Sandboxing Transactional Memory *

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

Abstract

In a transactional memory (TM) system, *conflict detection* ensures that transactions never commit unless they can be serialized. *Validation* further ensures that a doomed transaction (one that has seen an inconsistent view of memory) discovers its status and aborts before its inconsistency has a chance to cause erroneous, externally visible behavior.

In an *opaque* TM system, post-validation of reads ensures that the view of memory is always consistent. In a *sandboxed* TM system, pre-validation of potentially dangerous operations ensures that inconsistency is harmless. Sandboxing is straightforward in managed languages, but its feasibility for unmanaged languages has been questioned and remains an open problem.

In this paper we show that it is both possible and practical to implement sandboxing for C and C++ TM systems, with overheads (including wasted work) no worse than those of opaque implementations. Specifically, we present a software TM infrastructure using LLVM, the Dresden TM Compiler, and a novel sandboxing runtime, and describe its performance on a variety of benchmarks. Systems such as ours will, we believe, enable the development of new and faster TM algorithms (in both hardware and software) that depend on sandboxing for correctness—e.g. in conjunction with asynchronous (out-of-band) validation.

1. Introduction

All speculation-based TM systems require a mechanism to detect and resolve potential conflicts between transactions, thereby ensuring that transactions commit only as part of a global serialization order. Additionally, committed transactions must obey the semantics of the source programming language. A transaction that is going to abort may, in principle, do anything at all, provided it has no impact on observable program behavior. We may regard this observation as either cause or effect: a transaction may diverge from correct behavior if we know it is going to abort; alternatively, a transaction may be forced to abort because it has diverged from correct behavior.

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

In most TMs, inconsistency results from reading a pair of values that were not 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 transactions to abort if their view becomes inconsistent. To this end, STM runtimes typically *validate* transactions by double-checking their read sets at regular intervals, and aborting them in the event of inconsistency. At one extreme, validation may be performed conservatively after every transactional read. At the other, it may be performed immediately before any "dangerous" event—one whose behavior cannot be guaranteed to be invisible to other threads. A complete characterization of dangerous events is tricky (see Section 2.1); at the very least we must guard against faults, infinite loops, and stores to incorrectly computed addresses.

Guerraoui and Kapałka hold that aborted transactions are semantically relevant and should see a consistent view of memory, a property they term *opacity* [13, 14], and present it as essential to correctness. In an alternative view, we have argued [5, 30] that language semantics should speak only to committed transactions, and that speculation and aborted transactions should be a matter of implementation. This view requires only that a speculative implementation protect the rest of the system from possible errant behavior in transactions that ultimately abort. We refer to an implementation that meets this requirement but that is not opaque as *sandboxed*. This definition includes the techniques described by Spear et al. [32] for optimizing STM barriers, however here we will be concerned with much more relaxed algorithms.

Sandboxing is common in transactional runtimes for managed languages, which often execute untrusted code in the context of some larger system and thus both restrict programmer access to some problematic operations and naturally provide mechanisms for dealing with others. STMs for Java™ and C#, along with those built on top of dynamic binary instrumentors, have relied on sandboxing to avoid the need for conservative validation in running transactions [2, 16, 24].

In contrast, the designers of STMs for unmanaged languages were slow to appreciate the dangers of inconsistency, and have largely ignored the potentials for sandboxing. Some TM algorithms simply overlook the need for con-

sistent execution altogether [3]. Others depend on explicit programmer-inserted validation for correct execution [12, 15, 17, 28]. Still others guarantee opacity via repeated, incremental validation [18]. Later work leveraged knowledge of recent commits by other threads [8, 26, 33] or the presence of static redundancy [32] in an attempt to minimize the common-case quadratic overhead inherent in incremental validation, but remain quadratic in the worst case.

To the best of our knowledge, no previous project has carefully explored the feasibility of transaction sandboxing in unmanaged languages. (Some authors have even implied that it might not be possible [22].) There are compelling reasons to do so.

First, given that dangerous events (i.e., those that may result in errant behavior when executed inconsistently) tend to be much less common than transactional loads, sandboxing could substantially reduce the overhead of validation in running transactions. This may, in some cases, lead to wasted execution in doomed ("zombie") transactions, but we expect the impact of such delays to be modest, particularly given the assumption that abort rates will be low in scalable applications.

Second, and arguably more important, sandboxing makes it possible to perform validation in parallel with execution of the main program, thereby removing the overhead of validation from the application's critical path. Kestor et al. [20] consider this approach for multithreaded processors and obtain significant improvements. (Significantly, while they recognize the need for sandboxing, they do not actually implement it in their prototype; instead they limit their experiments to applications where nothing goes wrong.) Casper et al. [3] consider a similar mechanism, with dedicated hardware support, but without addressing the correctness issues it introduces. The appeal of sandboxing for HTM systems is that it may allow a looser coupling 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].

In Section 2 we characterize potentially dangerous events for zombie transactions in C and 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. Given this basic infrastructure, in Section 4 we develop a proof-of-concept, sandboxed STM using the open-source RSTM package [25]. We evaluate our sandboxing infrastructure and our novel STM on an assortment of benchmarks in Section 5, and conclude in Section 6. The principal conclusion is quite positive: sandboxing of unmanaged transactions is both feasible and efficient. In comparison to opacity, sandboxing leads to reductions in instrumentation overhead for a wide range of transactional benchmarks, and the measured increases in wasted work are comparatively modest. This in turn suggests that out-of-band validation is a viable strategy for both hardware and software TM systems, and that sandboxing should be a standard feature in future TM compilers.

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].

If aborted transactions play no role in language semantics, then we are free to consider an aggressively optimistic implementation that does not guarantee opacity, but simply SS for committed transactions. In such an implementation, sandboxing must ensure the isolation of even those "zombie" transactions that have diverged from correct execution due to an inconsistent view of memory (i.e., a read that cannot be explained by any serial execution of transactions [17]), but it need not restrict the internal behavior of such transactions beyond this constraint.

2.1 Dangerous Events

We say that an event 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 events (a formal proof of completeness is deferred to future work).

In-place stores Stores performed by a transaction to a public address, rather than a private log, are the primary concern for zombie transactions in a sandboxing runtime. In-place stores may target shared locations, in which case they may be instrumented (e.g., in an eager, in-place STM) or uninstrumented (e.g., if the compiler or programmer has concluded that there are no TSC executions where a particular store can be part of a race). They may also target private locations in the stack or other thread-local storage, in which case they may lead to some subsequent violation of sequential semantics.

In-place stores may be inconsistent because they are control dependent on an inconsistent value or because their target address is inconsistent. Such stores may create races, not be rolled back on abort, or even result in execution of arbitrary code. Storing an inconsistent value to a consistent address is not fundamentally unsafe, but has implications for

analysis because the value returned from an uninstrumented read of the stored location must be correctly detected as inconsistent. Our current analyses cannot track inconsistency through memory thus we treat this case as dangerous.

In-place stores that cannot be proven consistent must be handled with a run-time, pre-validation check. The STM implementations that we develop and test in this work all have *out-of-place* write barriers, a fact that we exploit in our analyses and testing. Sandboxing for in-place STMs requires that we treat `STM_write` barriers as in-place writes, and is left as future work.

Pre-validation exposes an additional concern. An inconsistent indirect branch may lead to an unprotected in-place store, or to executable data that *looks like* one. Such branches are used by programming languages to implement virtual function calls, large switch statements, computed `goto`s, and `return`s, and can represent large overheads. Fortunately, existing opaque STM implementations already perform the necessary instrumentation for indirect function calls. Such runtimes must map 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—a transition that includes an embedded validation.

Faults The sandboxing runtime must disambiguate faults (hardware exceptions) that occur due to inconsistent execution from those that would have occurred in some TSC execution of the program, and prevent their effects from becoming visible to the programmer. In managed languages, where such events are often exposed to the programmer through a software exception handling framework, run-time systems can validate before delivering the exception to the user.

In a POSIX-compliant C and C++ implementation, faults are encoded as synchronous signals. Olszewski et al. [24] suggest either suppressing inconsistent signals by modifying the operating system kernel to make it transaction aware—which they implement—in order to prevent the signal from propagating to user space, or rely on user-space signal handling functionality to suppress the signal once received.

Infinite Loops and Recursion A zombie transaction may enter into an infinite loop or infinite recursion due to inconsistent execution which a sandboxing runtime must be able to detect and recover from. Previous sandboxed STMs 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 that pollutes hardware resources like branch predictors.

An alternative approach, assuming appropriate OS support, is to use timers to force the STM to validate periodically. This avoids common-case overhead, and the timer period can be adaptively tuned so 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 their last read barrier. This assumption isn't valid in a sandboxed TM, which must ensure that read-only transactions have validated before committing.

2.2 Impact on existing STM algorithms

The immediate result of sandboxing is that TL2 and its derivatives no longer require post-read validation [6–8, 21]. This eliminates an ordering constraint in the read barrier in exchange for the possibility of wasted work. Sandboxing also allows these systems to tolerate privatization violations without necessitating additional barriers [31]. Unfortunately, sandboxing reduces the value of TinySTM-style timestamp extension [27], 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 4 as we develop our example, sandboxed STM.

3. Sandboxing Infrastructure

Our sandboxing infrastructure consists of the three main components: LLVM-based instrumentation, Posix signal chaining, and timer-based periodic validation.

3.1 LLVM-based Instrumentation

LLVM and its associated IR provide several benefits that make it suitable for sandboxing.

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

In principle, the only operations of concern are those detailed in Section 2.1. In practice, we have little direct control of how LLVM's code generator uses the stack. To guarantee the safety of `sp`- and `fp`-relative addressing, we must instrument anything that may update these registers in a potentially inconsistent way. Specifically, we instrument `alloca`s if they may be executed on an inconsistent control flow 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—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 [36, pp. 558ff], where the values produced by transactional reads are taint sources and the operands of dangerous instructions are taint sinks. The cost of tracking precise taint in the presence of aliasing and context sensitivity is high. We use conservative approximations and our evaluation finds that these are mostly adequate.

We start by dynamically tracking taint using a single bit of data. We extend the STM read barrier to mark the transaction as tainted, and we instrument every dangerous operation with a validation barrier that checks the tainted status and then calls the underlying algorithm-specific validation routine if needed. This conservative dynamic approach trivially satisfies our requirement that no in-place store is performed inconsistently. While the overhead of this barrier 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.

The single optimization that we perform is straight-line redundant validation elimination (SRVE), which hinges on the observation that a consistent transaction remains consistent until it performs a transactional read. SRVE tracks taint statically, at the basic block level. It initializes each basic block as tainted, and then scans forward. When SRVE encounters a dangerous operation in a tainted state it inserts a validation barrier and clears the tainted bit. When SRVE encounters an instrumented read or function call (SRVE is not context sensitive) it sets the tainted flag.

```
SRVE_instrument(BasicBlock bb)
  bool tainted = true;
  foreach Instruction i in bb
    if i is STM_read
      tainted = true
    else if i is function call
      tainted = true
    else if i is dangerous
      if tainted
        instrument(i)
        tainted = false
```

SRVE is conservative in initializing each basic block as tainted. Global analysis could be used to identify basic blocks that are clean on entry, however our results show that SRVE eliminates most of the redundant validation barriers in our benchmarks so we are not compelled to implement anything more costly yet.

3.2 POSIX Signal Chaining and Validation

We seek to implement signal sandboxing at run time without the aid of a dynamic binary translator, which requires some careful software engineering. First, we provide custom handlers for the necessary signals that perform validation and abort if the signal was generated by zombie execution. In the case that this signal is the result of consistent execution we forward the signal to a *chained* user handler if one exists, or perform the default action for that signal if it does not. We prevent users from overwriting our handlers using dynamic signal chaining interposition techniques in much the same way as Java™'s `libjsig` [34] library, using `libdl`-based interposing on `signal` and `sigaction`.

POSIX requires the use of an alternate stack in order to handle a `SIGSEGV` resulting from a stack overflow. In the absence of an alternate stack the default `SIGSEGV` handler is used without regard for potential user-registered handlers. If the TM user has not chained a `SIGSEGV` handler, or has not specified an alternate stack for `SIGSEGV` execution, we simply emulate the required terminate-and-core-dump for a stack overflow. Unfortunately, such a core dump will show that the sandboxing handler was running on an alternate stack which would be inconsistent with the user's expectation and violate the sandbox. At this time we simply require that, if the user registers a `SIGSEGV` handler, then they must do so with an alternate stack—even if they do not expect to successfully handle stack overflows.

We only need to sandbox the synchronous signals distinguished by the `libc` reference manual [35] 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.

3.3 POSIX Timer-based Periodic Validation

We guard against inconsistent infinite loops and infinite recursion by installing a POSIX timer that triggers periodic validation. This technique is a compelling choice in RSTM, 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.

Implementing the timer-based approach requires careful software engineering. The user application may attempt to use the process-wide POSIX timer functionality, so we must be prepared to interpose timer-based routines, multiplex library timers with client timers, and use the signal chaining infrastructure to call chained handlers if required.

Our handler leverages an existing RSTM transactional-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

```
volatile uintptr_t global_time
__thread uintptr_t start_time
__thread WriteSet writes
__thread ReadLog reads
__thread LockLog locks
__thread int cursor

do_lazy_hashing()
  if (cursor == reads.size()) return true
  while (cursor < reads.size())
    addr = reads[cursor].orec
    reads[cursor].orec = &orecs[hash(addr)]
    cursor++
  return false

STM_write(addr, val)
  writes.log(addr, val)

STM_read(addr)
  if (found = writes.find(addr)) return found
  reads.log(addr)
  return *addr

STM_validate()
  if (start_time != global_time)
    if (do_lazy_hashing() != true)
      snapshot = global_time
      foreach read in reads
        if (read.orec→time > start_time)
          abort
      start_time = snapshot

STM_begin()
  cursor = 0
  start_time = global_time

revert_locks_and_abort()
  foreach lock in locks
    lock.revert()
  abort

STM_commit()
  if (writes.empty())
    STM_validate()
    return
  foreach write in writes
    if (write.orec→time > start_time)
      if (write.orec→lock != my_lock)
        revert_locks_and_abort()
    if (!write.acquire_with(my_lock))
      revert_locks_and_abort()
    locks.log(write)
  end_time = fetch_and_increment(global_time) + 1
  if (end_time != start_time + 1)
    do_lazy_hashing()
    foreach read in reads
      if (read.orec→time > start_time)
        if (read.orec→lock != my_lock)
          revert_locks_and_abort()
  foreach write in writes
    *write.addr = write.value
  foreach lock in locks
    lock.orec→time = end_time
```

Figure 1. Simplified implementations of the relevant parts of our sandboxed STM. Many details are suppressed for clarity (not shown for instance: privatization, subword accesses, adaptivity, epochs, log maintenance, etc). The **abort** routine abstracts the non-local control flow associated with aborting a transaction, often implemented as `longjmp`.

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.

4. A Sandboxed STM

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. Using this infrastructure we have develop a novel, sandboxed STM algorithm that 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, using Marathe et al.'s two-counter commit protocol [21] to provide privatization safety. A simplified version of the algorithm appears in Figure 1.

While traditional, time-based `STM_read` barriers compute and inspect an orec and perform at least one memory fence, our runtime merely performs a read-after-write check, logs the address and returns the current value at this address. The simplicity results from our ability to lazily convert addresses into orec locations at validation time—a task we accomplish by keeping a `cursor` that indicates of the next read log entry that must be hashed. This `cursor` serves a second purpose in `STM_validate` as the `tainted` bit for conservative dynamic filtering (Section 3.1). Lazy hashing, combined with standard time-based filtering, eliminates the need to compute hashes during single-threaded execution, resulting in lower single-threaded overhead than comparable opaque systems. The lack of ordering constraints or volatile accesses in the `STM_read` barrier contributes further reductions in overhead.

We continue to use TinySTM-style timestamp extension, updating our `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 the new `start_time` before returning from the `STM_read` barrier. Our proposed STM will return the inconsistent value which may then be used in the client code, dooming the transaction.

5. Performance Evaluation

The goal of our performance evaluation is twofold. First, we show that the cost of the sandboxing infrastructure (i.e., the

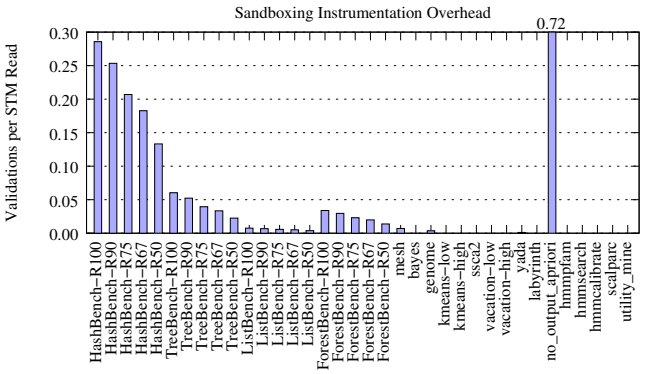

Figure 2. The number of dynamic validations that are required relative to the number of STM read barriers.

dangerous operation instrumentation and timer-based validation) is comparable to that of an opaque STM. Second, we show that zombie transactions do not waste a troublesome amount of work, even in an STM that is strongly susceptible to such wasted work.

All of our evaluation results are performed using a 6-core Xeon^{TM}E5649 processor running Linux 2.6.34. All benchmarks are compiled using a research version of the DTMC compiler that is compatible with LLVM-2.9. Sandboxing instrumentation is generated using a custom LLVM pass implementing the SRVE algorithm (Section 3.1). The STM runtimes are based on a development version of RSTM, compiled into a highly optimized archive library with gcc-4.6.2, and linked into the benchmarks as native 64-bit libraries.

RSTM contains a set of microbenchmarks that allow us to focus our evaluation on the performance of STMs in specific conditions. We use the *set* microbenchmark that 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 inconsistent `SIGSEGV` signals. The RSTM *mesh* application performs Delaunay mesh triangulation using transactions [29]. It uses transactions sparingly but does contain dangerous operations. We also use Minh et al.'s *STAMP* benchmark suite [23], patched to support compilation and instrumentation with DTMC, and Kestor et al.'s *RMS-TM* benchmark suite [19], containing transactionalized versions of benchmarks in the recognition, mining, and synthesis spectrum.

5.1 Sandboxing Infrastructure

Figure 2 compares the dynamic number of validation barriers when pre-validating dangerous operations to the number when post-validating every STM read operation. The $-R$ suffixes in the RSTM microbenchmark name indicate the percentage of read-only transactions in the particular execution. Here we see that sandboxing has the potential to reduce, sometimes dramatically, the number of validation barriers performed during an execution. In contexts with a large number of short and/or read-only transactions, sandboxing's forced validation at STM commit results in much higher than average validation rate, maxing out at about 0.28 per read for the read-only `HashSet` benchmark. In general, the rate is much lower, and we often find that transactions contain no dangerous operations and thus perform no validations aside from those required at writer commit for both sandboxing and opaque implementations.

The single outlier is RMS-TM's `no_output_apriori` where our SRVE analysis fails to effectively eliminate redundant validations, which make up 75% of the dynamic validations encountered. This suggests that a better static analysis may be important in a production environment. Note however that dynamic filtering means that the only cost incurred during a redundant validation is that of a function call and branch.

The overhead of timer-based validation, given the upper and lower frequency bounds of 100Hz and 1Hz, respectively, is so small that we cannot measure it.

5.2 Wasted Work

In an opaque STM implementation, a transaction that is doomed to abort discovers this fact as soon as possible by validating each instrumented STM read operation. The nature of a sandboxed implementation is that a doomed transaction will continue to run for some period of time before detecting that it is inconsistent. The work performed in this zombie state is known as *wasted work*.

While our sandboxing infrastructure provides a hard upper bound on the potential amount of wasted work due to its periodic validation, the real upper bound will be STM algorithm specific, and the actual amount will depend on the application. In particular, a principal motivation for sandboxing is to shift the overhead of validation off the critical path and into an available coprocessor [3, 20]. In this case wasted work can be expected to be inconsequential: zombie status will be detected "very soon."

Nevertheless, we would like to evaluate the performance of the sandboxed STM detailed in Section 4, to assess the impact of wasted work in this "maximally lazy" case—and of course to verify that our sandboxing infrastructure actually works. Figure 3 compares the throughput of our sandboxed STM (OrecSandbox) to that of an equivalent, privatization safe, opaque STM (OrecELA) using four RSTM microbenchmarks. Throughput numbers are averaged across five runs and normalized to those of a coarse-grained-lock STM implementation at 1 thread (CGL). Again, the $-R$ suffixes in the legend indicate the percentage of read-only transactions in the particular execution.

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

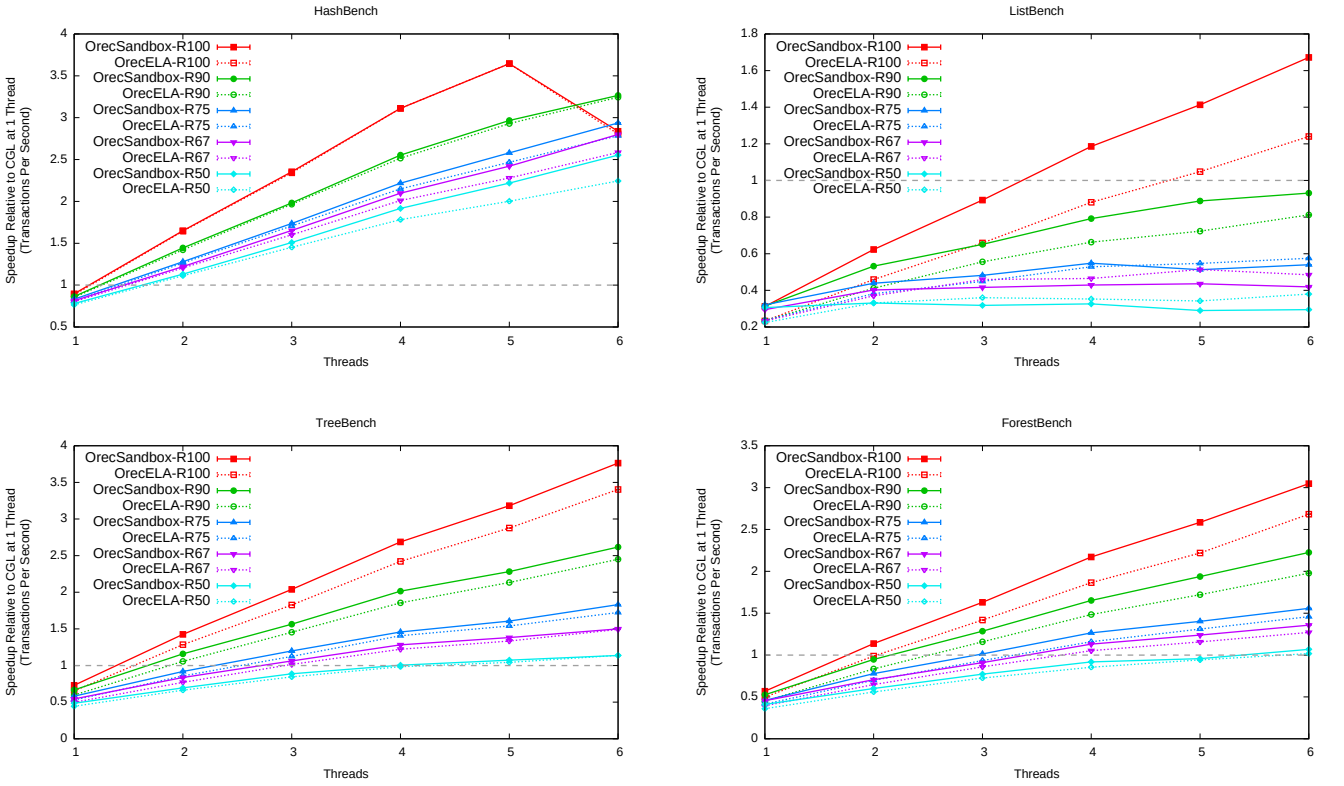

Figure 3. Our sandboxed STM compared to an equivalent opaque STM on the RSTM set microbenchmarks. Throughput is shown as speedup relative to coarse-grained-lock throughput at 1 thread.

overhead for OrecSandbox, 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. OrecSandbox suffers from negligible wasted work in this context, and thus its performance relative to OrecELA is entirely predictable. Read-only transactions typically read a single location, thus the single validation done by OrecELA is directly balanced by the single validation done during `STM_commit` in OrecSandbox. OrecELA occasionally preforms multiple validations per transaction when it encounters a chained bucket that must be followed. As writers become more common, OrecELA transactions must validate more frequently than their OrecSandbox counterparts resulting in slightly less scalable performance. The baseline OrecELA implementation contains a scalability bottleneck that manifests at six cores—mirrored by OrecSandbox—which we are still investigating.

The two tree-based benchmarks contain longer, larger transactions, as well as cases where rotation provides large asymmetries. These cases are interesting tests of our sandboxing infrastructure as they both suffer from inconsistent `SEGFAULT`s and infinite loops. Nonetheless, relative performance again tracks the results predicted by Figure 2. OrecSandbox consistently outperforms its opaque counterpart in these conditions.

The list microbenchmark evaluates performance in the context of large, as well as highly contended, transactions. Large, non-conflicting transactions are an ideal environment for our OrecSandbox implementation, which simply performs a single validation barrier at commit time, where the OrecELA implementation must perform the barrier for each node read during the search. This results in substantially less overhead for the OrecSandbox runtime, as well as better scalability when conflicts are infrequent. On the other hand, writer transactions trigger high abort rates that OrecELA tolerates better due to its opaque validation. OrecSandbox suffers from large amounts of wasted work in this context. It must be noted that neither runtime performs well in this common-conflict setting, however it suggests that RSTM's ability to dynamically adapt between opaque and sandboxed execution is a valuable feature.

These microbenchmark results show the potential of sandboxed STM. We were unable to get stable results for parallel execution of STAMP and RMS-TM at this time, but intend to continue this investigation and anticipate that Figure 2 will continue to provide accurate predictions for OrecSandbox performance.

6. Conclusions

We have shown that the infrastructure required to support a sandboxed STM in an unmanaged language is both possible and practical. Furthermore our positive preliminary results from testing a novel sandboxed STM indicate that the development of sandboxed STMs may be a promising avenue toward improved STM (and HTM) performance—particularly if sandboxing is used to enable out-of-band validation. We believe that this work will enable future 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 important cases where instrumentation is necessary. At the same time, formalization is likely to ignore certain practical interactions that the program has with the system, e.g., operating system accounting, performance counters, debugging, etc., where zombie execution will become visible. Working within such a setting remains a topic of research.

The analysis presented here is valid only for buffered-update STMs due to the fact that we are not considering STM write barriers to be dangerous. For in-place systems, `STM_write` barriers must be instrumented and thus the number of sandboxing validation barriers will be substantially larger. This is more than simply an academic issue, as sandboxed STMs such as the one presented by Kestor et al. [20] depend on in-place stores for their performance.

The opportunity exists 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.

Our microbenchmarks imply that sandboxed STMs should perform well in low-contention conditions with moderately-sized transactions, however more testing is needed to verify that this is the case.

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; Martin Nowak for his help in delivering a DTMC compiler compatible with LLVM-2.9; and Gökçen Kestor for assistance with the RMS-TM benchmark suite. In addition, 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. In *Proc. of the SIGPLAN 2006 Conf. on Programming Language Design and Implementation*, pages 26–37, 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. In *Proc. of the 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. In *Proc. of the 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. In *Proc. of the 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. In *4th ACM SIGPLAN Workshop on Transactional Computing*, Feb. 2009.

[8] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proc. of the 20th Intl. Symp. on Distributed Computing*, pages 194–208, Sept. 2006.

[9] A. Dragojević, R. Guerraoui, and M. Kapałka. Stretching Transactional Memory. In *Proc. of the 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. In *Proc. of the 13th ACM Symp. on Principles and Practice of Parallel Programming*, pages 237–246, 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. In *Proc. of the 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. In *OOPSLA 2003 Conf. Proc.*, Oct. 2003.

[16] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing Memory Transactions. In *Proc. of the SIGPLAN 2006 Conf. on Programming Language Design and Implementation*, pages 14–25, June 2006.

[17] M. Herlihy and J. E. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proc. of the 20th Intl. Symp. on Computer Architecture*, pages 289–300, May 1993. Expanded version available as CRL 92/07, DEC Cambridge Research Laboratory, Dec. 1992.

[18] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proc. of the 22nd ACM Symp. on Principles of Distributed Computing*, pages 92–101, July 2003.

[19] G. Kestor, S. Stipic, O. Unsal, A. Cristal, and M. Valero. RMS-TM: A Transactional Memory Benchmark for Recognition, Mining, and Synthesis Applications. In *4th ACM SIGPLAN Workshop on Transactional Computing*, Feb. 2009.

[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. In *Proc. of the 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. In *Proc. of the 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. In *Proc. of the 20th ACM Symp. on Parallelism in Algorithms and Architectures*, pages 314–325, June 2008.

[23] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proc. of the 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. In *Proc. of the 16th Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 365–375, Sept. 2007.

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

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

[27] T. Riegel, C. Fetzer, and P. Felber. Time-based Transactional Memory with Scalable Time Bases. In *Proc. of the 19th ACM Symp. on Parallelism in Algorithms and Architectures*, pages 221–228, June 2007.

[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. In *Proc. of the 11th ACM Symp. on Principles and Practice of Parallel Programming*, pages 187–197, Mar. 2006.

[29] M. L. Scott, M. F. Spear, L. Dalessandro, and V. J. Marathe. Delaunay Triangulation with Transactions and Barriers. In *Proc. of the 2007 IEEE Intl. Symp. on Workload Characterization*, Sept. 2007. Benchmarks track.

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

[31] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization Techniques for Software Transactional Memory. Tr 915, Dept. of Computer Science, Univ. of Rochester, Feb. 2007.

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

[33] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict Detection and Validation Strategies for Software Transactional Memory. In *Proc. of the 20th Intl. Symp. on Distributed Computing*, pages 179–193, Sept. 2006.

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

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

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