other deadlocks were design bugs, for example when one thread waits for a signal from a component that has already been destroyed (Mozilla#27486). Thus, for problems that do not relate to circular wait of locks, or locks with a single condition variable, transactional memory provides little value. In addition, transactional memory cannot help solve deadlocks that arise through fundamental design errors rather than the mechanisms for enforcing mutual exclusion.

When TM works well Simple memory-only transactions fixed 9 of the 12 deadlock bugs fixed. These 9 all involved acquiring pairs of locks out of order. Applying Recipe 1 addressed this by removing the possibility of deadlock. We found bugs involving non-preemptible code (*e.g.*, code that can only execute as an inevitable transaction) can only be fixed by replacing deadlock-prone locks with transactions, because Recipe 3 relies on the ability to roll back at least one thread.

We rate these fixes as easy if they required few code changes and otherwise as medium or hard depending on the code changes required. In one case described below (Section 5.4.1), we replaced code that held locks for an extended period with code that instead executes a series of short transactions, with the non-preemptible code occurring between transactions. Thus, the non-preemptible lock that was held for a long period was changed to shorter transactions that could abort to prevent deadlock. This fix we rated as hard.

In contrast, the developers' fix in all cases except one was to remove the acquisition of one of the locks held to break deadlock. Switching the acquisition order was not possible in many cases as it would require drastic changes in the design of the lock hierarchy. Giving up a lock is a conceptually challenging task as it requires deep understanding of the code to reason about the safety of this action so we judge such fixes as hard. In one case, the developer's fix was as simple as switching the acquisition order of locks as the locks were acquired locally in a single function. We judge this fix as easy and favor it compared to TM that requires replacing locks with transactions.

The remaining 3 bugs required using revocable locks with Recipe 3 to acquire locks within a transaction. Two of those bugs involve condition variable operations that could not be fixed using transactional condition variables, which commit the transaction before waiting, but could be fixed via a `retry`, which aborts. We rate these fixes as hard as they require reasoning that is safe to replace the condition variable with a `retry`. In addition, we found that Recipe 3 can reduce the implementation effort to fix 6 of the 9 bugs fixed using Recipe 1 by localizing fixes to the bug sites instead of modifying all uses of the deadlocking locks.

Overall, we favor 2 developers' fixes as they are as easy as TM or easier, and favor 10 TM fixes as conceptual and implementation effort is less than with the developer's fix. The 2 developers' fixes include the one easy fix described above, and another of medium difficulty, which requires a simple design change.

5.3.2 Atomicity-Violation Bugs

As shown in Table 1, TM can be used to fix 31 of 38 atomicity-violation (AV) bugs. Overall, we judge that a TM-based fix is simpler than the developers' for 24 bugs.

When TM does not work There were 7 atomicity bugs that could not be fixed with transactions for a variety of reasons. One situation occurs when the application must atomically issue a long-latency operation and process a callback event when the operation completes. For example, Mozilla holds a lock while loading a URL and invokes a callback once the URL is fetched (bug#19421), which can take an arbitrarily long time. Contention for the lock is not high so its use is acceptable. However, a memory transaction is global to the process, particularly if it uses inevitability mechanisms that acquire a global lock. Therefore, using transactions around a long-latency operation would prevent all other transactions from making progress. Having support to issue asynchronous I/O and execute a callback upon I/O completion within a transaction would help fixing this problem.

In other cases, the bugs were not simply atomicity violations, but required additional semantics: in Mozilla, one code section required both atomicity and that only one thread executed the code (exactly-once semantics). This requirement is beyond TM's guarantees.

Finally, there were several atomicity violations regarding I/O: lost notifications waiting for I/O to arrive (Mozilla#72965) and races between two processes reading from the same pipe (Apache #7617). Like the last two problems, these problems arise not from the atomicity of memory operations but from the atomicity of I/O operations across multiple processes (the kernel and a process, or two separate processes), which current TM systems do not address.

When TM works well Atomicity violations caused by code with completely missing synchronization are the best-case scenario for TM. Such bugs are fixed using Recipe 2. In our study, this type of violation included 22 out of the 38 atomicity-violation bugs, 17 of which could be fixed using Recipe 2. In 12 of the 17 fixes, bugs could be fixed with a single atomic block. We judge 9 of the 12 as easy fixes, while we judge the remaining 3 fixes as medium difficulty because we had to reason that wrapping downcalls inside the atomic block was safe.

Bug ID	Cause	Characteristics	Fix	Perf.	Lines of Code	
					Dev	TM
Mozilla-I	DL	Involves locks only	1 3	21% 85%	23 23	1039 16
Apache-I	DL	Involves lock and wait	3	78%	32	14
Apache-II	AV	Complete missing synchronization	2	96.5%	20	5
MySQL-I	AV	Partial missing synchronization	4	50%	103	4

Table 4. Bugs and corresponding fix recipes applied for demonstration purposes. Performance is relative to that of developer's fix. Lines of code (LOC) includes both lines added and lines modified.

In 3 of these 12 cases, the developers' fix was to wrap actions inside a critical section by adding a new lock, which we judge as medium difficulty because it requires reasoning that the new lock does not introduce a deadlock. The remaining 9 developer's fixes require profound understanding of the application to develop a specialized local fix, such as switching the order of statements in the source code [30]. We judge 2 of them as easy, though, because the fix is replacing a global variable with a local one, while we judge the rest as hard. As an example of a hard case, the developer implemented a custom optimistic concurrency protocol by copying shared data locally, updating the copy, and then updating the shared data atomically with a single store instruction. This is essentially a form of optimistic concurrency control that suits TM well. The remaining 5 of the 17 fixes required us to find all the places where synchronization is missing and place multiple atomic blocks. We judge these fixes as medium. We find though that using TM to fix these is a bit simpler than the developer fixes, again because a developer does not have an already existing synchronization mechanism to exploit.

The remaining 16 of the 38 atomicity violations include asymmetric atomicity-violation bugs, where most accesses to shared data use the correct lock but some do not. Of these, 14 could be fixed either using Recipe 2 or Recipe 4. Such bugs though are easier to fix by wrapping the code region intended to be executed atomically in an atomic section that is serialized to any other lock-based critical section (Recipe 4), because this recipe requires fewer changes compared to Recipe 2. Developers commonly fix such bugs by acquiring an existing lock either because they forgot to use it before or they used the wrong one. In 5 cases the fix was as easy as extending the coverage of an existing lock critical section without wrapping calls to functions that may be deadlock prone. In these cases the developer's fix is clearly simpler than a TM fix that replaces existing locks with transactions. Even when the correct lock is hard to identify, using TM is no simpler because the developer must still identify all places where the data are accessed. Nevertheless, Recipe 4 significantly simplifies the fix, making TM competitive with the developer's fix.

Overall, we favor 7 developer's fixes as they are as easy as TM or easier, and favor 24 TM fixes as the conceptual and implementation effort is less than with the developer's fix.

5.4 Demonstrating Applicability

To further clarify the application of transactional memory to concurrency bugs and to evaluate its performance, we demonstrate its application to a subset of the bugs we studied. We selected this subset to cover the spectrum of the bugs targeted by our fix recipes. For each bug, we give a description of the bug, present both the developer's and TM fix, and finally compare the two fixes. Table 4 summarizes the bugs we used.

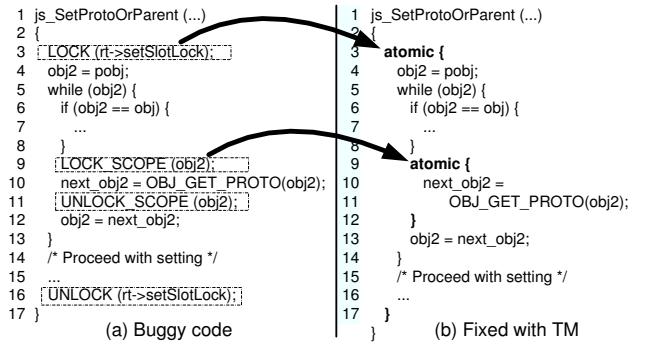

Figure 2. A deadlock code sample from Mozilla (SpiderMonkey: jsobj.c) modified for legibility.

5.4.1 Case Study 1: *Mozilla-I: Deadlock*

Figure 2 illustrates a deadlock bug found in SpiderMonkey, Mozilla's JavaScript engine, that involves locks only. SpiderMonkey uses an ownership-based mechanism to synchronize accesses to an object by multiple threads. Under this mechanism, objects are protected by a "scope lock" and an owner field. If a thread owns an object, it can access the object with a simple test of the owner field and without acquiring any locks. If a thread accesses objects owned by another thread, then SpiderMonkey follows a complex revocation protocol to switch the object from using the owner field to using the scope lock. Switching to a scope lock is a blocking operation because the owning thread could be actively accessing the object. The motivation behind this complex synchronization mechanism, is that most objects are only ever locked by a single thread [35].

The deadlock bug occurs when two threads try to lock the same set of objects in different orders. Suppose a first thread executing the function in Figure 2(a) owns `setSlotLock` and needs to lock an object's scope in line 9. But that scope is exclusively owned by a second thread, which is blocked behind `setSlotLock`. The first thread cannot claim the scope from thread 2 so the threads deadlock.

Developer fix SpiderMonkey developers solved this bug by forcing threads to drop ownership of objects before blocking. This required the addition of a new condition variable and small changes in three files. This fix adds overhead to drop and reacquire ownership, yet was adopted because it reliably prevents the deadlock. We judge this fix as hard, as the developers had to reason that dropping ownership was safe and would not violate atomicity.

TM fixes As this deadlock involves locks only, we may fix it using either Recipe 1 or Recipe 3 if locks can be made revocable. When using Recipe 1, we fix this bug by replacing the deadlock-prone locks with atomic sections as shown in Figure 2(b), deprecating the notion of ownership, and thus eliminating the complex revocation protocol. As there is no longer the notion of long-lived ownership, atomic sections, which synchronize access to objects, are shorter. While this fix is conceptually straightforward to reason about, it has a moderate implementation complexity as it requires modifications across 15 files to replace locks with atomics. However, as we demonstrate below it requires hardware support to perform well. We judge this fix as hard, similarly to the developers' fix.

When using Recipe 3, we fix this bug by making the locks involved in the deadlock revocable. We first interpose all calls to `lock`/`unlock` and `lock_scope`/`unlock_scope` with calls to the lock/unlock routines of our revocable lock implementation. We then fix the deadlock bug by wrapping the single deadlocking site inside a transaction. The deadlock-prone code executes within a

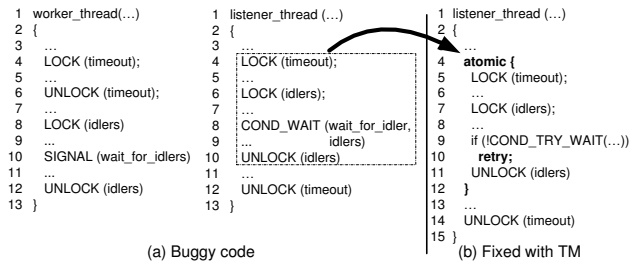

Figure 3. A deadlock code sample from Apache (httpd-2.2.0: event.c, fdqueue.c), modified for legibility.

transaction and acquires locks revocably; if the transaction cannot acquire a lock it aborts and retries, thus preventing any deadlock. We judge this fix as medium difficulty, as we had to reason about the safety of preemption.

Comparison While the first fix based on Recipe 1 has a moderate implementation complexity, it substantially simplifies the locking protocol. This results in more maintainable code and additionally has the advantage that it *solves another **four** reported deadlock bugs as a side effect* of removing scope locks, which overall took the developers a year to fix. So while hard to implement, we favor it compared to the developers' fix. In comparison, the second fix based on Recipe 3 requires far fewer code changes: only to interpose calls to lock/unlock routines (which could be done automatically by redefining macros) and the deadlocking code. However, it requires some conceptual effort to determine that is safe to revoke any resources used while executing in the transaction. Furthermore, this approach only fixes one bug, while the first fix fixed many other deadlock bugs found later by the developers.

We measured the performance of our fixes and compared it to the performance of the developers' fix using the SunSpider JavaScript benchmark [55]. We run tests in four threads running the same SunSpider script. Even if scripts do not share data, we are still able to exercise the multithreaded code path because all threads run within the same runtime. The first fix performs 79% worse than the developers' fix. This occurs because the code is on the critical path, as evidenced by the effort put in by developers to avoid acquiring a lock. The second fix improves performance significantly as it performs only 15% slower than the developer's fix. This performance improvement arises because critical path code can execute without transactions or atomic operations. As there is still non-critical path code that executes transactionally the workload experiences some drop in performance. These results show that avoiding synchronization is important and that asymmetric deadlock preemption can provide the benefit of transactions without hurting performance too much.

To gauge the benefit of hardware support for TM, we ran the code on the proposed hardware TM platform LogTM-SE [2] [59]. The performance of the first fix increases to 99.3% of the developer's fix. Thus, the performance of hardware TM is good enough that transactions can take the place of ad hoc locking protocols [58], such as SpiderMonkey's ownership protocol, and thereby greatly simplifies code.

5.4.2 Case Study 2: *Apache-I: Deadlock between lock and wait*

Figure 3(a) depicts a deadlock bug between lock and wait operations. This is a section of code taken from Apache in which the listener thread holds the `timeout` mutex protecting the timed-out

[2] We simulated LogTM-SE using Wisconsin GEMS [32]. The configuration we used is a 1GHz 8-core SPARC CMP. The operating system is Solaris 9 and the compiler we used is GCC 3.4.4.

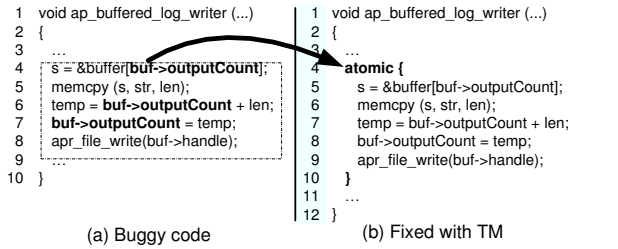

Figure 4. An unsynchronized code sample from Apache (httpd-2.0.45: mod_log_config.c), modified for legibility.

sockets list while waiting for an idle worker thread. The worker thread block waits for the timeout mutex before signaling to the listener thread its availability, thus leading to a deadlock. The listener holds this mutex while waiting to ensure that it atomically removes a timed-out socket from the list and hands the socket off to an idle worker.

Developer fix The Apache developer's fix removes the circular wait by releasing the timeout mutex before calling through the conditional wait. The developers reached this fix after three failed attempts. To make this fix correct, they wrote code that compensates for breaking atomicity. We judge this fix as hard, given its history and the compensation required.

TM fix As this deadlock does not involve only locks, we use preemption to fix the bug as suggested by Recipe 3. While this at first sight might appear as a nested monitor lockout, which as discussed in Section 5.3.1 cannot be fixed with TM, careful inspection reveals that there is no two-way communication between the worker and listener: the worker needs to communicate to the listener its availability but the listener does not need to assign work to the worker immediately. We therefore modify the listener thread to acquire locks and perform any wait operations revocably inside a transaction, as shown in Figure 3(b). If the listener thread executing inside the transaction does not find an idle worker, it aborts the transaction, releases any locks it acquired, and retries the transaction. This allows the worker thread to acquire the locks it needs and make progress. We judge this fix as hard.

Comparison We judge this fix to be simpler than the developer fix, because it leverages transactions to automatically guarantee the atomicity of the listener's operation without having to reason and write compensating code. We measure the performance of the two fixes using `ab`, the Apache HTTP server-benchmarking tool, running on the same machine as the web server to avoid any network bottlenecks. We saturated the machine by performing 128 multiple requests at a time. We find that the TM fix performs 22% slower than the developer's fix, but that overhead could be reduced by making `retry` block rather than spin. Since this is performance under a stress test workload, we expect the performance difference between the two fixes to be less under realistic workloads. Thus, we believe the TM fix offers a good balance between effort and performance.

5.4.3 Case Study 3: *Apache-II: Missing Synchronization Atomicity Violation*

Figure 4 illustrates a case of an atomicity violation caused by missing synchronization. This code, taken from the Apache web server, has a race condition in which two threads may compete over the buffer, independently advancing `outputCount` and producing either garbage in the log or buffer overflow. The race condition is prevented by executing all the logic of the function from line 2 onwards as a critical section.

Developer Fix Apache developers fixed this bug by assigning a lock to each log device (`buffered_log`) that they acquire on entry to the function `ap_buffered_log_writer`. While a single static lock could protect the entire function, concurrency favors the more scalable solution at the expense of introducing several new locks to the system. We judge this fix as medium difficulty.

TM Fix By following Recipe 2, we insert a single atomic block from line 4 to 10 to protect the critical section, effectively fixing the bug with only five lines of code. The file I/O is performed using an xCall that defers the actual I/O until the transaction commits. We judge this fix as easy.

Comparison Compared to the developers' fine-grained locks, the TM fix for this bug provides equal concurrency, as writes to different logs can proceed in parallel. In addition, the TM fix is local: the only code that changes is within a single function, whereas the developers' fix required changes elsewhere in Apache to add a lock to the `buffered_log` structure and manage its creation. We measure the performance of the two fixes using `ab` as described earlier in Section 5.4.2. Our TM fix performs comparable to the developer's fix, being only 4% slower. Overall, we believe our TM fix offers a simpler fix compared to the developer's one without penalizing performance.

5.4.4 Case Study 4: *MySQL-I: Partially Missing Synchronization Atomicity Violation*

This MySQL atomicity violation bug occurs when the following two queries execute in parallel:

```
INSERT INTO table VALUES (...);
DELETE FROM table;
```

The first query inserts values into a table while the second deletes the entire table. Within MySQL, the code in Figure 5 to execute the second command is not correctly synchronized with the query logging code: the two queries may execute in one order while the log lists the queries in the opposite order. This happens when an optimized delete function releases logical isolation over the table too early, unlocking `lock_open` before logging and allowing the entire insert function to begin and end before the delete is logged. In contrast, the insert operation holds a logical lock for both the insert and the log write.

Developer fix In the next working version of the code, this optimization disappeared completely, and the surrounding code was restructured. It is unclear whether the developer was unable to fix the bug or whether s/he recognized a fix and removed the code for other reasons. An obvious lock-based fix is to extend the global `lock_open` to cover both operations, but this requires an understanding of that lock's purpose as well as an understanding of the performance implications; `lock_open` is the most contended lock in old versions of MySQL. We judge this fix as hard.

TM Fix Transactional memory fixed this bug with little effort and without requiring a deep understanding of table synchronization, as illustrated in Figure 5. An atomic/lock serializable section (Recipe 4), on the right, wraps the physical operation of the delete query together with logging. This atomic section is serializable against any lock-based critical section. This includes the critical section of thread 1 that uses `lock_open` so any concurrent table-modifying operation is prevented. Overall, this change is local to the (very rare) delete-all-rows operation and obviates any need to understand the correct logical locking to apply in this scenario. We judge this fix as easy.

Comparison We measured the performance of our fix with two tests: repeatedly deleting all rows from different tables, and repeatedly inserting rows into a table. The performance of our fix is identical to that of the buggy version. Thus, the TM fix is simple, expressive and non-invasive, and therefore preferable to a rewrite of the code.

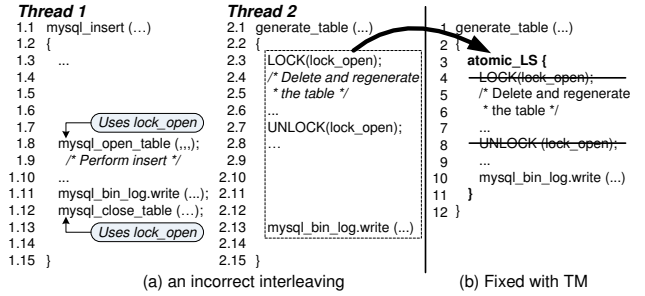

Figure 5. An unsynchronized code sample from MYSQL, modified for legibility.

6. Summary and Conclusions

With this study, we find that current TM is not useful in 17 of 60 atomicity-violation and deadlock bugs examined. This surprising result arises because many concurrency bugs are not about shared data; rather than they concern synchronization, such as condition variables, or I/O, such as file or network access. However, we do find that straightforward uses of TM can fix 40 of the 60 bugs, and sophisticated uses of TM can fix 3 additional bugs and simplify the fixes of 20 of the 40 bugs. Overall, we found that among the 43 bugs that could be fixed by TM, the TM fix may be preferable in 34 cases. These results demonstrate that transactional memory, as proposed, is moderately useful in concurrent programs, but that it does not address enough of the problems that cause bugs.

While replacing existing locks with transactions may require some non-trivial implementation effort, we have seen that transactions can simplify existing logic and fix other bugs as a side effect. Furthermore, the resulting code, structured as clearly identified atomic blocks, may be easier for developers to understand as demonstrated in Case Study 1.

On the other hand, some bugs are trivially fixed with locks, for example when extending an existing critical section to include more code. In such cases, applying transactions yields little value. Finally, even for bugs that can be more simply fixed with TM, the performance overhead of TM may be too large in the absence of hardware support, because software TM is generally more expensive than locking when there is no lock contention [52].

For some bugs, the primitive of atomic execution may not have sufficient expressive power to solve the problem. We identify three such deficiencies. First, deadlocks regarding I/O channels, such as pipes, are not addressed by memory transactions, as the contended resource (the I/O channel) is outside the process. Second, asynchronous I/O allows the OS to write to memory at a later time, possibly after the transaction commits or aborts, thus making it work poorly with transactional memory. Third, shared memory similarly allows other processes to access memory, while current STMs only enforce isolation for threads in the same process. Extending TM along these dimensions might be a fruitful research topic. At least for our work, such extensions would allow TM to fix five more bugs.

Finally, in several bugs the use of STM slowed down performance, especially when TM was used to replace ad hoc synchronization where performance is critical. While low-performance cases could be considered as non-applicable for TM, we believe that as STM systems mature or as TM becomes available in hard-

ware, the performance for those cases will greatly improve, thus making them more suitable for TM.

Acknowledgements

This work is supported in part by National Science Foundation (NSF) grants CNS-0720565 and CNS-0834473. We would like to thank Wei Zhang for help with the reproduction of bugs. We would also like to thank our shepherd, Michael Scott, and the anonymous reviewers for their invaluable feedback. Swift has a significant financial interest in Microsoft.

References

[1] The IBM Blue Gene/Q compute chip with SIMD floating-point unit. Hot Chips 23, Aug. 2011.

[2] A.-R. Adl-Tabatabai and T. Shpeisman. Draft specification of transactional language constructs for C++, version 1.0. `http://software.intel.com/file/21569`, Aug. 2009.

[3] AMD Corporation. Advanced synchronization facility: Proposed architectural specification, rev. 2.1. `http://developer.amd.com/assets/45432-ASF\_Spec\_2.1.pdf`, Mar. 2009.

[4] L. Baugh and C. Zilles. An analysis of I/O and syscalls in critical sections and their implications for transactional memory. In *Proc. of the 2nd ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Aug. 2007.

[5] 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 *Proc. of the 34th Annual Intnl. Symp. on Computer Architecture*, June 2007.

[6] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Commun. ACM*, 51(11):40–46, 2008.

[7] S. Cherem, T. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. In *Proc. of the SIGPLAN 2008 Conf. on Programming Language Design and Implementation*, June 2008.

[8] L. Chew and D. Lie. Kivati: Fast detection and prevention of atomicity violations. In *Proc. of the 5th ACM European Conf. on Computer Systems*, Apr. 2010.

[9] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Riviere. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *Proc. of the 5th ACM European Conf. on Computer Systems*, Apr. 2010.

[10] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid NOrec: A case study in the effectiveness of best effort hardware transactional memory. In *Proc. of the 16th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Mar. 2011.

[11] L. Dalessandro, M. L. Scott, and M. F. Spear. Transactions as the foundation of a memory consistency model. In *Proc. of the 24th International Symp. on Distributed Computing*, Sept. 2010.

[12] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by abolishing ownership records. In *Proc. of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Jan. 2010.

[13] P. Damron, A. Fedorova, Y. Lev, V. Luchango, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proc. of the 12th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[14] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proc. of the 14th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Mar. 2009.

[15] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proc. of the 20th International Symp. on Distributed Computing*, Sept. 2006.

[16] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *Proc. of the SIGPLAN 2009 Conf. on Programming Language Design and Implementation*, June 2009.

[17] P. Dudnik and M. M. Swift. Condition variables and transactional memory: Problem or opportunity? In *Proc. of the 4rd ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Feb. 2009.

[18] M. Emmi, J. S. Fischer, R. Jhala, and R. Majumdar. Lock allocation. In *Proc. of the 34th ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages*, 2007.

[19] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proc. of the 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Feb. 2008.

[20] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proc. of the 31st Annual Intnl. Symp. on Computer Architecture*, June 2004.

[21] T. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory, 2nd edition*. Morgan & Claypool Publishers, 2010.

[22] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In *Proc. of the 10th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, June 1991.

[23] HIPEAC. Gcc for transactional memory. `http://www.hipeac.net/node/2419`.

[24] Intel. Intel C++ STM compiler, Prototype edition. `http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/`, 2009.

[25] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *Proc. of the SIGPLAN 2011 Conf. on Programming Language Design and Implementation*, June 2011.

[26] H. Jula, D. M. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *Proc. of the 8th USENIX Symp. on Operating Systems Design and Implementation*, Dec. 2008.

[27] E. Koskinen and M. Herlihy. Dreadlocks: Efficient deadlock detection. In *Proc. of the 20th ACM Symp. on Parallel Algorithms and Architectures*, June 2008.

[28] B. W. Lampson and D. D. Redell. Experience with processes and monitors in Mesa (summary). In *Proc. of the 7th Symp. on Operating System Principles*, Dec. 1979.

[29] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased transactional memory. In *Proc. of the 2nd ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Aug. 2007.

[30] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proc. of the 13th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Mar. 2008.

[31] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-aid: Detecting and surviving atomicity violations. In *Proc. of the 35th Annual Intnl. Symp. on Computer Architecture*, June 2008.

[32] 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.

[33] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: Synchronization inference for atomic sections. In *Proc. of the 33rd ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages*, 2006.

[34] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. Logtm: Log-based transactional memory. In *Proc. of the 12th IEEE Symp. on High-Performance Computer Architecture*, Feb. 2006.

[35] Mozilla. Performance: Page load and DHTML performance. `http://wiki.mozilla.org/Performance:Home_Page`, Mar. 2010.

[36] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and implementation of transactional constructs for c/c++. In *Proc. of the 23rd SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Application*, Oct. 2008.

[37] V. Pankratius. Does transactional memory keep its promises? Results from an empirical study. Technical Report 2009-12, University of Karlsruhe, Germany, Sept. 2009.

[38] M. Pohlack and S. Diestelhorst. From lightweight hardware transactional memory to lightweight lock elision. In *Proc. of the 6th ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2011.

[39] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating systems transactions. In *Proc. of the 22nd ACM Symp. on Operating System Principles*, Oct. 2009.

[40] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies — a safe method to survive software failure. In *Proc. of the 20th ACM Symp. on Operating System Principles*, Oct. 2005.

[41] S. Rajamani, G. Ramalingam, V. P. Ranganath, and K. Vaswani. ISOLATOR: Dynamically ensuring isolation in concurrent programs. In *Proc. of the 14th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Mar. 2009.

[42] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *Proc. of the 10th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2002.

[43] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proc. of the 32nd Annual Intnl. Symp. on Computer Architecture*, June 2005.

[44] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel. MetaTM/TxLinux: Transactional memory for an operating system. In *Proc. of the 34th Annual Intnl. Symp. on Computer Architecture*, June 2007.

[45] 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 *Proc. of the 21st ACM Symp. on Operating System Principles*, Oct. 2007.

[46] C. J. Rossbach, O. S. Hofmann, and E. Witchel. Is transactional programming actually easier? In *Proc. of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Jan. 2010.

[47] A. Roy, S. Hand, and T. Harris. A runtime system for software lock elision. In *Proc. of the 4th ACM European Conf. on Computer Systems*, Apr. 2009.

[48] 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 *Proc. of the 11th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Mar. 2006.

[49] T. Shpeisman, A.-R. Adl-Tabatabai, R. Geva, Y. Ni, and A. Welc. Towards transactional memory semantics for C++. In *Proc. of the 21st ACM Symp. on Parallel Algorithms and Architectures*, June 2009.

[50] M. F. Spear, M. Silverman, L. Dalessandro, M. M. Michael, and M. L. Scott. Implementing and exploiting inevitability in software transactional memory. In *Proc. of the 37th Intnl. Conf. on Parallel Processing*, Sept. 2008.

[51] M. M. Swift, H. Volos, N. Goyal, L. Yen, M. D. Hill, and D. A. Wood. OS support for virtualizing hardware transactional memory. In *Proc. of the 3rd ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Feb. 2008.

[52] T. Usui, Y. Smaragdakis, and R. Behrends. Adaptive locks: Combining transactions and locks for efficient concurrency. In *Proc. of the 4rd ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Feb. 2009.

[53] H. Volos, N. Goyal, and M. M. Swift. Pathological interaction of locks with transactional memory. In *Proc. of the 3rd ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Feb. 2008.

[54] H. Volos, A. J. Tack, N. Goyal, M. M. Swift, and A. Welc. xCalls: Safe I/O in memory transactions. In *Proc. of the 4th ACM European Conf. on Computer Systems*, Apr. 2009.

[55] Webkit. Sunspider javascript benchmark. `http://www2.webkit.org/perf/sunspider-0.9/sunspider.html`.

[56] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable transactions and their applications. In *Proc. of the 20th ACM Symp. on Parallel Algorithms and Architectures*, June 2008.

[57] H. Wettstein. The problem of nested monitor calls revisited. *SIGOPS Operating Systems Review*, January 1978.

[58] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In *Proc. of the 9th USENIX Symp. on Operating Systems Design and Implementation*, Oct. 2010.

[59] 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 *Proc. of the 13th IEEE Symp. on High-Performance Computer Architecture*, Feb. 2007.

[60] F. Zyulkyarov, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguadé, T. Harris, and M. Valero. Atomic Quake: Using transactional memory in an interactive multiplayer game server. In *Proc. of the 14th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Feb. 2009.