Sandboxing Transactional Memory[*]

Luke Dalessandro and Michael L. Scott
University of Rochester
Department of Computer Science
{luked,scott}@cs.rochester.edu

ABSTRACT

Correct transactional memory systems (TMs) must address the possibility that a speculative transaction may read mutually inconsistent values from memory and then perform an operation that violates the underlying language semantics. TMs for managed languages can leverage type safety, just-in-time compilation, and fully monitored exceptions to *sandbox* transactions, isolating the rest of the system from damaging effects of inconsistent speculation. In contrast, TMs for unmanaged languages that lack these properties typically avoid erroneous behavior by *validating* a transaction's view of memory incrementally after each read operation.

Recent results suggest that performing validation *out-of-band* can increase performance by factors of 1.7× to 5.2× over incremental validation, but allowing a transaction's main computation to progress in parallel with validation introduces periods in which inconsistent speculative execution may violate language semantics. Without sandboxing—which some authors have suggested is not possible in unmanaged languages—programmers must manually annotate transactions with validation barriers whenever inconsistency might lead to semantic violations, an untenable task.

In this work we demonstrate that sandboxing for out-of-band validation is, in fact, possible in unmanaged languages. Our implementation integrates signal interposition, periodic validation, and a mix of static and dynamic instrumentation into a system comprising the LLVM-based Dresden TM compiler and the RSTM runtime. We show that these mechanisms introduce negligible overhead, thus extending the results of out-of-band validation to POSIX programs without requiring manual annotation. Furthermore, we establish sandboxing as a technique that can complement, or replace, incremental validation in any TM that keep speculatively written values in a private buffer.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; D.3.4 [**Programming Languages**]: Processors—*Run-time environments*

Keywords

Transactional memory, validation, sandboxing, opacity

1. INTRODUCTION

Transactional Memory (TM) [16, 18] raises the level of abstraction for synchronization, allowing programmers to delimit operations that need to execute atomically without specifying *how* that atomicity is to be achieved. The underlying system then attempts to execute atomic sections of separate threads concurrently whenever possible, typically by means of speculation.

Speculative transactions are said to *commit* at the end of their execution; others *abort*, roll back, and try again. All speculation-based TM systems require a mechanism to detect and resolve conflicts between transactions, ensuring that they commit only as part of a global serialization order. Additionally, committed transactions must obey the semantics of the source programming language.

A speculative transaction becomes inconsistent when it reads a pair of values that were never simultaneously valid in the execution history defined by the program's committed transactions. To ensure overall program correctness, at least in the general case, a TM must force inconsistent transactions to abort. To this end, transactions in software TM runtimes (STM) typically *validate* by double-checking their read sets, and abort in the event of inconsistency.

STM runtimes for managed languages like Java and C# [2, 17], as well as systems based on dynamic binary instrumentation [24], validate transactions before they commit, leveraging type safe, just-in-time compilation, and managed exceptions to *sandbox* speculative transactions and protect the system from the effects of inconsistent execution.

In unmanaged languages—where some authors have implied sandboxing is impossible [22], several historical TM systems depended on explicit programmer-inserted validation for correct execution [12, 15, 18, 28], a strategy that has been discarded as difficult and error prone. Modern TMs for unmanaged languages validate *incrementally* [19]—after every transactional read operation—a technique that has been formalized as the theory of *opacity* [13, 14]. Minor relaxations of opacity [31] and optimizations to dynamically elide redundant checks [8, 27, 32] can significantly re-

[*]This work was supported in part by the National Science Foundation under grants CCR-0963759, CCF-1116055, and CNS-1116109.

PACT'12, September 19–23, 2012, Minneapolis, Minnesota, USA.

duce the frequency and overhead of validation. In the worst case, however, validation remains a problem quadratic in the number of transactional read operations, a cost paid on the transaction's critical path.

Recent work by Casper et al. [3] and Kestor et al. [20] suggests that moving validation off of the critical path and performing it *out-of-band*—in parallel with the "real work" of the transaction—can result in performance improvements of 1.7× and 1.8×–5.2× respectively. Naive out-of-band validation, however, allows inconsistent speculative transactions ("zombies") to execute erroneous operations that result in visible violations of the semantics of the source programming language. Casper et al. and Kestor et al. echo historical techniques and require the programmer to manually annotate any operation within a transaction that may violate language semantics if performed by a zombie.

In the extreme, the programmer has the option of annotating every read—essentially restoring the validations that would be performed by an opaque system. Anything shy of this, however, requires reasoning about the potential effect of data races—a task we consider untenable. We believe the more attractive option, even in the context of an unmanaged language, is to rely on the compiler and runtime to protect the programmer from zombie execution by automatically suppressing the effects of dangerous operations—those whose behavior cannot be guaranteed to be invisible to other threads—until the transaction is known to be consistent. A complete characterization of dangerous operations is tricky (see Section 2.1); at the very least we must guard against faults and against stores to incorrectly computed addresses.

To the best of our knowledge, no previous project has carefully explored the feasibility of such sandboxing in unmanaged languages; however, its capacity to eliminate control dependencies between validation and most operations in the rest of the transaction makes it a compelling study.

Sandboxing is appealing for a number of reasons in addition to out-of-band validation. It offers the opportunity to relax the coupling in hardware TM systems between conflict detection in the memory hierarchy and execution in the processor core: precise, immediate notification of transaction conflicts would not be needed if the compiler always generated a "TM fence" before each dangerous instruction [4]. Given that dangerous operations (i.e., those that may result in errant behavior when executed inconsistently) tend to be much less common than transactional loads, it has the potential to reduce the validation overhead in TM implementations that perform in-line validation. Spear et al. [29] also suggests that it can serve as a solution to the privatization problem in some circumstances.

In this paper we show that transactional sandboxing is possible in POSIX-compliant C programs, and is practical both for TM systems that validate out-of-band and for lazy-versioning TMs in general. This result strongly suggests that the performance results reported by Casper et al. and Kestor et al. are valid for all programs—not just those that have been manually annotated for safety. More broadly, our results unlock a wide and previously proscribed design space for TM development.

In Section 2 we characterize potentially dangerous operations for zombie transactions in POSIX-compliant C. We also explore the interaction of sandboxing and several common STM algorithms. In Section 3 we describe our sandboxing implementation, including the algorithms used to instrument dangerous instructions and the algorithm-independent machinery used to address infinite loops and faults. We evaluate the overhead of this infrastructure for out-of-band validation TMs in Section 4. We extend the evaluation to systems with in-line validation in Section 5, by presenting and evaluating a "maximally lazy" sandboxed STM based on the open-source RSTM framework [25]. Finally, we conclude in Section 6.

2. SANDBOXING PRAGMATICS

We assume a semantics in which transactions are *strictly serializable* (SS): they appear to execute atomically in some global total order that is consistent with program order in every thread; further, each transaction is globally ordered with respect to preceding and following nontransactional accesses of its own thread. Given an SS implementation, one can prove that every data-race-free program is *transactionally sequentially consistent* (TSC) [5, 30]: its memory accesses will always appear to occur in some global total order that is consistent with program order and that keeps the accesses of any given transaction contiguous. These semantics are an idealized form of those of the draft transactional standard for C++ [1].

In contrast to opacity, which requires aborted transactions to observe the same consistency constraints as successful ones, aborted transactions play no role in SS. In principle, a transaction that is going to abort may do anything at all, provided the resulting history of committed transactions is still SS. In an implementation adhering to these semantics, sandboxing must ensure the isolation of *zombie transactions*, which have performed a speculative read that cannot be explained by any serial execution [18], but it need not restrict the internal behavior of such transactions beyond this isolation constraint.

2.1 Dangerous Operations

We say that an operation is *dangerous* if its execution in a zombie transaction may allow the user to observe an execution that is not SS. Here we enumerate what we believe to be a complete list of dangerous operations (a formal proof of completeness is deferred to future work). Section 3 details the concrete sandboxing mechanisms that we implement.

In-place Stores.

Zombie transactions that perform a store in-place (to a native address rather than a private buffer), are a primary concern for a sandboxing runtime. Reading such an inconsistently written value, or a stale value if the write was to an incorrect address, may constitute a data race, violate the sequential language semantics, or even result in execution of arbitrary code. In-place stores may target shared or private locations (e.g., the stack), and may be either instrumented (e.g., in an eager, in-place STM) or uninstrumented (e.g., if the compiler or programmer has concluded that there is no TSC execution in which a particular store can be part of a race).

In-place stores that cannot be proven consistent must be preceded by a run-time validation check.

Indirect Branches.

An inconsistent indirect branch may lead to an unprotected in-place store, or to executable data that *looks like* one. Compilers use such branches to implement virtual function calls, large **switch** statements, computed **goto**s, and **return**s. Fortunately, STM implementations already perform run-time mapping of native function pointers to their transactional clones' addresses. A successful lookup implies that the target will have proper sandboxing instrumentation. If a clone is not found, then the transaction will switch to *serial-irrevocable* execution, which precludes roll-back. This transition includes an embedded validation. These observations imply that, as long as we protect the integrity of the stack, no new mechanism is required to sandbox indirect calls.

Faults.

The sandboxing runtime must distinguish faults (hardware exceptions) that occur due to inconsistent execution from those that would have occurred in some TSC execution of the program, and prevent inconsistent signals from becoming visible to the programmer. In a POSIX-compliant C/C++ implementation, such faults are encoded as synchronous signals. Olszewski et al. [24] suggest either containing inconsistent signals inside the operating system by making the kernel transaction-aware, or relying on user-space signal handling to suppress inconsistent signals once received. They implemented the first option; we will use the second (Section 3.2).

Infinite Loops and Recursion.

A zombie transaction may enter into an infinite loop or infinite recursion due to inconsistent execution. These may not be a concern if validation is performed continually and out-of-band, and reported to the transaction thread via asynchronous interrupt. Some out-of-band systems, however (including those of Casper et al. and Kestor et al.), require the transaction thread to poll for notifications, and any systems with in-line validation similarly requires an active mechanism to detect and recover from an inconsistent infinite loop or recursion. (Exhaustion of stack space is a synchronous event that can, with care, be treated as any other fault.)

Previous systems have instrumented loop back edges with a check to force periodic validation [24, 28]. This approach adds overhead that can be expensive for tighter loops, and pollutes hardware resources like branch predictors. We use an alternative approach in which timers force the STM to validate periodically. This avoids common-case overhead, and the timer period can be sufficient long that applications that do not suffer from inconsistent infinite loops pay very little overhead. If infinite loops are common, however, timer-based validation may be slow to detect the problem. We could imagine a hybrid approach in which hot-patch locations are left on loop back edges so that polling code can be injected into loops that show a high probability of infinite looping.

TM Commit.

Many opaque TM algorithms allow read-only transactions to commit without validating, under the assumption that they were correct as of the validation performed at their last read operation. This assumption isn't valid in a sandboxed TM, which must validate again at commit if any shared locations have been read since the most recent previous validation. In a hardware TM system (not considered here) commit instructions would likewise need sandboxing [4]. This said, end-of-transaction validation imposes no additional penalty on systems with out-of-band validation: these always synchronize with the validation entity at commit.

Waivered Code Regions.

TM APIs may allow programmers to specify that certain *waivered* code regions should run without instrumentation. Because these regions may perform arbitrary actions, they require pre-validation.

System and Library Calls.

Many system calls are incompatible with optimistic transactional execution. We expect that TM systems will force transactions to become irrevocable—as suggested by the draft C++ TM proposal [1]—before performing a system call. This already entails validation so we do not consider system calls to be dangerous. Library calls may be compiled to be transactional, and in such case they must be compiled in a manner compatible with zombie execution.

We have found that a smart compiler like the DTMC [11] will occasionally allow a library call, e.g., abort, to occur without becoming irrevocable first. If we do not have direct control of this behavior we must detect and instrument such calls.

2.2 Impact on existing STM algorithms

The immediate impact of sandboxing is that TL2 and its derivatives [6, 7, 8, 21] no longer require post-read validation. Omitting this validation eliminates an ordering constraint in the read barrier in exchange for the possibility of wasted work—continued execution in a transaction that is doomed to abort. Sandboxing also allows these systems to tolerate *privatization* violations without additional barriers [29]. Unfortunately, sandboxing reduces the value of TinySTM-style timestamp extension [26], as a zombie transaction will probably have already used an inconsistent value by the time it tries to validate. We address this issue more thoroughly in Section 5.1 as we develop our prototype sandboxed STM.

3. SANDBOXING INFRASTRUCTURE

Our sandboxing infrastructure consists of the three main components: LLVM-based instrumentation for run-time taint tracking and pre-validation of dangerous operations, POSIX signal interposition and chaining, and timer-based periodic validation.

3.1 LLVM-based Instrumentation

LLVM provides several benefits that make it suitable for sandboxing: (1) it explicitly encodes the dangerous operations that we need to consider without exposing the analysis to low-level details such as stack manipulation; (2) the publicly available Dresden Transactional Memory Compiler (DTMC) and its Tanger instrumentation pass [11] produce instrumented LLVM IR that is ready for analysis and instrumentation; and (3) LLVM's link-time-optimization functionality allows us to perform whole-program analysis and instrumentation, which can result in less conservative instrumentation.

In principle, the only operations of concern are those detailed in Section 2.1. In our current work we assume an *out-of-place* STM system, which keeps speculatively written values in a *redo log* and moves them to regular memory on commit. For an in-place STM (one that keeps an *undo log* of old values to restore on abort), we would need to treat speculative stores as dangerous operations; we leave this for future work.

In practice, we have little direct control over how LLVM's code generator uses the stack. To guarantee the safety of stack-relative addressing, we instrument anything that may update the stack structure in a potentially inconsistent way. Specifically, we instrument allocas if they may be executed on an inconsistent control path, or if their size parameter may be inconsistent. A benefit of aggressively protecting the consistency of the stack is that **return** instructions—technically indirect branches, but not normally mapped by the STM runtime (Section 2)—are not dangerous because the return address on the stack cannot have been corrupted.

Our goal is to instrument all dangerous operations that will execute in an inconsistent context. Identifying such contexts precisely is an information-flow-tracking problem reminiscent of *taint* analysis [35, pp. 558ff], where the values produced by transactional reads are taint sources and the operands of dangerous instructions are taint sinks. Precise taint tracking at compile time is challenging in the presence of aliasing and context sensitivity. We currently employ conservative dynamic tracking and simple static barrier elimination; our evaluation finds that these are mostly adequate.

We start by dynamically maintaining a single-bit taint flag. We extend the STM read barrier to set the flag, and we instrument every dangerous operation with a *validation barrier* that checks the flag and, if it is set, clears it and invokes STM-system-specific validation. Dynamic taint tracking trivially satisfies our requirement that no dangerous operation is performed inconsistently. While the overhead of dynamically checking the flag is small in the common case (a function call, thread-local access, and branch) we would still like to statically eliminate as many redundant barriers as possible.

We currently perform a single static optimization, which we term straight-line redundant validation elimination (SRVE). It exploits the observation that a consistent transaction remains consistent until it performs a transactional read. SRVE tracks the possibility of taint statically, at the basic block level. It initializes each basic block as "possibly inconsistent," and then scans forward. When SRVE encounters a dangerous operation in a possibly inconsistent state, it inserts a validation barrier and changes the state to "consistent." When SRVE encounters a transactional read or function call (SRVE is not context sensitive) it reverts to "possibly inconsistent":

```
SRVE_instrument(BasicBlock bb)
  bool consistent = false;
  foreach Instruction i in bb
    if i is STM_read
      consistent = false
    else if i is function call
      consistent = false
    else if i is dangerous
      if not consistent
        instrument(i)
        consistent = true
```

SRVE is conservative in initializing each basic block as possibly inconsistent. Global analysis could be used to identify basic blocks that are sure to be consistent on entry, but our results have not yet shown the need: SRVE eliminates many of the redundant validation barriers in our benchmarks, and dynamic taint tracking (which remains in place) eliminates most of the work for dynamically redundant barriers that precede any transactional read in their respective basic blocks.

3.2 POSIX Signal Chaining and Validation

We suppress delivery of inconsistent signals using only user space mechanisms. We provide custom signal handlers for all potential faults. These perform validation, and abort if the signal was generated by zombie execution. Otherwise, they forward the signal to a *chained* user handler if one exists, or perform the default action for that signal if it does not. We arrange for any dynamically installed user handlers to fall behind our custom handlers, using libdl-based interposing on signal and sigaction, in a manner reminiscent of the interposition and chaining in Java's libjsig [33] library. We interpose on pthread_sigmask as well in order to restore the correct state during an abort.

We only need to sandbox the synchronous signals distinguished by the libc reference manual [34] as *program error signals*. The remaining signals are asynchronous notifications of events that the program has asked to, or needs to, know about. We believe that user (or default) handlers for these asynchronous signals can be run without regard for the current transactional state of the interrupted execution. These signal handlers are effectively independent threads of execution and thus must be properly synchronized and will be protected from potential zombies with standard transactional mechanisms.

While the details of our signal system are quite complex, they accomplish a straightforward goal: masking out signals that reflect faults in inconsistent transactions, without any kernel assistance, and in a manner that otherwise preserves all standard signal semantics.

3.3 Timer-based Periodic Validation

We guard against inconsistent infinite loops and infinite recursion by installing a timer that triggers periodic validation. This technique is a compelling choice in RSTM [25], which allows dynamic adaptation among numerous STM algorithms, most of which are opaque. Instrumenting loop back edges statically would force those algorithms to pay the back-edge overhead needlessly.

The user application may attempt to use the process-wide POSIX timer functionality, so we must be prepared to interpose on timer-based routines, multiplex library timers with client timers, and use the signal chaining infrastructure described above if required.

Our handler leverages an existing RSTM *epoch* mechanism to detect transactions that have made progress since the last timer event, and uses pthread_kill to trigger validation in those that have not. If all threads have made progress, we reduce the frequency of future validation interrupts. If an interrupted thread detects that it is in an inconsistent infinite loop, we increase the frequency of future interrupts. We set upper and lower bounds on timer frequency at 100Hz and 1Hz, respectively. We also provide a low overhead mechanism to enable and disable timer-based validation on a per-thread basis; this can be used to protect critical, non-reentrant, STM library code, as well as to suppress timer-based aborts during waivered code execution.

4. BASIC SANDBOXING OVERHEAD

Section 3 presents the three components of our sandboxing infrastructure. Though we have yet to develop a formal proof of safety, we believe that these components demonstrate the feasibility of sandboxing in an unmanaged language. Here we examine the cost of our mechanisms to show that sandboxing is a *practical* alternative to (error prone) programmer annotation for use in an out-of-band validation TM.

4.1 Experimental Setup

We run our experiments on a dual-socket system equipped with two 6-core Intel Xeon E5649 processors running Linux 2.6.34. STM runtimes are based on a development version of RSTM, compiled into a highly optimized archive library with gcc-4.7.2, and linked into the benchmarks as native 64-bit libraries. Benchmarks that require sandboxing are compiled using a research version of the DTMC compiler that is compatible with LLVM-2.9. Instrumentation for dangerous operation pre-validation is generated using a custom LLVM pass implementing the SRVE algorithm (Section 3.1).

RSTM provides a set of microbenchmarks that allow us to focus on instrumentation overhead. Specifically, we use the *set* microbenchmark, which performs repeated inserts, lookups, and deletes in sets implemented as lists, hashtables, and red-black trees. This microbenchmark has no dangerous operations, aside from read-only STM_commit operations, but does suffer from inconsistent infinite loops and segmentation faults.

We also use applications from the *STAMP* benchmark suite [23]. STAMP is manually instrumented using a macro-based API that relies heavily on implicitly waivered code, without which the read sets of transactions in many of the included applications get to be so large that the transactions will not terminate. We have added the DTMC equivalent of explicit code waivers where possible to permit us to run the standard test configurations; the results, however, are not directly comparable with other published STAMP results. Our results include the cost of pre-validation and special timer handling for waivered code (Section 3). Conversely, STAMP performs manual checks for inconsistent results. We elide those checks, as they are not necessary in our opaque STMs, and are redundant given sandboxing.

We have been unable to successfully apply transactional waivers to bayes or intruder, and yada results are unreliable, so these three benchmarks are not included in our analysis.

4.2 Results & Analysis

Our signal interposition and chaining implementation consists of a small amount of additional code for each interposed signal handler invocation, including an extra indirect branch and a synchronization operation. Measurements on our Xeon test platform show that this approximately doubles the cost of native signal delivery. While this might in principle impact the performance of an application that depends on the efficiency of small, synchronous signal handlers, we could not measure a negative effect on any of the applications that we tested. In the same vein, an application that is designed to field a large number of signals *during* transactional execution may experience additional overhead from the validation we perform before delivering such signals. In our applications, however, we observed that all synchronous signals that occur in transactions were segfaults resulting from zombie execution.

For programs that use timers, timer-based validation requires SIGALRM and related system call interposition, as well as related multiplexing and demultiplexing code. This could add overhead in applications that depend on the performance of high-frequency timers, but we could not measure an impact on our applications.

Finally, we must consider the cost of the instrumentation for pre-validation of dangerous operations. Pre-validation consists of a lookup and branch on the thread-local taint bit (Section 3.1). This represents real overhead relative to the dangerous operation, which is typically an alloca or a simple store, and can impact the overall performance of a TM system if dangerous operations are common. Of the benchmarks that we tested, only STAMP's labyrinth and genome benchmarks perform substantial numbers of dangerous operations. In each case, these consist of expensive waivered computations that dominate the cost of pre-validation.

These sandboxing mechanisms are entirely thread-local overhead. As such, these results lead us directly to the conclusion that sandboxing is a viable replacement for manual annotation in out-of-band validation TMs, and that the results of Casper et al. and Kestor et al. should hold for general programs.

5. SANDBOXING IN GENERAL

Section 4 evaluated the practicality of sandboxing in the context of out-of-band validation, finding that it effectively replaces manual annotation as a means of ensuring strict serializability. We would like to show that similar results hold for generic buffered update TMs, with in-line validation. To do so we must account more fully for the potential performance degradation resulting from "wasted work" that a zombie transaction performs between the time that it becomes inconsistent and the time that it aborts. With out-of-band validation, wasted work occurs only in systems that poll for notification, and then only in infinite loops and recursions that contain no shared memory accesses (and thus no polling operations). With in-line validation, large amounts of wasted work can, at least in principle, occur in any long-running transaction that performs no dangerous operations (and thus no validations prior to commit).

In order to establish that wasted work is not a problem in practice for buffered update TMs, we develop and test a novel, "maximally lazy" sandboxed STM algorithm that is designed to detect inconsistency as slowly as possible without violating strict serializability.

5.1 Maximally-Lazy Sandboxed STM

Our *maximally lazy STM* descends from the time-based algorithms of Dice et al.'s TL2 [8], Felber et al.'s TinySTM [10], and Dragojević et al.'s SwissTM [9] in their buffered-update forms. It manages concurrency control by mapping individual words of memory to entries in a large table of versioned locks (called ownership records, or *orecs*). Version numbers are taken from a global counter (clock) that is incremented by writer transactions during their commit; the value in an orec indicates the logical time of the most recent update of any word mapping to that orec.

A representative time-based read barrier appears as STM_read_opaque in Figure 1. It identifies and inspects the appropriate orec (Lines 4–7), performs at least one memory

```
1  STM_read_opaque(addr)
2      if addr in write_buffer
3          return value from write_buffer
4      compute orec guarding addr
5      read orec as o1
6      if o1 is locked
7          restart STM_read_opaque
8      FENCE
9      read addr as val
10     FENCE
11     read orec as o2
12     if o2 is locked
13         restart STM_read_opaque
14     if o2 > my_start_time
15         validate
16         set my_start_time to o1
17         restart STM_read_opaque
18     log orec in read_log
19     return val

21 STM_read_sandboxed(addr)
22     if addr in write_buffer
23         return value from write_buffer
24     log addr in read_log
25     read addr as val
26     return val

28 STM_validate_sandboxed()
29     read global time as snapshot
30     for addr in read_log
31         compute orec guarding addr
32         read orec as o
33         while o is locked
34             wait
35         if o > my_start_time
36             abort
37     set my_start_time to snapshot
```

Figure 1: An example read barrier for an opaque STM, and the corresponding sandboxed read/validate decomposition used in our maximally lazy STM.

fence, and validates before returning (Lines 11–17). The use of time-based version numbers provides an opportunity to avoid validation overhead in the absence of recent writes to addr (Line 14) and, in some systems, in the absence of concurrent writer commits (not shown).

Sandboxing allows us to safely return an inconsistent value, and thus to delay both orec identification and validation—in the extreme, until just prior to performing a dangerous operation. Delaying in turn eliminates the need for strong ordering between the reads of an address and its corresponding orec, thereby exposing additional compiler and pipeline parallelism. Psuedocode for this relaxation appears in Figure 1.

Note that we continue to use TinySTM-style timestamp extension, updating my_start_time on a successful validation; however, where TinySTM can recover from reading an inconsistent value, we cannot. TinySTM can validate and re-execute the inconsistent read based on a newly extended my_start_time before returning from STM_read_opaque (Figure 1, Lines 16–17). By contrast, STM_read_sandboxed will return the inconsistent value, which may then be used in the client code, dooming the transaction. We can, however, extend our timestamp after a successful validation (Figure 1, Line 37).

Not shown in Figure 1, but described in Section 2.1, is that the STM_commit_sandboxed routine must validate at the end of a read-only transaction—a validation that its opaque precursors may elide.

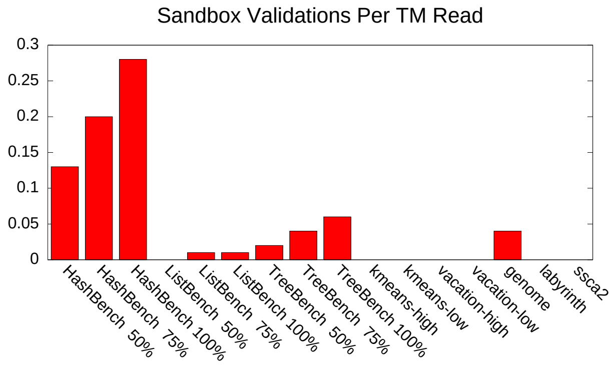

Figure 2: The number of dynamic pre-validation barriers that are executed relative to the number of STM read barriers in our maximally lazy STM. This includes pre-validation of read-only STM_commit_sandboxed routines. The suffixes on microbenchmark names indicate the percentage of read-only transactions in the particular execution.

The cost of STM_read plays a key role in overall transaction latency, since reads are the most common transactional operation. The STM_read_sandboxed barrier performs the minimum possible instrumentation for a buffered-update STM, and thus we expect it to have low overhead. At the same time, it makes long-running zombie transactions possible, and introduces the specter of a decrease in scalability due to potential wasted work, as desired for testing.

5.2 Results and Analysis

Using the same experimental setup as described in Section 4.1, we compare the maximally-lazy, sandboxed STM of Section 5.1 (*Sandbox*) to the *Opaque* counterpart from which it was derived, available as *OrecELA* in the freely available RSTM suite [25]. *Opaque* ensures that a transaction that is doomed to abort discovers this fact as soon as possible by validating each instrumented STM read operation.

Pre-validation Overhead.

Sandbox trades *Opaque's* post-validation of read operations for pre-validation of dangerous operations. We can estimate the impact of the trade simply by looking at the ratio of validations between the two implementations. Intuitively, we expect to eliminate more validations than we introduce; Figure 2 confirms this expectation. In the absence of wasted work, this reduction in the number of validation checks should benefit performance. The RSTM microbenchmarks ("HashBench") have the highest ratios; these reflect the additional validations required at the end of read-only transactions, which in these applications are both numerous and short. Note that the ratios of Figure 2 are suggestive of performance but do not fully determine it: they capture the number of instrumentation points, but not their actual cost (these depend on details of the STM back end).

RSTM set benchmarks.

Figure 3 compares the throughput of the three RSTM microbenchmarks. For these we report results using only one of the two processors: the frantic rate at which transactions are executed in the microbenchmarks causes RSTM's global

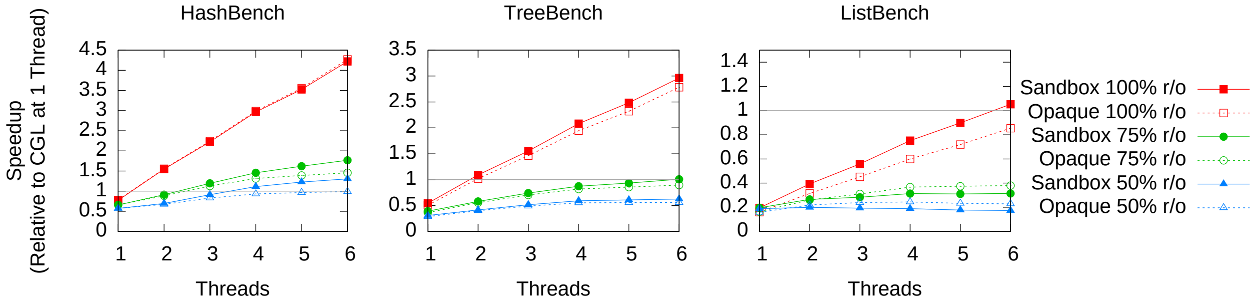

Figure 3: Our maximally lazy STM compared to an equivalent opaque STM on the RSTM set microbenchmarks. Lines are labeled with the percentage of read-only transactions that are performed. Throughput is shown as speedup relative to coarse-grained-lock throughput at 1 thread for the respective read ratio.

commit counter to become an unacceptable bottleneck once cross-chip communication is required.

Lines are labeled with the percent read-only transactions that they perform. This number is important, as *Sandbox* performs a commit-time validation for read-only transactions that *Opaque* does not. The results of these tests are encouraging and consistent with our expectations given Figure 2. The relative number of validation barriers is a good predictor of a reduction in the overhead of *Sandbox*, and we see few ill effects from wasted work.

The hashtable set consists of tiny, CPU-bound transactions, where the cost of validation is quite low. *Sandbox* suffers from negligible wasted work in this case, and thus its performance relative to *Opaque* is entirely predictable. Read-only performance is indistinguishable, and as writers become more common, *Opaque* transactions must validate more frequently than their *Sandbox* counterparts, resulting in slightly less scalable performance.

The tree-based benchmark contains longer, larger transactions, as well as cases where rotation provides large asymmetries. The linked structure is an interesting test of our sandboxing infrastructure as it suffers from both inconsistent SEGFAULTs and infinite loops. *Sandbox*'s reduced validation overhead results in an advantage for read-only execution, but its inability to perform in-barrier timestamp extension (Figure 1, Lines 14–17) results in more frequent aborts as writers become more common.

The list microbenchmark evaluates performance for large transactions. Large, non-conflicting transactions are an ideal workload for *Sandbox*, which simply performs a single validation barrier at commit time, while *Opaque* must execute a barrier for each node read during the search. This results in substantially less overhead for *Sandbox*, as well as better scalability when conflicts are infrequent. On the other hand, writer transactions trigger high abort rates that *Opaque* tolerates better due to its early conflict detection. *Sandbox* suffers from large amounts of wasted work in this case. It must be noted that neither runtime performs well in this common-conflict setting, but RSTM's ability to dynamically adapt between opaque and sandboxed execution (not exploited in these experiments) may prove a valuable feature.

STAMP Benchmarks.

RSTM's microbenchmarks are intended as a stress test for STM systems: their performance depends almost entirely on the STM infrastructure. On the other hand, STAMP's large scale and reliance on waivered execution means that performance is much less sensitive to details of the STM algorithm (though still heavily dependent on abort rates). Figure 3 shows results on five of the eight STAMP benchmarks as described in Section 4.1. RSTM's bottlenecks are not a factor with more reasonable rates of transactional execution, so we test *Opaque* and *Sandbox* using native Linux scheduling out to the full 12 cores of the machine.

STAMP's large transactions raise the specter of reduced scalability due to wasted work, but we see no evidence of this in the results, where *Opaque* and *Sandbox* are generally very well matched. *Sandbox* outperforms *Opaque* at one thread due to its lower overheads and continues to perform well out to 12 threads in most cases. *Sandbox* suffers a somewhat higher abort rate than *Opaque* due to its reduced ability to do timestamp extension. This manifests as lower performance in labyrinth, whose enormous transactions lead to a very high abort penalty.

Overall, the performance of *Sandbox* is at least on par with that of *Opaque* when parallelism is available, and often slightly better. This result demonstrates that, for the applications that we test, wasted work is unlikely to negatively impact performance. As such, it is reasonable to conclude that our sandboxing infrastructure is a viable option for validation in buffered update TMs in general.

6. CONCLUSIONS

We have shown that sandboxed STM in an unmanaged language is both possible and practical. Our results with maximally lazy sandboxing further suggest that wasted work is unlikely to be a real-world problem. Given the appeal of out-of-band validation for both hardware and software TM, we conclude that sandboxing is a promising avenue toward increased performance in future TM systems. Our work should enable researchers to explore this design space without concern.

Future Work.

Our general notion of dangerous operations and sandboxing semantics must be formalized to prove that we have not overlooked any cases where instrumentation is necessary. At the same time, formalization is likely to ignore certain practical interactions between the program and the system —operating system accounting, performance counters, debugging, etc.—in which zombie execution may be visible.

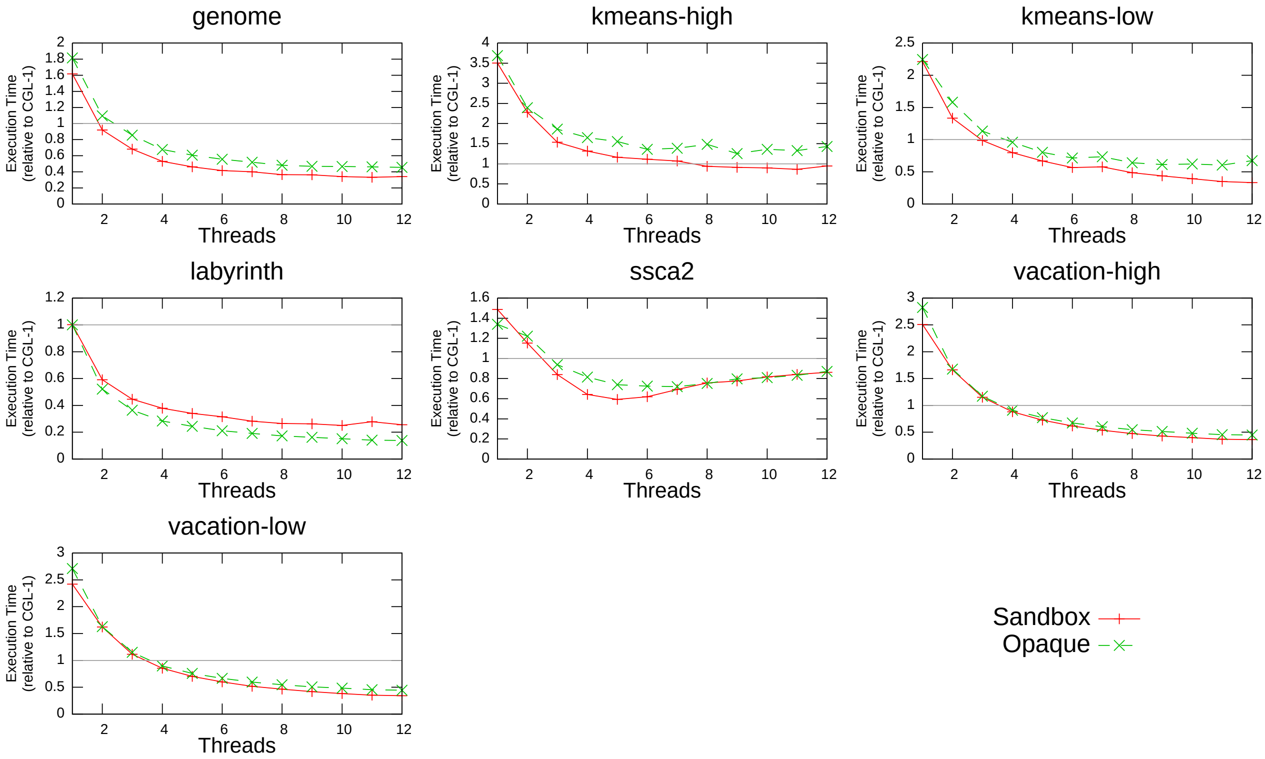

Figure 4: Our lazy STM compared to an equivalent opaque STM for five STAMP benchmarks. Throughput is shown as execution time relative to that of coarse-grained-lock at 1 thread.

Working in the context of these interactions remains a topic of research.

The results presented here should be considered indicative of likely performance only for buffered-update STMs, in which transactional writes are not considered dangerous. For in-place systems, the number of sandboxing validation barriers will be substantially larger. The performance of sandboxing in such a setting will be more strongly contingent on the quality of the pre-validation instrumentation algorithm.

Along this line, there is a clear opportunity to implement more expensive static analysis to attempt to eliminate always-redundant instrumentation on dangerous operations. This could include full information-flow tracking as well as potential code cloning and specialization to partition occasionally-redundant instrumentation points into always-redundant and necessary pairs.

And of course, we encourage researchers to aggressively explore the space of concurrent-validation TM, both in software and in hardware.

Acknowledgments.

Our thanks to: Patrick Marlier for his help in understanding the DTMC/TinySTM code base, as well as his implementation of the DTMC shim for RSTM used in our evaluation; Michael Spear for his assistance in the development of our sandboxed STM, as well as contributing some text and pseudocode in Section 5.1; Martin Nowak for his help in delivering a DTMC compiler compatible with LLVM-2.9; and Gökçen Kestor for her assistance with benchmarking. Finally, we thank our reviewers for their careful reading of our original draft and their resulting valuable advice.

References

[1] Draft Specification of Transaction Language Constructs for C++. Version 1.0, IBM, Intel, and Sun Microsystems, Aug. 2009.

[2] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and Runtime Support for Efficient Software Transactional Memory. *SIGPLAN 2006 Conf. on Programming Language Design and Implementation*, June 2006.

[3] J. Casper, T. Oguntebi, S. Hong, N. G. Bronson, C. Kozyrakis, and K. Olukotun. Hardware Acceleration of Transactional Memory on Commodity Systems. *16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Mar. 2011.

[4] 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. *16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Mar. 2011.

[5] L. Dalessandro, M. L. Scott, and M. F. Spear. Transactions as the Foundation of a Memory Consistency Model. *24th Intl. Symp. on Distributed Computing*, Sept. 2010. Earlier but expanded version available as TR 959, Dept. of Computer Science, Univ. of Rochester, July 2010.

[6] D. Detlefs. Unpublished manuscript, 2007.

[7] D. Dice and N. Shavit. TLRW: Return of the Read-Write Lock. *4th ACM SIGPLAN Workshop on*

Transactional Computing, Feb. 2009.

[8] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. *20th Intl. Symp. on Distributed Computing*, Sept. 2006.

[9] A. Dragojević, R. Guerraoui, and M. Kapałka. Stretching Transactional Memory. *SIGPLAN 2009 Conf. on Programming Language Design and Implementation*, June 2009.

[10] P. Felber, T. Riegel, and C. Fetzer. Dynamic Performance Tuning of Word-Based Software Transactional Memory. *13th ACM Symp. on Principles and Practice of Parallel Programming*, Feb. 2008.

[11] P. Felber, E. Rivière, W. M. Moreira, D. Harmanci, P. Marlier, S. Diestelhorst, M. Hohmuth, M. Pohlack, A. Cristal, I. Hur, O. S. Unsal, P. Stenström, A. Dragojevic, R. Guerraoui, M. Kapalka, V. Gramoli, U. Drepper, S. Tomić, Y. Afek, G. Korland, N. Shavit, C. Fetzer, M. Nowack, and T. Riegel. The Velox Transactional Memory Stack. *IEEE Micro*, 30(5): 76–87, Sept.-Oct. 2010.

[12] K. Fraser. Practical Lock-Freedom. Ph. D. dissertation, UCAM-CL-TR-579, Computer Laboratory, Univ. of Cambridge, Feb. 2004.

[13] R. Guerraoui and M. Kapałka. On the Correctness of Transactional Memory. *13th ACM Symp. on Principles and Practice of Parallel Programming*, Feb. 2008.

[14] R. Guerraoui and M. Kapałka. *Principles of Transactional Memory.* Morgan & Claypool, 2010.

[15] T. Harris and K. Fraser. Language Support for Lightweight Transactions. *OOPSLA 2003 Conf. Proc.*, Oct. 2003.

[16] T. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory.* Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2nd edition, 2010.

[17] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing Memory Transactions. *SIGPLAN 2006 Conf. on Programming Language Design and Implementation*, June 2006.

[18] M. Herlihy and J. E. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. *20th Intl. Symp. on Computer Architecture*, May 1993. Expanded version available as CRL 92/07, DEC Cambridge Research Laboratory, Dec. 1992.

[19] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. *22nd ACM Symp. on Principles of Distributed Computing*, July 2003.

[20] G. Kestor, R. Gioiosa, T. Harris, O. S. Unsal, A. Cristal, I. Hur, and M. Valero. STM^2: A Parallel STM for High Performance Simultaneous Multithreading Systems. *2011 Intl. Conf. on Parallel Architectures and Compilation Techniques*, Oct. 2011.

[21] V. J. Marathe, M. F. Spear, and M. L. Scott. Scalable Techniques for Transparent Privatization in Software Transactional Memory. *2008 Intl. Conf. on Parallel Processing*, Sept. 2008.

[22] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Practical Weak-Atomicity Semantics for Java STM. *20th ACM Symp. on Parallelism in Algorithms and Architectures*, June 2008.

[23] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. *2008 IEEE Intl. Symp. on Workload Characterization*, Sept. 2008.

[24] M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. *16th Intl. Conf. on Parallel Architectures and Compilation Techniques*, Sept. 2007.

[25] Reconfigurable Software Transactional Memory Runtime. Project web site. `code.google.com/p/rstm`.

[26] T. Riegel, C. Fetzer, and P. Felber. Time-based Transactional Memory with Scalable Time Bases. *19th ACM Symp. on Parallelism in Algorithms and Architectures*, Jun 2007.

[27] T. Riegel, C. Fetzer, and P. Felber. Snapshot Isolation for Software Transactional Memory. *1st ACM SIGPLAN Workshop on Transactional Computing*, June 2006.

[28] 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. *11th ACM Symp. on Principles and Practice of Parallel Programming*, Mar. 2006.

[29] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization Techniques for Software Transactional Memory (brief announcement). *26th ACM Symp. on Principles of Distributed Computing*, Aug. 2007. Extended version available as TR 915, Dept. of Computer Science, Univ. of Rochester, Feb. 2007.

[30] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. Ordering-Based Semantics for Software Transactional Memory. *12th Intl. Conf. on Principles of Distributed Systems*, Dec. 2008.

[31] M. F. Spear, M. M. Michael, M. L. Scott, and P. Wu. Reducing Memory Ordering Overheads in Software Transactional Memory. *Intl. Symp. on Code Generation and Optimization*, Mar. 2009.

[32] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict Detection and Validation Strategies for Software Transactional Memory. *20th Intl. Symp. on Distributed Computing*, Sept. 2006.

[33] `libjsig`: Java$^{\text{TM}}$RFE 4381843. Online documentation. `docs.oracle.com/javase/1.5.0/docs/guide/vm/signal-chaining.html`.

[34] The GNU C Library. Online documentation. `www.gnu.org/s/hello/manual/libc/index.html`.

[35] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl.* O'Reilly Media, Third Edition, 2000.