JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory

Marek Olszewski, Jeremy Cutler, and J. Gregory Steffan
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario M5S 3G4, Canada
{olszews, cutler, steffan}@eecg.toronto.edu

Abstract

With the advent of chip-multiprocessors, we are faced with the challenge of parallelizing performance-critical software. Transactional memory (TM) has emerged as a promising programming model allowing programmers to focus on parallelism rather than maintaining correctness and avoiding deadlock. Many implementations of hardware, software, and hybrid support for TM have been proposed; of these, software-only implementations (STMs) are especially compelling since they can be used with current commodity hardware. However, in addition to higher overheads, many existing STM systems are limited to either managed languages or intrusive APIs. Furthermore, transactions in STMs cannot normally contain calls to unobservable code such as shared libraries or system calls.

In this paper we present JudoSTM, a novel dynamic binary-rewriting approach to implementing STM that supports C *and* C++ *code. Furthermore, by using value-based conflict detection, JudoSTM additionally supports the transactional execution of both (i) irreversible system calls and (ii) library functions that may contain locks. We significantly lower overhead through several novel optimizations that improve the quality of rewritten code and reduce the cost of conflict detection and buffering. We show that our approach performs comparably to Rochester's RSTM library-based implementation—demonstrating that a dynamic binary-rewriting approach to implementing STM is an interesting alternative.*

1. Introduction

As chip-multiprocessors become increasingly ubiquitous, researchers are faced with the daunting task of making it easier to exploit them through parallel programs. Traditionally, programmers have relied on lock-based synchronization for protecting accesses to shared data in concurrent applications. However, parallel programming with locks requires considerable expertise to obtain scalable performance. Typically, programmers must resort to fine-grained locking—a method that is difficult to implement correctly, often falls prey to problems such as deadlock and priority inversion, and can hinder modularity since locks must often span software component boundaries.

Transactional memory (TM) has emerged as a promising solution to the parallel programming problem by simplifying synchronization to shared data structures in a way that is scalable, safe, and easy to use. When programming an application with TM, a programmer encloses any code that accesses shared data in a coarse-grain transaction that will be executed atomically; the underlying TM system executes transactions optimistically in parallel while remaining deadlock free. Optimistic concurrency is supported through mechanisms for *checkpointing*, *conflict detection*, and *rollback*, which comprise the foundation of TM systems and have been the subject of continued research into hardware [2, 10, 19], software [5, 8, 11, 17, 18, 22, 25], and hybrid [4, 14, 23] approaches. Software-only implementations of TM (STMs) are especially compelling since they can be used with current commodity hardware, and provide an opportunity to gain experience using real TM systems and programming models.

1.1. Challenges for STM Systems

STMs have several challenges remaining, the most significant being the large overheads associated with a software-only approach. However, the following two additional key challenges for STMs have so far received little attention.

Support for Unmanaged/Arbitrary Code Many STM designs [9, 11, 17] target managed programming languages such as Java and C# to reduce overheads by leveraging the availability of objects, type safety, garbage collection, and exceptions. However, these systems do nothing to support

TM for the majority of programmers who work in unmanaged languages such as C and C++. Fortunately there has been a growing interest in supporting these languages: both OSTM [8] and RSTM [18] support object oriented C and C++ applications, respectfully. However, only recently has a solution targeting arbitrary (i.e., non-object oriented) C and C++ code been proposed [25]. Unfortunately, the design requires programmers to hand-annotate all functions that might be called within a transaction, precluding support for calls to pre-compiled code.

Support for Library and System Calls For TM to achieve wide adoption, programmers must be able to use modular software components, including the crucial ability to use pre-compiled and legacy libraries—without such support, programmers must re-implement STM-friendly versions of any required library code at great development and verification cost. The development of STM-capable libraries will be slowed by the lack of STM standards in programming model, API, and hardware interface (if applicable). Furthermore, if library code contains system calls they must be hand-modified to follow the semantics of *open-nested* transactions [20].

1.2. STM via Dynamic Binary-Rewriting

In this paper we introduce JudoSTM, a novel STM system that uses *dynamic binary-rewriting* (DBR) to instrument applications for transactional execution. With DBR, code is transformed to support transactional execution on-the-fly at run-time—hence DBR intrinsically supports arbitrary unmanaged code, as well as static and dynamic shared libraries. Concurrent with our work, preliminary work by Ying *et al.* has demonstrated that DBR can be used to augment a compiler-based STM to support legacy library code [26]. With a DBR-based STM, programmers can immediately enjoy the software engineering benefits of using transactions even when programming with existing legacy component binaries.

Unfortunately, DBR does introduce additional overheads. While the cost of the rewriting process can be quickly amortized, the resulting code is typically slower to execute than before, even when not instrumented. While a portion of this extra overhead is due to an increase in branch instructions and code duplication, the majority is due to the cost of translating the targets of indirect control instructions such as returns, indirect jumps, and call instructions. Since targets vary during execution, the translation of targets must also be done at runtime at a significant cost. However, our optimized implementation of the translation mechanism—and other recently-proposed methods [12, 24] concurrent with our work—can significantly reduce this overhead. In Section 2 we will demonstrate that the performance of DBR is quite acceptable for implementing a STM.

JudoSTM detects conflicts by comparing the *values* of memory locations accessed by each transaction, ensuring that they have not been modified after they were originally read within that transaction. We call this approach *value-based conflict detection* [6]. Since value-based conflict detection allows each transaction to independently verify its read-set without posing any requirements on concurrent transactions, it can be used to detect conflicts between arbitrary transactions. However, value-based conflict detection is expensive: unlike other conflict detection algorithms, its overheads cannot be reduced by grouping multiple accesses to adjacent regions of memory to verify them simultaneously. Instead, a compare instruction must be executed for each memory location accessed. To reduce this overhead, we present a novel technique that emits custom *transaction-instance-specific* read-set validation code to reduce the number of instructions and data accesses required to verify a read-set. This same technique is also used to reduce the cost of committing write-buffered data.

In addition to supporting legacy and unmanaged code, a DBR-based STM such as JudoSTM can offer the following desirable features.

Sandboxing and Optimistic Read Concurrency *Optimistic read concurrency* (also referred to as *invisible-readers*) has been shown to improve performance by reducing inter-processor communication [18]. If a transaction executes with optimistic read concurrency, it must perform expensive read-set validation after each read to prevent it from operating on inconsistent data. Not doing so allows it to use inconsistent data to make control-flow decisions that, for a conventional STM, may lead execution to a region of code which has not been transformed for transactional execution—execution of such code cannot be rolled back and is therefore unsafe. Some degree of safety can be obtained with support from trap handlers or with safe load instructions [5]; however, only a truly *sandboxed* transaction can execute safely when operating on inconsistent data. Since DBR transforms *all* code at runtime it can efficiently implement truly sandboxed transactions with the help of trap handler support, enabling optimistic read concurrency without performing incremental read-set validation.

Privileged Transactions Blundell *et al.* [2] proposed that TM systems should support system calls since programmers are typically unaware of when their code may make such calls, especially when compiling and linking system components and libraries separately. Consequently they also proposed a hardware TM design that allows a single *unrestricted* transaction (i.e., one which is allowed to make system calls) to execute concurrently with simpler *restricted* ones. JudoSTM similarly supports a single transaction that can make system calls which we call a *privileged* transaction. Since system calls may perform I/O (which cannot be

undone), a privileged transaction cannot be rolled back once it makes a system call. Instead, JudoSTM ensures that any other non-privileged transactions are aborted should a conflict be detected. Since the system call escapes the control of JudoSTM, we cannot instrument its memory writes to verify the read-sets of the non-privileged transactions. As a result we cannot detect conflicts by comparing the memory addresses accessed by each of the transactions, or by comparing data versions [5] or access timestamps [25]; instead we depend on JudoSTM's value-based conflict detection mechanism, which allows us to detect conflicts despite system calls in a privileged transaction that escapes JudoSTM's control.

Legacy Lock Elision A store that does not change the contents of the overwritten memory location is called a *silent* store. These types of stores have been shown to be quite common in general-purpose applications [15]. For a TM system that employs write-buffering, any sequence of stores to a single address that ends with a store of the original value can be compressed down to a single silent store at commit time. A compelling example of such a sequence can occur during the acquisition and release of a test-and-set or compare-and-exchange lock that maintains its state using two values, such as `1` and `0`, to represent locked and unlocked states. If a transaction were to execute legacy code containing such a lock, it would first acquire it by atomically writing a `1`, assuming the lock is un-contended, and later free it by writing a `0`. A write-buffered STM could replace these two writes with a silent store at commit time, overwriting the lock without changing its state. Hence by performing value-based conflict detection, JudoSTM can efficiently ignore existing lock acquisitions in legacy software, allowing it to optimistically execute any lock-protected code across multiple concurrent transactions instead of detecting false conflicts. For locks that are coarse-grained, eliding them in this way can dramatically reduce transaction aborts.

1.3. Contributions

This paper makes the following contributions: (i) we present a novel STM system based on dynamic binary-rewriting that supports both statically and dynamically linked arbitrary `C` and `C++` code; (ii) we demonstrate the feasibility of such an approach by comparing a prototype to the Rochester Software Transactional Memory (RSTM) system; (iii) we propose the use of value-based conflict detection to efficiently support the transactional execution of already thread-safe library code and unobservable and irreversible code such as system calls; (iv) we introduce a new technique for improving the performance of software write-buffering and conflict detection by emitting and executing custom transaction-instance-specific verification and write-buffer commit code.

2. Judo

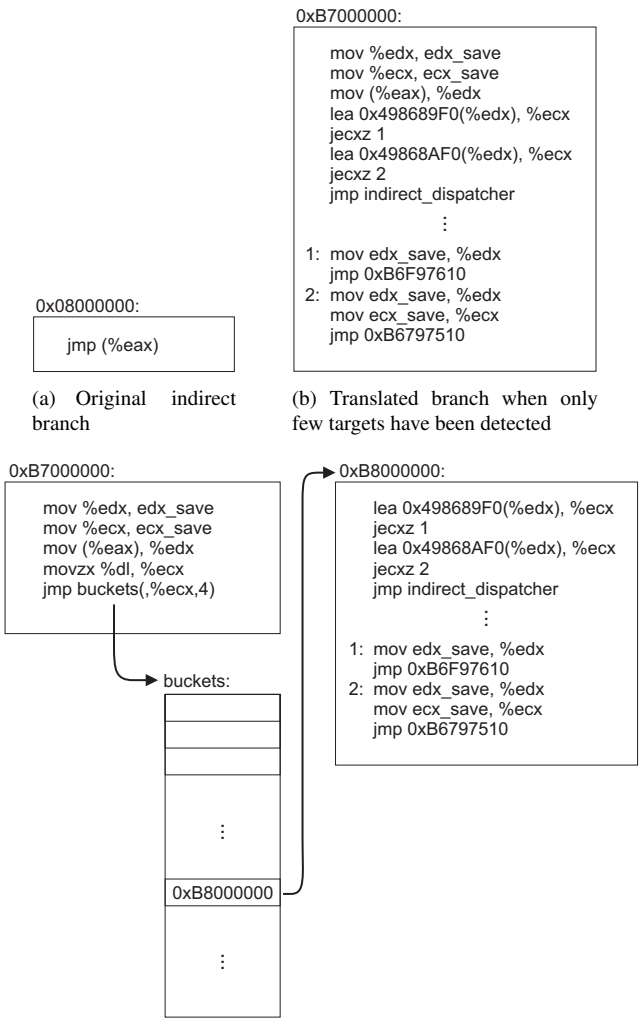
JudoSTM is built on Judo, our x86 DBR framework. Like most DBR systems, Judo lazily rewrites portions of an application *just-in-time* (i.e., right before the code is about to execute) into a *code-cache* from where it is executed. Within the code-cache, Judo is free to augment rewritten code with arbitrary instrumentation for a variety of purposes. Judo rewrites all control instructions within the code-cache to point to their code-cache equivalent targets; however, if a rewritten branch has a target that itself has yet to be rewritten, the rewritten branch will instead target stub code that invokes the just-in-time compiler (JIT) to rewrite the actual target. Hence Judo is able to sandbox an application, guaranteeing that any code that executes will be instrumented including any calls to legacy libraries or code executed by accident.

Like all DBR systems, Judo incurs overhead. However, since the cost of JITing can be quickly amortized we find that JITing alone is not the most significant component of total overhead. Instead, much of it can be attributed to (i) inferior code layout that results in extra branch instructions and code duplication, and (ii) the cost of indirect branch target translation that must be performed at runtime for each indirect branch, call, and return instruction. There has been much recent research on DBR, particularly on reducing these overheads. Judo implements many of these optimizations in addition to more novel ones described in further detail here. A more complete description of Judo's implementation is available in a previous publication describing Judo's predecessor JIFL [21].

2.1. Trace-Level JITing

Similar to both Pin [16] and HDTrans [24], Judo JITs at a *trace* granularity. These traces should not be confused with DynamoRIO's dynamically profiled hot traces [3]—instead, Judo selects traces statically at JIT time, although their selection can be indirectly influenced by previously executed code through the current contents of the code-cache. Many trade-offs exist when deciding trace sizes. JITing at a basic block granularity will reduce the size of the code-cache, since only code that is guaranteed to execute is rewritten and cached. However, basic block JITing can introduce new branches between basic blocks that were previously connected with fall-through edges, and place basic blocks according to their first execution order which may be unrepresentative of future executions. Alternatively, JITing large traces will likely reduce code-cache locality by JITing superfluous code that will never execute.

In Judo we attempt to balance these trade-offs by creating small traces that follow the original basic block layout. Judo JITs a small number of adjacent basic blocks together and connects them with the normal fall-through edges of conditional branches and call instructions. A trace is termi-



(a) Original indirect branch

(b) Translated branch when only few targets have been detected

(c) Translated branch when many targets have been detected

Figure 1. Indirect branch predication in Judo

nated early if an indirect branch/call, or return instruction is encountered, or if a fall-through target is already in the code-cache. Judo will not create traces that span unconditional branches; however, it will combine unconditionally linked traces if they are JITed consecutively. Of related schemes, this trace selection strategy is the most similar to that of HDTrans [24], except that HDTrans has unlimited basic blocks within a trace. We found that imposing a limit can significantly improve the performance of certain applications through improved code-cache locality and reduced pollution in unified upper-level caches. To minimize code duplication, Judo maintains a mapping between all re-written instructions and their original addresses (for basic blocks that are not instrumented with per-basic block instrumentation)—allowing jumps to target the middle of JITed traces rather than duplicating the targeted portions of such traces.

2.2. Indirect Branch Chaining

Indirect branches (such as that pictured in Figure 1(a)) can have multiple varying targets which must be translated to their code-cache equivalents every time the branch is executed. Judo makes use of *predicated indirect branch chaining*, a popular method of reducing the cost of translation that quickly translates and jumps to commonly-used targets through a sequence of comparison and jump instructions. Judo performs comparisons using the widely adopted `lea`/`jecxz` approach originally used in DynamoRIO [3], which does not affect condition flags. Each comparison requires two registers that Judo frees-up by spilling the resident values to global variables (`edx_save` and `ecx_save`), and later restoring them only for targets where they are live. Like Pin and HDTrans, Judo incrementally inserts new comparisons as new targets are detected. However, in Judo new instructions are emitted adjacent to each other in pre-allocated memory (as shown in Figure 1(b)), thus improving code-cache locality and eliminating the need for the extra jump instructions required by the predicated chains of dynamically-allocated memory used in Pin and HDTrans.[1]

If a substantial number of unique targets are encountered for a specific branch, Judo reduces the number of comparisons required to translate the target address by rewriting the branch to perform the lookup using a local *executable hash table* (shown in Figure 1(c)) that resembles the global *sieve* used in HDTrans [24]. Like the sieve approach, we use the `movzx` (move and zero-extend) instruction, since it does not overwrite the condition flags, as a hash function to convert an original branch target into a per-branch 256-entry table index that is used to indirectly branch to a shorter sequence of `lea`/`jecxz` comparisons.

2.3. Judo Performance

In Figure 2 we compare the performance of Judo and DynamoRIO on SpecINT2000 benchmarks; to demonstrate best-case performance, each was measured without actually applying instrumentation. Each of the benchmarks were compiled with gcc v3.3.6 at the `-O2` optimization level, and executed with the largest input file. Judo incurs only 15% overhead, compared to 26% for DynamoRIO. Furthermore, in rare cases (MCF and TWOLF) Judo can speed up execution by up to 7%: we attribute these speedups to the improved spatial code locality of the code-cache, which results in a smaller instruction working set and in turn less pollution in the unified caches. Both the benchmarks benefit from freed cache capacity due to their memory intensive nature. These promisingly-low overheads are what originally inspired us to build on Judo to develop JudoSTM.

[1] Note that we found our linear pre-allocated memory approach to outperform predicated chains of dynamically-allocated memory, even when elements are ordered optimally through dynamic profiling.

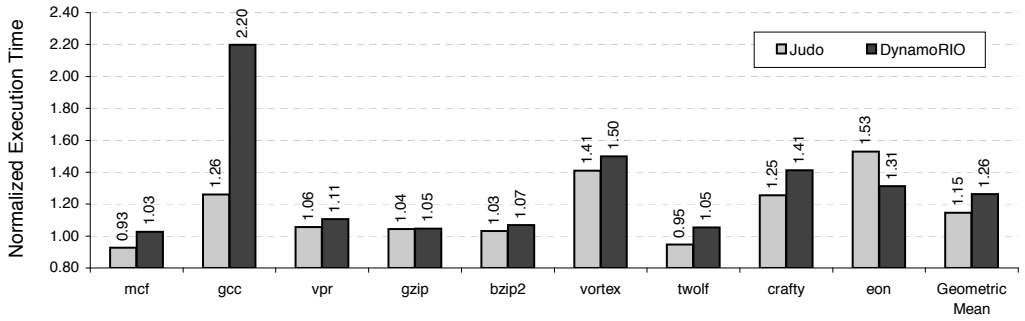


Figure 2. Dynamic binary rewriting performance comparison

```
#define atomic \
int __atomic_count = 0;\
asm volatile ("":::"eax", "ecx", "edx", "ebx",\
        "edi", "esi", "flags", "memory");\
judostm_atomic_start();\
for (; __atomic_count < 1; judostm_atomic_stop(),\
         __atomic_count++)
```

Figure 3. Definition of JudoSTM's atomic macro used to specify transactions

3. JudoSTM

Next we describe how we built on the Judo DBR framework to implement JudoSTM. In particular we explain how to program with JudoSTM, how we arrived at various design decisions for JudoSTM, and how it is implemented—including detailed descriptions of support for efficient read-set validation and commit, system calls, and library calls.

3.1. Programming with JudoSTM

JudoSTM is implemented as a self-contained static library which can be linked to any application requiring STM support. When using JudoSTM, a programmer need only include our `judostm.h` header file, and can then compile the program using any compiler.

To specify transactions, JudoSTM supports the commonly-used `atomic{}` syntax. Since JudoSTM does not have compiler support, it defines the `atomic` keyword using the macro definition shown in Figure 3. To insert calls to our runtime system both before and after the transaction statement block, the macro is defined as a `for` loop that executes its body once, after a call to `judostm_atomic_start` and before a call to `judostm_atomic_stop`. We find that the loop is typically eliminated by the compiler even at low optimization settings. The macro also contains GAS advanced inline assembly to specify that all of the registers may be *clobbered*, thus forcing the compiler to automatically checkpoint live input registers to the stack. We do not checkpoint live local input variables which have been spilled to the stack, and therefore encourage programmers not to create transactions that write to input variables in case they do not get checkpointed; doing so is easy and does not limit program expressibility. However, this limitation could be easily overcome with compiler support, for example by using a comprehensive checkpointing algorithm [25].

3.2. JudoSTM Design Decisions

JudoSTM's design is succinctly described as a write buffering, blocking, invisible-reader STM using value-based conflict detection. The following describes our rationale for choosing this point in the STM design space.

Value-Based Conflict Detection Value-based conflict detection was chosen because it is critical for allowing unobservable code, such as system calls, to execute safely in parallel with other speculative transactions. Furthermore, it efficiently ignores silent stores which has the nice benefit of allowing JudoSTM to elide legacy locks within transactions. Finally, value-based conflict detection is especially attractive for *ordered transactions*, since the oldest transaction need not execute with instrumentation for this conflict detection scheme.

Read Buffering JudoSTM uses a *read buffer* to store the initial value of each address loaded while executing a transaction. Upon commit, the read-set is validated by comparing the memory locations read by the transaction with the contents of the read buffer. While a transaction is executing, all accesses to previously-read locations must be redirected into the read buffer. The need for this indirection may seem unnecessary at first—but it is required to prevent conflicts from going undetected in the presence of three or more transactions (or two where one is privileged), which may cause a subtle read/write/read/write race condition as illustrated by the following example.

Consider two transactions that both access a global variable. Suppose that a transaction (`transaction1`) is executing concurrently with a second privileged transaction that has made a system call (`transaction2`)—since `transaction2` is privileged its writes are not

buffered. Assume that both transactions are operating on the same global variable (A). First transaction1 reads A and copies it to the read buffer for eventual read-set validation. Next transaction2 increments A, and transaction1 reads it again directly from memory. If transaction2 modifies A back to its original value before transaction1 performs read-set validation, transaction1 will fail to detect the conflict. To solve this problem, the second read is instead redirected to the read buffer—this way transaction1 will still commit but will have executed with a consistent view of A (i.e., the original value of A before transaction2 modified it). Therefore JudoSTM must introduce this level of indirection for all memory accesses that may conflict.

Write Buffering We also chose to use a write-buffer rollback mechanism instead of an undo-log approach because of the write-buffer's compatibility with the level of indirection already required by value-based conflict detection. While undo-logging has been shown to exhibit impressive results for low contention applications [7, 25], it suffers significantly in high contention scenarios where many conflicts trigger expensive rollbacks [5]; under these conditions, write-buffering fairs much better. Furthermore, recent implementations of write-buffering have been shown to perform competitively even in low contention scenarios [5].

Blocking To support system calls, which cannot be rolled back, JudoSTM requires a blocking design. While blocking does reduce dependability, recent STMs [5, 22, 25] have also implemented blocking designs to improve performance. Blocking transactions were originally championed by Ennals [7] who argued that many of the perceived negative attributes of blocking—such as convoying and priority inversion—are either largely irrelevant on future chip-multiprocessor hardware or acceptable to the majority of programmers in exchange for better performance.

Commit JudoSTM supports two types of commit: coarse and fine-grained. In coarse-grained commit, a single global lock is used to synchronize transaction validation and committing to enforce that transactions atomically either succeed or abort. To allow read-only transactions to commit in parallel, and to minimize the bus traffic caused by writes to the lock, read-only transactions do not acquire the lock. Instead, they check to ensure that the lock is not held by anyone at the start and after validating their read-set, and that no transaction committed any data between the two points. By using the upper 31 bits in the lock word for storing commit version numbers, we can perform both checks with just three instructions. Additional, memory fence instructions are inserted to prevent the comparisons from executing out-of-order with the read-set validation.

For fine-grained commit, JudoSTM uses a hash to map the address space into 8192 regions (a.k.a. strips) and associates a lock with each region. To commit, a transaction acquires the locks associated with the regions that it will write to. Similarly, a transaction verifies that the locks associated with the regions being validated are not acquired by other transactions during read-set validation. To prevent deadlock, rather than sorting the order of acquisitions, we instead rely on bounded spin-wait times.

Conflict Resolution JudoSTM resolves conflicts differently depending on which commit variant is used. For fine-grained commit, JudoSTM re-runs the failed transaction by executing its native (un-rewritten) code while holding the commit lock. Doing so is desirable because it prevents livelock and eliminates an extra (costly) lock acquisition while still allowing other transactions to execute (but not commit) concurrently. For coarse-grained commit, JudoSTM must acquire all locks to execute a transaction natively and therefore only does so after 1000 failed attempts.

Invisible Readers Since Judo can effectively sandbox arbitrary code, JudoSTM can sandbox transactions and hence safely implement invisible readers without having to verify the read-set after each load. We additionally use trap handler support to detect and recover when faults such as invalid pointer dereferencing occur. We currently exploit custom light-weight per-thread fault handler support that we added to our Linux Kernel; however, this support can alternatively be implemented using standard unix signal handlers, though possibly at a small performance cost.

3.3. Instrumenting a Transaction

When a thread first enters judostm_atomic_start, it is assigned a unique thread ID which it writes to a thread-local global variable. This thread ID is used to uniquely identify the transaction while it executes, allowing it to access any per-thread data it requires. It is also used to jump to a per-thread privatized transaction start routine that saves the contents of the stack and frame pointer registers and enables a per-thread fault handler. Finally, execution is handed-off to the Judo runtime system which instruments and executes the transaction in a per-thread private code-cache. Despite some inefficiency due to code duplication, we claim that private code-caches are desirable for DBR-based STMs since they enable efficient thread-private instrumentation, and since it will likely not be beneficial to have more threads than processors.

During execution JudoSTM instruments every instruction which it believes may access global memory, ignoring stack accesses which are implicitly write-buffered on the stack. As a heuristic, JudoSTM assumes that all instructions that implicitly or explicitly use the stack or frame pointers do not access global memory. We found this assumption to hold for any code that we tested; however, ensuring that this

```
...
cmp $0x256, 0x80B10CFC
jne,pn judostm_trans_abort
cmp $0x1, 0x80B10CA4
jne,pn judostm_trans_abort
cmp $0x80B10CFC, 0x80B10BB8
jne,pn judostm_trans_abort
cmp $0x80B10CA4, 0x80B10BCC
jne,pn judostm_trans_abort
ret
```

Figure 4. Example of emitted transaction-instance-specific read-set validation code

is the case is essential to making JudoSTM robust. Ying *et al.* describe a number of methods for determining when this assumption might be broken [26], which we plan to investigate in future work.

To provide the level of indirection required by both value-based conflict detection and write-buffering, instructions that do not access the stack are rewritten to use an effective address obtained through a hash table lookup. We use two linear-probed open-address hash tables: one for looking-up read accesses, and one for looking-up writes. Judo inlines the first portion of the lookup instrumentation, so that a hit in the hash table requires only five extra instructions plus any instructions needed to save and restore the `%ecx`, `%edx`, and `eflags` (condition flags) registers should they be live after the instrumentation. If this code misses, then a function is called that continues probing the hash table. If the effective address is absent then the target data is stored in the read/write buffer and the hash tables are updated accordingly.

Arbitrary `C/C++` code can often alias stack variables and access them with pointers other than the frame and stack pointers. Like regular stack accesses, these need not be write-buffered since they can be rolled back implicitly along with the stack. In fact, write buffering is undesirable in this case since it can lead to stack corruption during commit time [25], and can make regular stack instructions—that use frame or stack pointers—access inconsistent data since they are not redirected to the read/write buffers. Hence, JudoSTM must check all newly-discovered effective addresses to ensure that they are not on the stack before redirecting them to the read/write buffers. Currently, we do not support sharing of stack local data between threads; however, such sharing can be detected using a page protection mechanism [26].

Because JudoSTM does not insert read-set validations after each memory read, transactional threads that have read inconsistent data can sometimes enter infinite loops. To prevent such loops, JudoSTM inserts instrumentation to validate the transaction's read-set on every backward branch edge; to reduce overhead, this instrumentation is guarded by `inc` and `jne` instructions which increment a byte counter and jump over the instrumentation until the branch is encountered 256 times.

```
...
movl $0x0, 0x80B10CA4
movl $0x80B10CFC, 0x80B10BCC
movl $0x80B10CA4, 0x80B10BB8
ret
```

Figure 5. Example of emitted transaction-instance-specific commit code

3.4. Efficient Validation and Commit

When a transaction completes its execution, it acquires the commit locks (either the single coarse-grain commit lock or the appropriate fine-grain commit locks), verifies its read-set, copies its write-set to main memory, and then releases the commit lock(s). Minimizing the duration of this critical section is therefore crucial, and JudoSTM does so by emitting *transaction-instance-specific* code that performs read-set validation and the commit operation. This code is emitted incrementally in straight-line sequences as the transaction executes. Time spent in the commit-time critical section is minimized because the sequences are emitted ahead of time specifically for the dynamic instance of each transaction, and the emitted code contains only the bare-minimum of control flow instructions.

For read-set validation, a straight-line sequence of `cmp` and `jne` instructions (as in Figure 4) is emitted to compare all values read by the transaction with the contents of their original locations in memory. Should any comparison fail, the corresponding `jne` instruction jumps to an abort handler that flushes the read and write buffers, unrolls the stack, and restarts execution. To improve ILP we statically hint each `jne` branch with a *branch not taken* prefix. In addition, to minimize L1 data cache traffic, we store the values and their corresponding effective addresses as immediates in the `cmp` instruction. Finally, this list of immediates actually constitutes our read buffer since they are emitted each time a new read address is discovered.

We use a similar approach for commit code, for which we emit a straight-line sequence of `mov` instructions (as in Figure 5) that directly copy the contents of the write-buffer to their corresponding memory locations. Again, the write-buffer is comprised of the list of immediates encoded directly in the `mov` instructions.

To expedite the creation of validation and commit code, a large buffer of each sequence is pre-allocated and initialized with the appropriate instructions so that only the immediate values need be filled in during transaction execution. Immediates are written into the instructions in reverse order as the transaction executes—this way JudoSTM can track the top-most instruction as the sequence fills, and can later use an indirect call to jump directly to the start of the sequence.

Each pre-allocated sequence ends with a `ret` instruction so that execution properly returns at the end of the sequence. While executing this frequently-emitted code will increase instruction cache misses, this cost is quickly amortized for large read and write-sets. Furthermore, JudoSTM attempts to prefetch these instructions while waiting for the commit lock.

Finally, it is important to note that even though it relies on emitting code, this technique is not limited to DBR frameworks, and could be easily used by a compiler-based STM to expedite committing the write-buffer to memory.

3.5. Supporting System Calls

JudoSTM allows a single privileged transaction to execute system calls and be executed concurrently with other non-privileged transactions; however, since the privileged transaction cannot rollback, no other transaction can commit until the privileged transaction is complete. Therefore, JudoSTM rewrites all system call traps (`int80` instructions) with jumps to its own system call handler. This handler acquires either the single coarse-grained commit lock, or all fine-grained commit locks, validates the read-set, and finally jumps to the original trap instruction (in the original code). This ensures that the system call and the remainder of the transaction execute while the commit lock(s) are held, ensuring that no conflict can occur. Other transactions can continue to execute and detect conflicts (using value-based conflict detection) concurrently with the privileged transaction; however, they must still await the commit lock(s) before they can complete.

3.6. Transactional Memory Management

Since JudoSTM supports shared libraries, it can also dynamically instrument the gnu `libc malloc()` and `free()` functions, enabling them to execute optimistically via the write-buffer. Furthermore, because JudoSTM supports system calls, `malloc()` will continue to execute correctly even when it needs to extend the heap via a call to `sbrk()` or `mmap()`. Supporting the native `malloc()` implementation eliminates the need for custom transaction-aware allocators [13], garbage collectors, or quiescing [5]. Through JudoSTM, `malloc()` and `free()` execute with full transaction semantics: memory allocations or frees will only be externally visible upon successful completion of a transaction. This prevents the heap from "blowing up"[2] in the case of frequent failed transactions, and eliminates any risk of accessing stale pointers. In addition, memory that is dynamically allocated within a transaction can be freed outside of any transaction, and vice-versa.

Unfortunately, the gnu `libc malloc()` has yet to be optimized for scalable concurrent execution. As a result, we found that a significant portion of transaction aborts were caused by conflicts related to concurrent memory allocation requests. As a temporary solution, we recommend whenever possible to link with a dynamic memory allocation library that is designed for scalable parallel execution. In this paper we use the Hoard highly scalable parallel memory allocator [1]. Because JudoSTM eliminates all `lock` prefixes in the re-written code, the JITed version of Hoard becomes a highly-efficient non-blocking transactional memory allocator.

[2] For the heap to "blow up" means for it to grow exceedingly larger than necessary because of poor memory recycling.

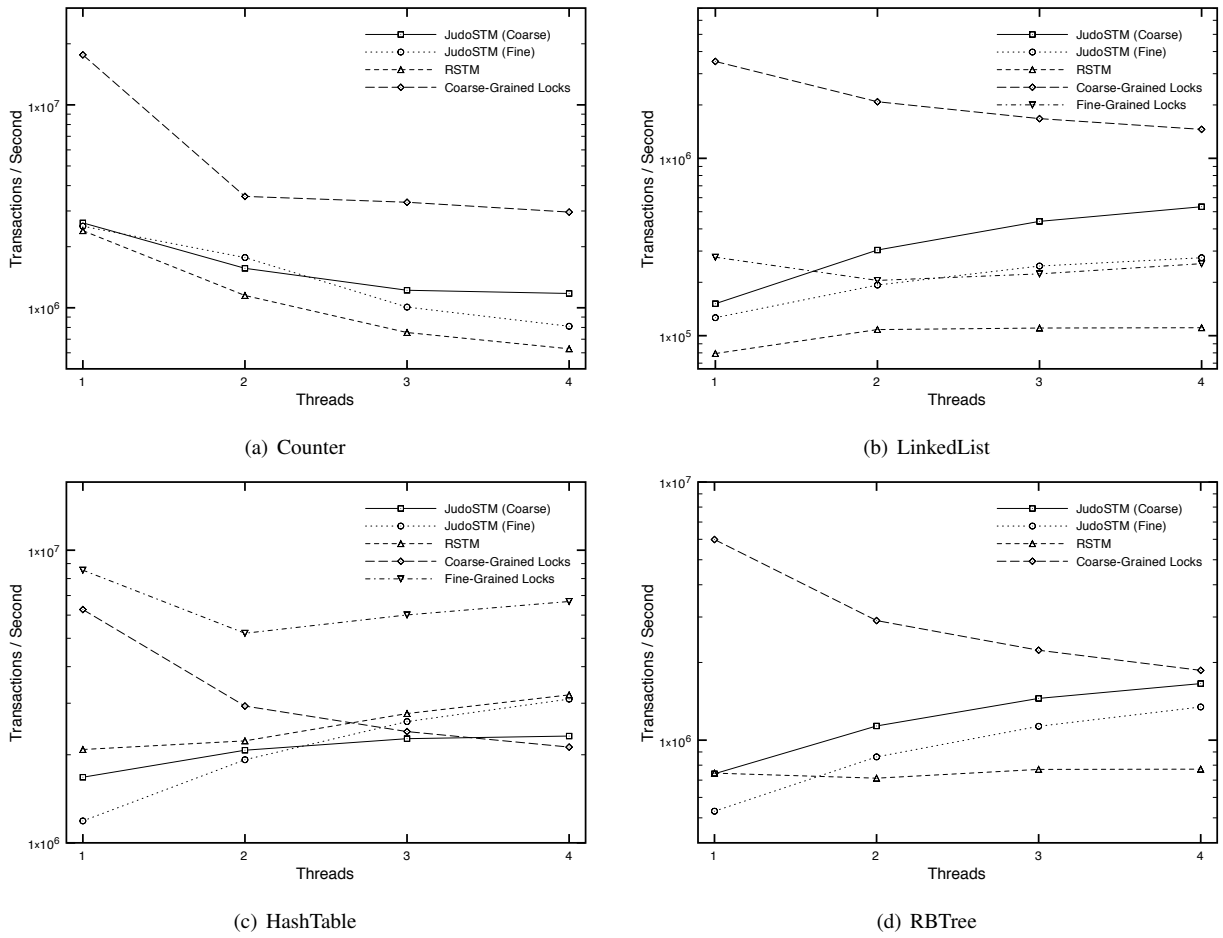
4. Evaluation

In this section we compare the performance of JudoSTM to both a conventional lock-based execution and also to the RSTM system [18] on a set of micro-benchmarks.

4.1. Experimental Framework

We measure JudoSTM on a multiprocessor machine with four 2.8GHz Intel Xeon MP processors and 16GB of main memory. Each processor implements a 12K μOps instruction trace cache, 8KB L1 data cache, 512MB L2 and 2MB L3 unified cache. JudoSTM and the lock-based implementations are wrapped in the RSTM `C++` API framework to ensure fair comparisons. Each system was built with the compiler that gave the highest performance at the `-O3` optimization level: the RSTM and lock-based systems were compiled using `g++` v4.1.2, while for JudoSTM `g++` v.3.3.6 was used. We measure throughput in *transactions-per-second* over a period of 10 seconds for each benchmark, and vary the number of threads from one to four. All results are averaged over a set of ten test runs. In all experiments involving RSTM, the `Polka` contention manager is used with eager acquire, invisible readers, and the epoch-based RSTM memory manager. JudoSTM was linked with Hoard 3.6.2.

Micro-benchmarks We evaluate using a subset of micro-benchmarks available with the RSTM API. These include a simple counter (COUNTER), as well as three different integer benchmarks: a sorted linked list (LINKEDLIST), a hash table with 256 buckets (HASHTABLE) and a red-black tree (RBTREE). For the integer benchmarks each thread performs an equal mix of *insert*, *remove*, and *lookup* operations. The COUNTER benchmark comprises short transactions that simply increment a single shared counter—for this reason we only consider coarse-grained locking for COUNTER (which in this case is equivalent to fine-grained locking). COUNTER provides a base comparison for the performance of each parallelization method in the case of high contention. In the LINKEDLIST benchmark, transactions traverse a sorted list to locate an insertion or removal point; when found, either a new node is inserted or an existing node is removed, and the relevant pointers are up-



(a) Counter

(b) LinkedList

(c) HashTable

(d) RBTree

Figure 6. Benchmark comparisons (Note the log scale)

dated. The HASHTABLE benchmark is implemented using 256 buckets with linked list overflow chains and a simple modulus hash function; as there are roughly an equal number of insertion and removal operations in our experiments, the hash table is maintained at roughly 50% capacity during execution. Finally, in the RBTREE benchmark, *insert* and *remove* operations generate one or more modifications to other tree nodes during the height-balancing phase. For RBTREE a fine-grained locking solution is non-trivial and is not provided by RSTM.

4.2. Performance

Figure 6 presents the throughput results for each of the four micro-benchmarks. Note that the y-axes are in a log-scale. In every scenario JudoSTM performs competitively with RSTM.

Counter In COUNTER, high contention causes all synchronization methods to perform poorly as more threads are added. Both versions of JudoSTM perform relatively strongly in this worst case scenario, degrading less than RSTM with additional threads.

LinkedList In LINKEDLIST, the coarse-grained version of JudoSTM scales close to linearly, obtaining a throughput of 3.5x its single threaded performance at four threads. Because of a large number of conflicts, this version benefits from its contention heuristic which causes the transaction to execute natively after a rollback. JudoSTM's fine-grained commit implementation suffers due its higher overhead coupled with the large number of transaction aborts; however, it is still able to outperform RSTM and fine-grained locking at three and four threads.

HashTable The HASHTABLE micro-benchmark is perhaps the most interesting as it represents a low contention scenario. Here we begin to see the benefit of using fine-grained commit, which performs up to 34% better than coarse-grained commit. At four threads, JudoSTM's fine-grained commit beats coarse-grained locking by 46%, and performs similarly to RSTM despite having significantly

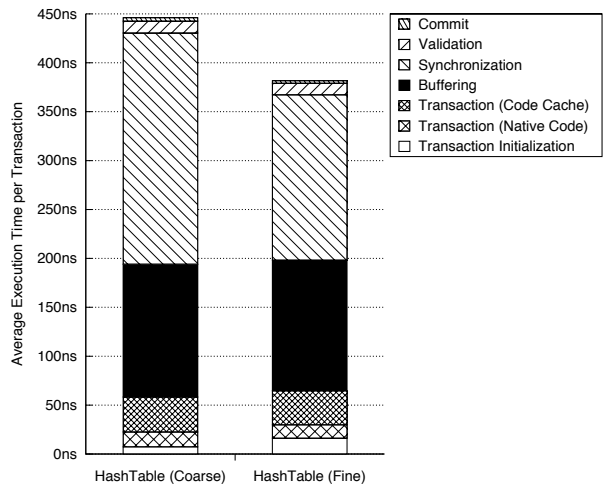


Figure 7. Execution breakdown of HashTable on 4 processors

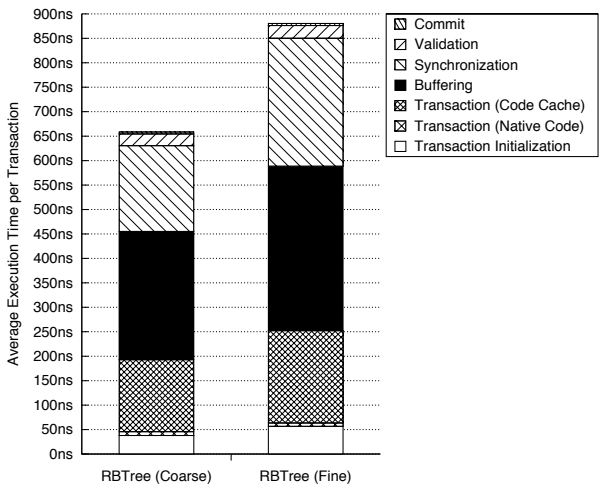


Figure 8. Execution breakdown of RBTree on 4 processors

more single threaded overhead than RSTM.

RBTree In RBTREE, both versions of JudoSTM scale fairly well, approaching the performance of coarse-grained locking at four threads. In contrast RSTM fails to scale at all performing over 2x slower than JudoSTM at four threads.

4.3. Examining Execution

Figures 7 and 8 show the average execution breakdown of a transaction for both fine and coarse-grained commit variations of JudoSTM for both HASHTABLE and RBTREE. In all cases, JudoSTM spends the majority of its time buffering reads and writes, and synchronizing commits with other transactions. Thanks to the per-transaction-instance emitted validation and commit code, JudoSTM spends only 3.7% of its time, on average, validating and committing read and write-sets.

For HASHTABLE, the coarse-grained commit implementation spends 53% of its time attempting to acquire the single commit lock (*Synchronization*). Because of the large amount of parallelism in HASHTABLE, fine-grained JudoSTM is able to reduce its synchronization time by 28.5% through acquiring more locks which are each less contended, despite the additional time needed to compute the set of locks that need to be acquired. However, in the higher contention case represented by RBTREE, the overhead of fine-grained commit increases the time spent on synchronization by 49.2%. Furthermore, the increased chance of deadlock further degrades performance by triggering more transaction re-executions reflected by the greater amount of time spent executing the transaction and buffering its reads and writes.

5. Future Work

JudoSTM does not currently provide *strong atomicity*. As a result, programmers must be cautious not to concurrently call functions that operate on shared memory, from both inside and outside of atomic blocks, even if they are thread safe (e.g. `malloc`). While JudoSTM's use of value-based conflict detection does enable transactions to detect conflicts between themselves and non-atomic code, in this case it is possible that conflicts are introduced between read-set validation and the completion of the write-set commit. Hence extending JudoSTM to provide a stronger level of safety is interesting work for the future.

6. Conclusion

As transactional memory systems move closer to mainstream use, we must make them easier to integrate into typical programming environments—hence for software transactional memory (STM) systems it is important to support arbitrary `C` and `C++` code, as well as library functions that may themselves contain system calls and locking code. We have presented JudoSTM, a STM system based on Judo, our dynamic binary-rewriting framework. Judo implements several key optimizations including trace-level JITing and highly efficient indirect branch chaining, allowing it to incur only a modest overhead, and thus serve as a feasible base for an STM system. JudoSTM is a write-buffering, blocking, invisible reader STM that uses value-based conflict detection and is programmed using only a simple `atomic{}` macro and may be built with any compiler. JudoSTM supports several desirable features including sandboxing, optimistic read concurrency, legacy lock elision, and transactions that can execute system calls. We have demonstrated that JudoSTM performs comparably to Rochester's

RSTM library-based implementation—demonstrating that a dynamic binary-rewriting approach to implementing STM is an interesting alternative.

References

[1] E. Berger, K. McKinley, R. Blumofe, and P. Wilson. Hoard: A scalable memory allocator for multithreaded applications. Technical report.

[2] C. Blundell, E. C. Lewis, and M. M. K. Martin. Unrestricted transactional memory: Supporting i/o and system calls within transactions. Technical Report CIS-06-09, Department of Computer and Information Science, University of Pennsylvania, Apr 2006.

[3] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, Washington, DC, USA, 2003.

[4] C. Cao Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. Jun 2007.

[5] D. Dice and N. Shavit. Understanding tradeoffs in software transactional memory. In *Proceedings of the International Symposium on Code Generation and Optimization*, Mar 2007.

[6] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*.

[7] R. Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, Jan 2006.

[8] K. Fraser. *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003.

[9] R. Guerraoui, M. Herlihy, and B. P. P. Contention. Management in sxm. In *Proceedings of the 19th International Sympposium on Distributed Computing*, Sep 2005.

[10] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA '04: Proceedings of the 31st annual International Symposium on Computer Architecture*, pages 102–113, June 2004.

[11] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. pages 92–101, Jul 2003.

[12] J. D. Hiser, D. Williams, W. Hu, J. W. Davidson, J. Mars, and B. R. Childers. Evaluating indirect branch handling mechanisms in software dynamic translation systems. In *Proceedings of the International Symposium on Code Generation and Optimization*, Mar 2007.

[13] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. C. Hertzberg. Mcrt-malloc: a scalable transactional memory allocator. In *Proceedings of the International Symposium on Memory management*, New York, NY, USA, 2006.

[14] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. in proceedings of the 11th. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, Mar 2006.

[15] K. M. Lepak and M. H. Lipasti. On the value locality of store instructions. In *Proceedings of the International Symposium on Computer Architecture*, New York, NY, USA, 2000.

[16] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. f Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the Conference on Programming Language Design and Implementation*, New York, NY, USA, 2005.

[17] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive software transactional memory. In *Proceedings of the International Symposium on Distributed Computing*, Cracow, Poland, Sep 2005.

[18] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the overhead of software transactional memory. Technical Report TR 893, Computer Science Department, University of Rochester, Mar 2006.

[19] A. McDonald, J. Chung, H. Chafi, C. Cao Minh, B. D. Carlstrom, L. Hammond, C. Kozyrakis, and K. Olukotun. Characterization of tcc on chip-multiprocessors. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*. Sept 2005.

[20] Y. Ni, V. Menon, A.-R. Adl-Tabatabai, A. Hosking, R. Hudson, E. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Mar 2007.

[21] M. Olszewski, K. Mierle, A. Czajkowski, and A. D. Brown. Jit instrumentation—a novel approach to dynamically instrument operating systems. In *EuroSys 2007*, Mar 2007.

[22] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*. Mar 2006.

[23] A. Shriraman, V. Marathe, S. Dwarkadas, M. L. Scott, D. Eisenstat, C. Heriot, W. N, S. III, and M. F. Spear. Hardware acceleration of software transactional memory. Technical Report 887, Department of Computer Science, University of Rochester, Dec 2005.

[24] S. Sridhar, J. S. Shapiro, E. Northup, and P. P. Bungale. Hdtrans: an open source, low-level dynamic instrumentation system. In *Proc. of the International Conference on Virtual Execution Environments*, New York, USA, 2006.

[25] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *Proceedings of the International Symposium on Code Generation and Optimization*, Mar 2007.

[26] V. Ying, C. Wang, and Y. Wu. Dynamic binary translation and optimization of legacy library code in an stm compilation environment. In *Proceedings of the Workshop on Binary Instrumentation and Applications*. Oct 2006.