A Dynamic Instrumentation Approach to Software Transactional Memory

by

Marek Olszewski

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

A Dynamic Instrumentation Approach to Software Transactional Memory

Marek Olszewski

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2007

With the advent of chip-multiprocessors, we are faced with the challenge of parallelizing performance-critical software. Transactional memory has been proposed as a promising programming model, with software transactional memory (STM) being especially compelling for today's use. However, in addition to high overheads, existing STMs 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 dissertation we present JudoSTM, a novel dynamic binary-rewriting approach to implementing STM that supports `C` and `C++` code. By using value-based conflict detection, JudoSTM additionally supports the transactional execution of both (i) irreversible system calls and (ii) code that contains locks. We lower overhead through several novel optimizations that improve invisible-reader and validate/commit performance. We show that our approach performs comparably to RSTM—demonstrating that a dynamic binary-rewriting approach to implementing STM is an interesting alternative.

Acknowledgements

I am deeply grateful to everyone that made this dissertation possible. I am especially thankful to Professor J. Gregory Steffan for offering to take me on mid-year, and for patiently providing guidance, feedback, and engaging conversations. I would also like to thank my family for their continued support and inspiration. Additionally, I must thank Natalia Borecka and my close friends who have been an invaluable source of encouragement. Lastly, I would like to acknowledge The Edward S. Rogers Sr. Department of Electrical & Computer Engineering for its financial support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The continued proliferation of chip-multiprocessor (CMP) computer architectures will make multiprocessor workstations ubiquitous. Like traditional multiprocessors, CMPs can improve program performance if applications are parallelized appropriately. Therefore, in the interest of remaining competitive, software companies are left with the daunting task of parallelizing many of their general purpose applications. In the hope of helping to expedite this transition, researchers are rapidly searching for ways to make it easier to write parallel programs that can leverage the new hardware. Using today's parallel programming models, software developers writing parallel applications rely on lock-based synchronization to protect accesses to shared data. Unfortunately, parallel programming with locks requires considerable expertise. To obtain scalable performance, programmers must employ fine-grained locking strategies, which are both tedious and difficult to implement correctly. In addition to the usual race conditions common in lock-based code, fine-grained locking is highly susceptible to deadlock and priority inversion. Finally, since lock-based code is not composable, programmers must often access locks across component boundaries, breaking software abstractions and leaving sound software engineering practices behind.

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 and composable, while still 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 executes atomically; the underlying TM system executes transactions optimistically in parallel while remaining deadlock-free. Optimistic concurrency is supported through mechanisms for *isolation*, *conflict detection*, and *commit/abort*, which comprise the foundation of TM systems. These mechanisms have been the subject of continued research in hardware [2, 6, 17, 30, 31, 36], software [11, 12, 15, 20, 27, 28, 37, 41], and hybrid [9, 24, 39] approaches. While hardware transactional memory systems (HTMs) will most likely prevail, software-only systems (STMs) are especially compelling today, since they can be used with current commodity hardware and provide an early opportunity to gain experience using real TM systems and programming models. Furthermore, STMs comprise a key component of many hybrid TMs (HyTMs), which leverage HTM hardware if available, but fall back on the STM should a need arise. Such HyTMs are appealing because they allow future HTMs to be integrated incrementally, reducing risk and development costs.

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

Support for Unmanaged/Arbitrary Code: Many STM designs [16, 20, 27] 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 software developers who work in unmanaged languages such as `C` and `C++`. Fortunately there has been a growing

interest in supporting these languages: both OSTM [15] and RSTM [28] provide a library that developers can use to hand-port object oriented `C` or `C++` source code for transactional execution. However, only recently has a fully-automatic solution, which targets arbitrary (i.e., non-object oriented) `C` and `C++`, been proposed [41]. 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, software developers 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, in the likely event that the library code contains system calls, current research proposes the use of *open-nesting* transactions, which requires that libraries be hand-modified to follow the complicated (non-serializable) semantics that open-nesting designs dictate [29].

Support for Legacy Locks: Current STM systems have yet to make an effort in supporting transactions that contain code operating on legacy locks. Such transactions can easily arise when porting legacy code that calls existing thread-safe code, e.g.: the `libc malloc()`. If an STM ignores legacy locks it will incur false conflicts every time two concurrent transactions acquire or release a lock. Allowing such false conflicts to occur effectively serializes the transactions at the lock boundaries irrespective of whether true contention is actually present. If the legacy locking strategy employed is coarse-grained, aborting transactions on such conflicts will lead to detrimental performance.

1.1 Software Transactional Memory Through Dynamic Binary Rewriting

In an effort to address these challenges, this dissertation proposes the use of dynamic binary rewriting (DBR) to instrument applications with software checkpointing, rollback and conflict detection mechanisms, which are needed to enable transactional execution. With DBR, the STM instrumentation is added to a compiled binary application 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 research by Ying *et al.* has demonstrated that DBR can be used to augment a compiler-based STM to support legacy library code [42]. With a DBR-based STM, programmers can immediately enjoy the software engineering benefits of transactional memory even when programming with existing legacy component binaries.

Unfortunately, DBR can introduce additional overheads. As with any dynamic instrumentation, there is a cost to rewriting the application during its execution. Furthermore, the rewritten code often becomes inferior in quality, even when not instrumented, resulting in overhead that cannot be amortized. On the other hand, DBR has desirable properties that can benefit the performance of an STM system. Since the DBR system must maintain control over the target application at all times, DBR provides a level of *sandboxing* that allows an STM to optimistically schedule transactions in parallel, without requiring each transaction to incrementally verify that it is working on valid data after each load from memory. Currently, incremental validation is necessary to ensure that the application does not accidentally perform an illegal operation such as a segfault, or an arithmetic error.

To support system calls and legacy locks, we propose detecting 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* [13, 18, 35]. 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 caused by other threads, even if they are executing system calls, which cannot be directly observed through dynamic instrumentation. Additionally, we show that value-based conflict detection can efficiently ignore existing lock acquisitions in legacy software, allowing it to optimistically execute lock-protected code across multiple concurrent transactions.

1.2 Research Goals

The focus of this dissertation is to develop an STM system that uses dynamic binary rewriting and value-based conflict detection to support transactions that contain arbitrary `C` or `C++` code, calls to pre-compiled libraries, system calls, and legacy locks. To this end, we have the following goals:

1. To develop a dynamic binary rewriting system efficient enough to support STM;
2. To demonstrate that DBR can feasibly implement an automatic STM system, and perform comparably to existing STM systems;
3. To leverage value-based conflict detection to address the shortcomings of previous STMs; and
4. To identify and optimize the key bottlenecks in performance.

To satisfy the first of these goals, we have developed the Judo dynamic instrumentation framework—a highly optimized dynamic binary rewriting system capable of outperforming existing dynamic instrumentation tools. Judo emits code at a trace granularity

to minimize the introduction of superfluous instructions which degrade the quality of the instrumented code. Additionally, Judo introduces a novel optimization to dramatically reduce the cost of translating addresses targeted by indirect control instructions, such as indirect jumps or returns. To address the second goal, we built on Judo to develop JudoSTM, a prototype STM system that dynamically rewrites transactions to contain the necessary instrumentation needed for transactional execution. Despite the overheads of performing dynamic binary rewriting, we show that JudoSTM is competitive to current STMs by comparing it to the Rochester Software Transactional Memory (RSTM) system. With the third goal in mind, we developed JudoSTM to use value-based conflict detection. As a result, JudoSTM is the first STM that can efficiently support transactions containing system calls and/or legacy locks. Finally, we thoroughly profiled and optimized JudoSTM to reduce performance bottlenecks, as reported in this dissertation.

1.3 Organization

The rest of this dissertation is organized as follows. Chapter 2 provides a background in parallel programming, transactional memory and dynamic binary rewriting, and discusses the relevant related work. Chapter 3 describes the design of the Judo dynamic binary rewriting framework and evaluates its performance. Chapter 4 provides a detailed description of the design and implementation of the JudoSTM system. Chapter 5 compares the performance of JudoSTM to RSTM, and takes a closer look at JudoSTM's execution breakdown. Finally, Chapter 6 concludes by summarizing the dissertation, listing its contributions, and describing potential extensions for the future.

Chapter 2

Background and Related Work

In this chapter, we present the background material and related work in the areas of parallel programming, transactional memory and dynamic binary rewriting. The chapter is organized as follows: Section 2.1 discusses the main challenges of parallel programming with locks. We focus exclusively on programming of shared memory parallel architectures since this is the communication paradigm chosen for chip-multiprocessors; Section 2.2 introduces transactional memory and explains how it reduces many parallel programming challenges; Section 2.3 describes the design decisions and trade-offs made by designers of software transactional memory systems; Section 2.4 summarizes research which is related to our work; and finally, Section 2.5 introduces the reader to instrumentation through dynamic binary rewriting.

2.1 Parallel Programming

With the advent of the chip-multiprocessor era, software engineers are left with little choice but to start developing parallel applications that can run efficiently on today's parallel processors. While parallel computing has long been regarded as a feasible way of extracting additional performance beyond what a single microprocessor can provide,

little progress has been made at making it easier to develop software that can effectively utilize the additional processor cores present in parallel machines. As such, programming parallel applications with today's programming models is both tedious and error prone. The programmer starts by correctly identifying parallelism within an algorithm, and creates code that leverages it by distributing independent work across the parallel machine, usually, using one thread per processor. For work that must be performed sequentially (e.g.: an update to a shared resource such as a queue), the programmer must use synchronization constructs to coordinate between threads to ensure that the work is never performed concurrently. Typically, programmers use mutual exclusion locks, or mutexes, to create *critical sections* of code, which may be entered by only one thread at a time. To ensure that the programmer's shared data structures are always protected from concurrent accesses, he must associate all data accessed within the critical section with the lock used to specify the critical section, and must preserve this association when constructing other critical sections. Since critical sections may only be entered by one thread at a time, they can be used to guarantee linearizability[22]: all changes made to memory from within a critical section will appear to occur instantly, or *atomically*, to other threads so long as their accesses are also contained within critical sections protected by the same lock. If however, a sequence of code accessing shared memory is mistakenly left unprotected, it may incorrectly observe an intermediate state resulting in a *race condition* that often causes a bug. Such bugs can be hard to fix because their nondeterministic nature makes them difficult to reproduce.

Critical sections can be protected by either a single or multiple locks, and more than one critical section can reference the same lock. The number and size of the critical sections referencing a single lock determines the coarseness of the locking strategy employed. Coarser strategies are generally easier to program but suffer from poor scalability since more code is executed sequentially. Better performance can be obtained using a

fine grained strategy. Fine grained locking improves performance without sacrificing linearizability by carefully allowing non-conflicting operations on shared data structures to execute in parallel. Unfortunately, fine-grained locking is notoriously difficult to get right and can be onerous to debug. Furthermore, fine-grained locking introduces additional synchronization overhead which may offset some, if not all, of the performance benefits gained from any increased level of concurrency. Choosing the appropriate locking strategy for any given problem is both difficult and time consuming, and is further complicated by the following additional problems and challenges of the lock-based programming model.

Deadlock: Deadlock occurs when two or more threads each hold locks that are required by other threads to proceed, causing a circular dependence which cannot be satisfied. Since each thread waits until another releases its lock, no threads can make progress and the application hangs. Deadlock can easily arise when using fine-grained locking if a strict order of lock acquisitions is not enforced. If such an ordering cannot be ensured deadlock detection and resolution schemes can act as a backup measure; however, such schemes are difficult to implement and susceptible to livelock—where threads repeatedly interfere with each other preventing progress.

Convoying: Convoying occurs when a thread is de-scheduled while holding a lock. While sleeping, other threads execute until they need the lock, eventually causing many threads to wait on the same lock. When the lock is released, the waiting threads will all contend for the lock causing excessive context-switching and/or cache-coherence traffic. Unlike under deadlock, the application will continue to make progress, though at a slower pace.

Priority Inversion: Priority inversion is a scenario where a lower priority thread holds a lock required by a higher priority thread. Since the higher priority thread cannot continue to execute until the lower priority thread releases the lock, it is effectively

demoted to the priority level of the second thread temporarily. If an additional thread of medium priority is present, it can delay the low priority thread and therefore the high priority thread as well, thus inverting the medium and high priorities. Priority inversion poses a problem for realtime systems since the higher priority thread may be prevented from meeting its response time guarantees. For general purpose computing, high priority threads are often used in user interaction tasks. While not critical for program correctness, a reduction in priority can impact the perceived performance of an application.

Composability: Lock-based code is not composable. In other words, it is often impossible to combine lock-protected atomic operations into larger operations that remain atomic. This can be easily appreciated by considering the following example originally presented by Harris *et al.* [19]. Suppose we have a hash table with thread-safe insert and delete operations, and assume that given two instances of the hash table, we would like to atomically transfer an element from one to another. Unless the hash table provides a means to manually lock and unlock each of the tables, there is no way to ensure that both tables are protected for the duration of both operations. Protecting both tables is required to prevent other threads from observing the intermediate state, which, in this case, occurs when neither table contains the element being moved. While exposing the hash table's locks can help solve the problem, it is undesirable because it breaks the hash table's abstraction, and opens the door to deadlock and race condition bugs.

Fault Tolerance: Finally, lock based code is vulnerable to failures and faults. If a single thread fails while holding a lock, all other threads will eventually fail to make progress once they require the lock. As the number of processors available in parallel machines continues to grow, failures become increasingly likely.

Due to these problems, lock-based parallel programming does not appear to be a

viable paradigm suitable for the average programmer. If chip-multiprocessors are to succeed, we have to make them easier to program efficiently.

2.2 Transactional Memory

Transactional memory (TM) has been proposed as a promising alternative to lock-based programming. Under the transactional memory model, programmers enclose multiple statements accessing shared memory into *transactions*. As with lock-based critical sections, changes made within the transactions appear to occur instantly to other transactions; however, unlike lock-based code, atomicity is not guaranteed through mutual exclusion, but rather through optimistic concurrent execution that can be rolled back in the event of concurrent non-linearizable operations. Therefore, transactional memory synchronizes concurrent operations only when they conflict, as opposed to when they *may* conflict.

To support optimistic execution, transactional memory systems implement the following mechanisms: *isolation*, *conflict detection*, and *commit/abort*. Isolation allows transactions to execute concurrently without interfering with one another. The conflict detection mechanism monitors all concurrent transactions to determine whether the combined modifications are linearizable. If not, a *conflict* is said to occur and the conflicting transaction is *aborted* and must be re-executed. Aborting requires that all changes made by a transaction are undone. Finally, when a transaction finishes executing and ensures that it is not in conflict with any other, it atomically commits to make its modifications visible to all other threads.

Programming with transactions provides several benefits over lock-based code. The programmer does not have to associate data with different locks: the transactional memory system will ensure that all shared data accessed within any transaction is protected

correctly. Perhaps most importantly, transactional memory's optimistic concurrency model can efficiently execute coarse-grained transactions without loss of scalability. Thus, programmers using transactional memory need not worry about choosing an optimal locking granularity for their application, but are instead free to create transactions that best match their application's software architecture and abstraction barriers. Additionally, transactions can abort and re-execute at any moment, thus deadlock prevention is easily supported. Furthermore, because each transaction can execute without waiting for another to complete, convoying is avoided. Likewise, priority inversion is avoided since a lower priority thread cannot block a higher priority thread; additionally, if a lower priority thread causes a higher priority thread to abort, the higher priority thread can re-execute quickly without further delays. Transactional memory can support transaction nesting allowing programmers to compose multiple transactions by wrapping them into a new outer transaction. Thus, programmers are able to control atomicity in a modular manner that respects abstraction barriers. Finally, transactions provide a much higher degree of fault tolerance as compared to lock-based code, since transactions can roll back in the event of an error.

While many agree on the benefits of transactional memory, the underlying mechanisms used to implement TM systems continue to be the subject of research and development. Many proposed systems exist, ranging from full hardware approaches [6, 17, 30] to full software solutions [12, 15, 20, 27, 28, 37, 41]. In between these two are systems implemented in both hardware and software, which are known as hybrid TMs [9, 24, 39].

Hardware TMs offer exceptional performance but often lack generality. *Bounded* HTMs enforce strict limits on transaction working set sizes, guaranteeing that only transactions with working sets under a certain size will be able to commit. The original HTM proposal followed this approach, using a fixed-size associative memory to temporarily store a transaction's memory writes [21]. *Best-effort* HTMs, on the other hand, isolate

writes by leveraging available memory already present in L1 or L2 caches. Such HTMs are easy to implement by making simple modifications to cache-coherence protocols; however, since data caches are never fully-associative, best-effort HTMs can offer no guarantees as to whether a transaction will be able to fully execute. More recently, HTMs supporting *unbounded* transactions have been proposed; however, such systems are significantly more complex. The cost of this additional complexity is difficult to justify, given the general lack of experience in using transactional memory within a large body of software.

Software transactional memory systems implement TM mechanisms in software without imposing any hardware requirements. Since all mechanisms are implemented entirely in software, STMs can offer good flexibility and generality. STMs are free to adapt to different workloads, using whichever software implementation best matches the target application's execution patterns and levels of contention. Additionally, support for unbounded transactions comes naturally with STMs, since software is not limited by the size of data caches or other on-chip resources. Unfortunately, since STMs implement all communication and low-level TM mechanisms in software, they incur significant overheads which cannot be eliminated. Despite this disadvantage, they remain of great interest to researchers as they offer an excellent platform to experiment with TM programming models on today's hardware. Furthermore, STMs are used extensively in hybrid transactional memory systems [10, 24]. When combined with best-effort HTMs they can be used to provide support for unbounded transactions without the need for additional complex hardware. On such systems, small transactions benefit from the low overhead of HTMs, while less common larger transactions fall back onto the slower but unbounded STM. This model of execution is especially appealing since it allows new hardware to be introduced incrementally, with lower development and testing costs and reduced risk. As a result, STMs can provide a crucial foundation upon which future more-efficient HTMs can build on. For these reasons, we have chosen to focus on STM systems in this dissertation.

2.3 The STM Design Space

In this section we describe several STM design decisions and trade-offs that have been explored in the literature.

Transaction Granularity: Software transactional memory systems can detect conflicts at various granularities. Detecting conflicts at a *word-level* granularity offers the highest accuracy, but incurs excessive communication and bookkeeping costs. By increasing the granularity of the conflict detection strategy, these overheads can be reduced, though at the risk of incurring *false conflicts*. Introducing false conflicts can be undesirable because they degrade performance by making transactions abort when no real conflict occurred. One method of increasing the conflict detection granularity, is to divide memory addresses into a finite set of *strips*. In this approach, a carefully constructed hash function maps memory locations to separate strips which have their accesses monitored to track whether transactions conflict. Alternatively, for STMs that target object oriented applications, the granularity can be increased by tracking conflicts at the *object-level*. Under this approach, false conflicts occur when two transactions concurrently modify different member fields within the same object.

Concurrency Control: To make a persistent change to memory, transactions must first acquire the object, strip, or word location associated with the memory they need to modify. An STM has the choice of letting transactions acquire these locations at the time of a first write, or at commit-time. The former is known as *eager acquire*, while the latter is referred to as *lazy acquire* [27]. Since acquisitions announce a transaction's intention to modify memory, they are used for conflict detection. Therefore, eager acquire allows transactions to detect contention sooner; however, since eager acquire eagerly detects conflicts between all transactions irrespective of whether they will complete, it detects

conflicts that may not materialize: for example, if a transaction causing a conflict fails to commit for some other reason.

Reader Visibility: In addition to communicating writes through location acquisitions, transactions can opt to communicate their reads to others as well. With this approach, referred to as *visible-readers*, transactions can quickly determine whether a store will cause a conflict between themselves and another concurrent transaction. By rolling back on such stores, STMs guarantee that no transaction will operate on inconsistent data. Such a guarantee is necessary for software transactional memory systems, as it prevents threads from mistakingly performing illegal operations such as a segfault or divide-by-zero error, or worse yet, jumping to code that cannot be undone because it was not compiled for transactional execution. With *invisible-readers* reads are not reported to others and are instead stored locally to verify against location acquisitions. Performing reads invisibly drastically reduces communication overheads, but allows transactions to operate on inconsistent data. Consequently, STMs employing invisible-readers incrementally validate their transactions after each load. Validation is performed by either expensive full read-set verification [27, 28], or by looking up a centralized global commit counter to ensure that all data being read was updated before the transaction began executing [11, 41].

Undo-logging and Write-buffering: To implement rollback support, STMs have the option of one of two logging mechanisms: write-buffering or undo-logging. Under write-buffering, each transaction caches its stores into a software *write-buffer*. Upon commit, the data within the buffer is copied to its required locations throughout memory. If the transaction aborts, rolling back is as easy as clearing the buffer and restoring the architectural state of the processor. With undo-logging, transactions directly modify shared memory, and keep a log saving the contents of the overwritten memory so that

it can be restored in the event of an abort. Since each transaction executes in place, committing is cheap: the undo-log is simply discarded; however, the cost of rolling back increases since all writes have to be undone. Write-buffering incurs slightly more runtime overhead than undo-logging since all loads must lookup the write-buffer to ensure that they never read stale data. This check is typically performed by a hash table lookup or bloom filter, which when performed repeatedly can significantly degrade performance. Write-buffering is sufficient to completely isolate transactions from one another, allowing multiple transactions that write to the same location to execute with higher concurrency. When combined with lazy acquire, write-buffering allows certain combinations of conflicting transactions to execute concurrently and commit successfully, so long as their commits are sequential with an order that satisfies the dependencies violated by the conflicts. Undo-logging, on the other hand, must always be combined with eager acquire to ensure that transactions do not interfere with one another while executing. The general rule of thumb is that undo-logging performs better when few conflicts are expected, while write-buffering is more desirable for high contention workloads.

Synchronization: An important trade-off made by today's STM designers is the liveness vs. performance trade-off. For an STM to offer all of transactional memory's robustness benefits, it must be implemented with *non-blocking* synchronization. Two variants of non-blocking STMs have been proposed: *lock-free* and *obstruction-free* [15, 20]. With lock-freedom, the STM system guarantees that one thread can always make progress in a finite amount of time, even in the case of contending concurrent operations or thread failure. Lock-freedom is implemented by allowing threads to *help* others commit their results. Obstruction-freedom is weaker than lock-freedom. It guarantees progress in the presence of thread failures, provided that no contention exists. Under this approach, transactions are able to abort other transactions preventing deadlock, but risking live-

lock. While the benefits of non-blocking STMs are extremely desirable, their overheads limit their application. *Blocking* STMs offer superior performance but risk deadlock in the presence of thread failure and are susceptible to convoying and priority inversion. Blocking transactions were originally championed by Ennals [14] who argued that convoying and priority inversion will be rare for multi-threaded applications executing on chip-multiprocessor hardware, since each thread executes privately on a processor with minimal context switching. Therefore, if a programmer is willing to exchange some degree of fault-tolerance for substantially better performance, blocking can be desirable. Such a trade-off can be acceptable for now to allow programmers to gain experience with transactional memory.

Atomicity: Whether or not a transaction appears to complete atomically to non-transactional code determines the level of *atomicity* employed by an STM system. *Strongly atomic* systems guarantee atomicity between transactional and non-transactional code, while systems that provide atomicity between transactions only are said to offer *weak atomicity* [5, 38]. To date, most STMs implement weak atomicity due to the large cost of supporting the stronger consistency model, which can be considered equivalent to a weakly atomic system with all non-transaction instructions placed in their own single-instruction transactions.

Under weakly atomic systems, simple single instruction accesses to shared data from outside of a transaction can lead to unexpected, implementation dependent behaviour. Figure 2.1 (adapted from [38]) presents a number of such accesses. In Figure 2.1(a), Thread 1 will be unable to detect the write performed by Thread 2 and will fail to abort and re-execute, thus, potentially allowing Thread 1 to observe two different values for `x`. In Figure 2.1(b), Thread 1 will once again fail to detect the conflict created by Thread 2 allowing `x` to be assigned the value of `1`. This result cannot occur if both snippets of code

Assume that initially `x==0`

```
// Thread 1  // Thread 2
atomic {     ...
  t1=x;      ...
  ...       x=100;
  t2=x;      ...
}            ...
```

`t1` might not equal `t2`

(a) Non-repeatable reads

```
// Thread 1  // Thread 2
atomic {     ...
  t=x;       ...
  ...       x=100;
  x=t+1;     ...
}            ...
```

`x` might equal `1`

(b) Lost updates

```
// Thread 1  // Thread 2
atomic {     ...
  x+=1;      ...
  x+=1;     t=x;
}            ...
...          ...
```

`t` might equal `1`

(c) Dirty reads

Figure 2.1: Atomicity violations on an STM system.

```
// Thread 1
Node* priv;
atomic {
  priv = head;
  if (head != NULL)
    head = head->next;
}
if (priv) {
  t1 = priv->val;
  t2 = priv->val;
}
```

```
// Thread 2
atomic {
   Node* n = head;
   while (n != NULL) {
     n->val += 1;
     n = n->next;
   }
}
```

can `t1!=t2`?

Figure 2.2: Potentially unsafe privatization of node in linked list.

were executed sequentially. Finally, in Figure 2.1(c), Thread 2 may incorrectly observe the intermediate state of `x==1`, which would be unobservable if both threads executed one after another. While these examples will also cause unexpected results when using lock-based synchronization, next we show that this need not necessarily be the case.

Figure 2.2 (adapted from [9]) presents code where a node in a linked list is privatized by a transaction (Thread 1), while the node's value is incremented by another (Thread 2). Once privatized, Thread 1 reads the privatized node's value multiple times outside of the transaction, assuming that it is safe to do so. Indeed, when using lock-based critical sections to protect the privatization and increment code, the privatization works

as expected; however, when using a weak atomicity TM, Thread 1 may commit its transaction without noticing that Thread 2 is operating concurrently on the values of the linked list's nodes. Since Thread 1 does not read the `val` field of its privatized node inside its transaction, it cannot detect the conflict introduced by Thread 2. Furthermore, since Thread 2 may have started executing before Thread 1 updated the head pointer, it can potentially reach the privatized node and increment its value, even after Thread 1 commits its transaction[1]. As a result, a weakly atomic TM cannot guarantee that multiple reads operating on the privatized node's value will yield the same result. This result is especially troubling for transactional research as it implies that weakly atomic TMs can at times be *less* reliable than lock-based code. Strong atomicity offers a solution to this problem but at significantly greater implementation complexity and, in the case of STMs, significantly more overhead.

2.4 Closely Related Work

In this section, we examine the closely related work in the field of transactional memory. Specifically, we focus on STM systems that target `C` or `C++` code, since such TMs are most applicable to the majority of programmers. We start with STMs that require programmers to hand modify their existing code, and end with systems that are either already fully automatic, or designed with automatic compilation in mind. Lastly, we examine an HTM that can execute system calls.

OSTM: Fraser and Harris proposed OSTM [15], a highly efficient STM that targets object-oriented `C` code. To use OSTM, programmers are required to manually wrap all objects that they would like to access from within a transaction into special *object handles*, which they must explicitly allocate and deallocate. Next, they must inform

[1]The exact order of events depends on whether the TM implements write-buffering or undo-logging.

the STM of all reads and writes that they intend to make to their objects, by using `OSTMOpenForReading()` and `OSTMOpenForWriting()` accessor functions. OSTM tracks and communicates these accesses through a *transaction record*, which operates at an object granularity. All writes to objects are write-buffered using object clones, which are announced to other transactions only at commit time (lazy acquire), while reads are never communicated (invisible-readers). OSTM uses the level of indirection provided by the object handles to support efficient atomic commit. Cloned objects are committed by simply swapping them in using a *compare-and-swap* (CAS) operation on the object pointer in each object handle. Finally, OSTM is non-blocking, offering lock-freedom by allowing transactions to help other transactions commit their write-buffers, in the event of thread-error or thread de-scheduling.

While OSTM provides an interesting tool for software engineers to test the transactional memory programming model, it suffers from a number of drawbacks that make it unsuitable for real use. First, as a result of the explicit level of indirection required, the API is cumbersome and verbose. Second, OSTMs lock-free approach incurs substantial overhead, for perhaps less gain than benefit. Finally, in an effort to reduce overheads, OSTM does not incrementally validate its read-set after each read, thus it permits transactions to operate on inconsistent data. In light of this problem, OSTM registers a signal handler with the OS to catch and recover from segmentation faults. While this approach is very efficient, it is not completely safe since transactions may still accidentally jump to non-transactional code: for example, by executing an indirect branch that had its target accidentally overwriting, or by executing a return instruction that had its return address overwritten by code writing to a stack allocated array while using an incorrect index.

RSTM: In an effort to improve on these drawbacks, Marathe *et al.* developed Rochester STM (RSTM) [28]: a flexible and robust STM targeting object oriented `C++` code. As

with OSTM, RSTM requires that programmers wrap their objects in *object headers*. Specifying reads and writes is simpler than with OSTM, due to the use of `C++`. Additionally, allocating new shared objects is easy thanks to a mandatory transactional object base class, which overloads the `new` and `delete` operators to automatically allocate object headers. Like OSTM, RSTM tracks conflicts at an object granularity; however unlike OSTM, RSTM supports both lazy and eager acquire, as well as both invisible and visible readers. When executing with invisible readers, RSTM incrementally validates each transaction's read-set, guaranteeing that transactions never execute with inconsistent data. Interestingly, Marathe *et al.* find that despite the $O(n^2)$ cost of incremental validation, invisible-readers outperformed visible-readers in most cases that they tested. Finally, RSTM reduces complexity and overheads by offering an obstruction-free non-blocking design, which allows transactions to abort—but not help—others, in the event of thread failures or stalls.

RSTM provides an excellent and robust STM that programmers can use to develop transactional memory applications that run on today's computers; however, while greatly simplified, RSTM's API is still considerably involved. Furthermore, due to the fact that RSTM is restricted to object oriented code, it offers little benefit for porting legacy non-object oriented code.

TL: Dice and Shavit introduced Transactional Locking (TL) [12], an STM that targets arbitrary `C` and `C++` code. While not an automatic STM compiler, TL is designed with automatic STM code generation in mind. As the name suggest, TL uses locks to implement the transactional memory mechanisms, and is therefore *blocking*. TL supports conflict detection at word, strip, and object (if available) granularities, with the use of versioned locks that are associated with each transactional shared variable (word, range of memory, or object). TL supports two modes: (i) an eager acquire, undo-logging mode

called *encounter mode*, and (ii) a lazy acquire, write-buffering mode referred to as *commit mode*. Unfortunately, both modes use invisible-readers with no read-set validation, thus permitting transactions to operate on inconsistent data.

TL2: Under a slightly newer design, Dice *et al.* propose the use of a centralized global version-counter to reduce the cost of incrementally validating the read-set after each load. With this approach, all writing transactions increment a global version-counter and store its value in the versioning locks associated with each updated shared variable. Subsequent transactions simply save the global version number when they begin, and verify that each shared variable that they read from has a version number that is less than or equal to the version number saved. This approach greatly reduces the cost of incrementally validating a read-set, since each validation takes constant time. Unfortunately, it does so with the introduction of a highly contended centralized counter. Regardless, TL2 exhibits impressive performance, and we look forward to when it will be integrated into an STM-capable compiler.

McRT STM Compiler: Extending the McRT STM [37], Wang *et al.* were the first to develop a fully-automatic STM compiler [41]. The modified version of Intel's `icc` compiler is able to mechanically rewrite `C` or `C++` code to include calls to their McRT STM runtime, enabling transactional execution. The McRT STM supports only one mode of execution, which is most similar to TL2's "encounter mode". It uses undo-logging with eager acquire, invisible-readers, and maintains a global version-counter for faster read-set validation. Programming with the STM compiler is easy—developers simply enclose transactions in code blocks that have been demarcated with a new `#pragma tm_atomic` construct. Additionally, the programmer annotates all functions that might be called from within a transaction with the `tm_function` pragma. Unfortunately, since the STM can only

execute functions that have been rewritten by the compiler to support transactional execution, it does not permit transactions to call pre-compiled legacy libraries. Such libraries comprise a critical building component to many of today's applications, and their exclusion limits the applicability of the compiler.

Concurrent with this research, Ying *et al.* have performed preliminary work to augment the McRT STM runtime with a dynamic binary rewriter [42]. Their system can dynamically rewrite pre-compiled legacy binaries to execute with transactional semantics; however, it sidesteps the problem of supporting system calls, citing open-nesting as a possible solution to this difficult problem.

Unrestricted TM: We believe that support for system calls within transactions is absolutely necessary for the long term success of transactional memory. We are joined in this opinion by Blundel *et al.* [6] who argue that support for system calls is critical, since programmers are typically unaware of when their code may make such calls. They state that this is especially the case when compiling and linking system components and libraries separately. Blundel *et al.* propose 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. We find this approach inspiring.

2.5 Dynamic Binary Rewriting

In this section, we introduce the reader to dynamic binary rewriting, a key process that enables us to support pre-compiled legacy libraries in our transactional memory system.

Dynamic binary rewriting (DBR) is a well established method of transforming applications to include new code, or *instrumentation*, to observe or slightly modify the behaviour of the application. DBR has numerous applications. It is used by VMware

to virtualize operating systems, allowing them to execute in user-space under a second host operating system. VMware dynamically rewrites the guest operating system so that protected instructions are replaced by instructions that call the host operating system, requesting that the protected operations be performed on the guest's behalf. Alternatively, DBR provides an excellent means to debug or profile applications. Users can select from a number of instrumentation tools such as Pin [26], DynamoRIO [8], or Valgrind [32], which provide easy to use APIs that can be used to specify arbitrary instrumentation.

Dynamic binary rewriting offers a number of advantages over its static counterpart. Most importantly, DBR does not need to determine static basic block boundaries and control flow graphs to determine where it is safe to insert instrumentation. Doing so statically can be very difficult for low level machine code, due to an abundance of indirect branches. DBR discovers basic blocks and indirect branch targets at runtime, by following the application's execution. Detecting code at runtime is crucial for applications that dynamically generate code, or call dynamically-loaded shared libraries. Furthermore, since DBR never modifies the original code, applications that make assumptions about the original structure of the binary will continue to execute correctly. This is especially important for applications that patch or self-modify themselves.

At the heart of any dynamic binary rewriting system is the just-in-time compiler, or JIT for short. A binary rewriting JIT works much like a Java bytecode JIT (e.g., Sun's HotSpot [34] or IBM's Jalapeno [1]), or the JIT in a dynamic binary translator (e.g., QEMU [3]). A Java JIT translates bytecode to native code to improve performance, and binary translators typically translate from one ISA to another for a host of reasons, such as running legacy binaries on new hardware. The binary rewriting JIT, in contrast, produces code in the same ISA as its input, but with additional instrumentation instructions inserted. A JIT can compile at various granularities; a Java JIT typically compiles at a method granularity, while instrumentation tools and binary translators prefer basic

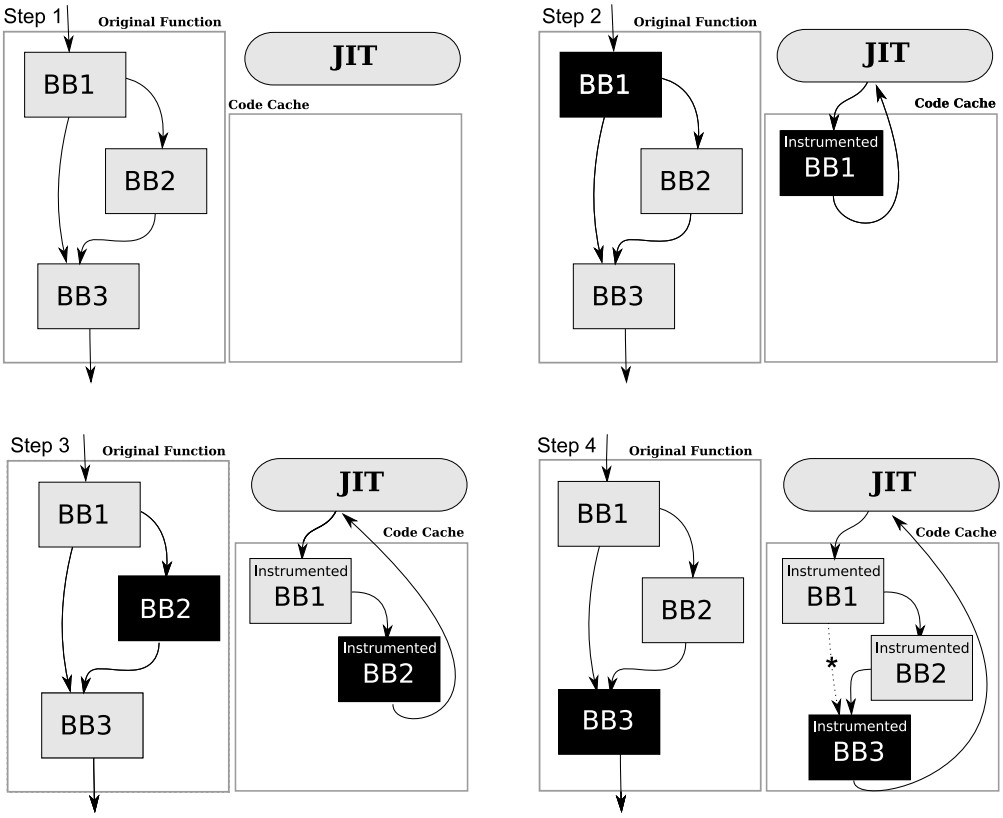


Figure 2.3: A brief look at the JIT process. Step 1 shows the initial state. Step 2 shows the result of copying the first basic block into the code cache. Step 3 shows how the branch targets of existing cached blocks are updated as destination blocks are brought into the code cache. Step 4 shows the final result, with all three basic blocks now instrumented and executing from the code cache.

block or trace granularities. The resulting instrumented basic blocks are stored in a code cache, from whence they are dispatched instead of the original code.

Figure 2.3 shows the dynamic binary rewriting process for a simple graph of three basic blocks. Initially, in Step 1, no code has been instrumented and the code cache is empty. When execution reaches `BB1`, it is redirected into the JIT. The JIT makes a copy of `BB1`, inserting the required instrumentation and modifying the final instructions to first record the original destination, `BB2`, and to then redirect execution back to the JIT at the end of the block. The new block is then placed into the code cache, as shown in

Step 2 of Figure 2.3. When control returns to the JIT after executing the instrumented `BB1`, the JIT sees that the next block to execute is not in the code cache. Therefore, the JIT copies `BB2` into the code cache, transforming it as with `BB1`. Additionally, the JIT updates `BB1` to branch directly to the instrumented copy of `BB2`, rather than the JIT. The result is shown in Step 3 of Figure 2.3. This process continues in Step 4 as `BB3` is discovered, instrumented, and placed in the code cache. Note that had control flowed directly from `BB1` into `BB3`, through the dotted branch marked with a * in Step 4, then `BB2` would never enter the code cache at all; this is the nature of dynamic binary rewriting—only executed code is instrumented.

Since the resulting instrumented code is stored and executed from a new location, the JIT must modify all control instructions to account for this change, as shown in the simple example of Figure 2.3. Direct control instructions can be updated to point straight to their code cache equivalent targets, if present. Indirect control instructions such as indirect jumps, calls, and return instructions have targets that cannot be resolved at JIT-time, and therefore must be translated at runtime. These instructions are modified to point to a translation function or stub code, which invokes the JIT if the target is not in the code cache. Since the binary rewriting systems always ensures that it gains control before executing new code, it offers a degree *sandboxing* which can be used to prevent the application from accidentally executing unintended code.

To initiate binary rewriting, an entry point from which control can be redirected to the JIT is required. After that point, all further execution occurs within the JIT engine. A number of methods exist for gaining control. One approach, taken by Pin, is to gain control using `ptrace`: the library that is used by debuggers to attach themselves to target applications and to insert breakpoints. Pin injects its JIT into the address space of the application (using the same primitives used to insert breakpoints), and modifies the application program counter to point to it. Alternatively, application developers with

access to source code can compile and link their applications with a binary rewriting system. With this approach, available as an option in DynamoRIO, programmers can call start and exit functions to transition between native and DBR execution.

2.6 Summary

This chapter introduced the reader to the problem of parallel programming with locks, and presented transactional memory as a possible solution. We motivated our interest in software—as opposed to hardware—transactional memory, and introduced the reader to the terminology and basic concepts in the area of STMs. We also surveyed the related STM research that focuses on the popular, but difficult to support, `C` and `C++` programming languages. Finally, we concluded with a brief introduction to the process of dynamic binary rewriting. The next chapter will describe the design and implementation of our dynamic binary rewriting system that we used to develop our software transactional memory system.

Chapter 3

The Judo Dynamic Binary Rewriting System

In this chapter we present Judo: our x86 dynamic binary rewriting framework. Like other DBR systems, Judo lazily rewrites target applications *just-in-time* (i.e., right before the code is about to execute) into a *code cache* from where they are executed. Within the code cache, Judo is free to augment rewritten code with arbitrary instrumentation for a variety of purposes. Judo was originally developed by Olszewski *et al.* for JIFL [33]: a dynamic binary instrumentation framework for kernel-space; however, it has since grown into a high performance user-space instrumentation tool.

3.1 Design

In this section, we take a closer look at the design and implementation of our binary rewriting framework.

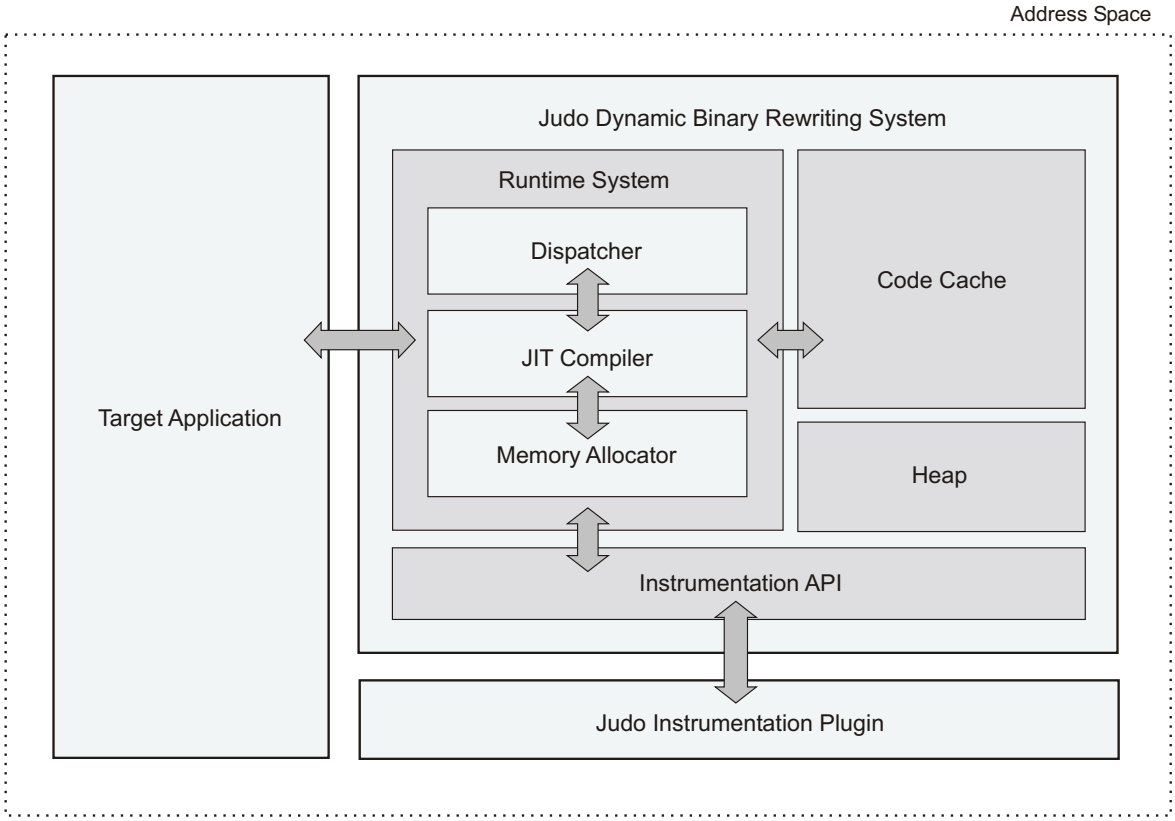


Figure 3.1: Judo's software architecture.

3.1.1 System Overview

Figure 3.1 illustrates Judo's software architecture. At the highest level, Judo consists of a runtime engine, a private heap, a code cache and an instrumentation API for interfacing with Judo instrumentation plugins that dictate what instrumentation to insert into the rewritten application. The runtime engine contains the JIT compiler, dispatcher, and a custom memory allocator.

To execute the target application through the dynamic binary rewriting system, Judo must load itself into the application's address space and obtain control of the thread. Judo achieves this in one of two ways: 1) It can be compiled as a Pin instrumentation plugin (or pintool), and have Pin perform the address space injection using `ptrace`. Once loaded,

Judo requests that Pin instrument the application's first instruction with a call to a routine that takes control of the thread and hands it to Judo. 2) Alternatively, Judo can be built as a static library that developers can link with directly. Under this approach, programmers call `judo_start()` and `judo_stop()` functions to transition between native and instrumented code. Finally, since Judo instrumentation plugins are compiled as separate binaries, they too must be loaded into the target application's address space. Upon initialization, Judo uses the Unix `dlopen()` function to load and link the plugin binary.

In the rest of this section, we take a closer look at Judo's runtime engine. Specifically, we describe the design of the dispatcher, JIT compiler, and memory allocator, and present a number of optimizations that improve the quality of the rewritten code.

3.1.2 The Dispatcher

The dispatcher is responsible for saving and restoring the processor's architectural state, as well as locating (via a hash table) and redirecting control to the code cache version of the next instruction to be executed. If the required code does not exist in the code cache, the JIT compiler is invoked to insert it there. Due to the fact that some of these tasks are low level, the dispatcher is written in assembly.

3.1.3 The JIT Compiler

The JIT compiler copies the target application's machine code into the code cache, while instrumenting it as specified by the Judo plugin. Similar to both Pin [26] and HDTrans [40], Judo JIT compiles (or JITs) at a *trace* granularity. These traces should not be confused with DynamoRIO's dynamically profiled hot traces [8]—instead, Judo selects traces statically at JIT time, although their selection can be indirectly influenced by pre-

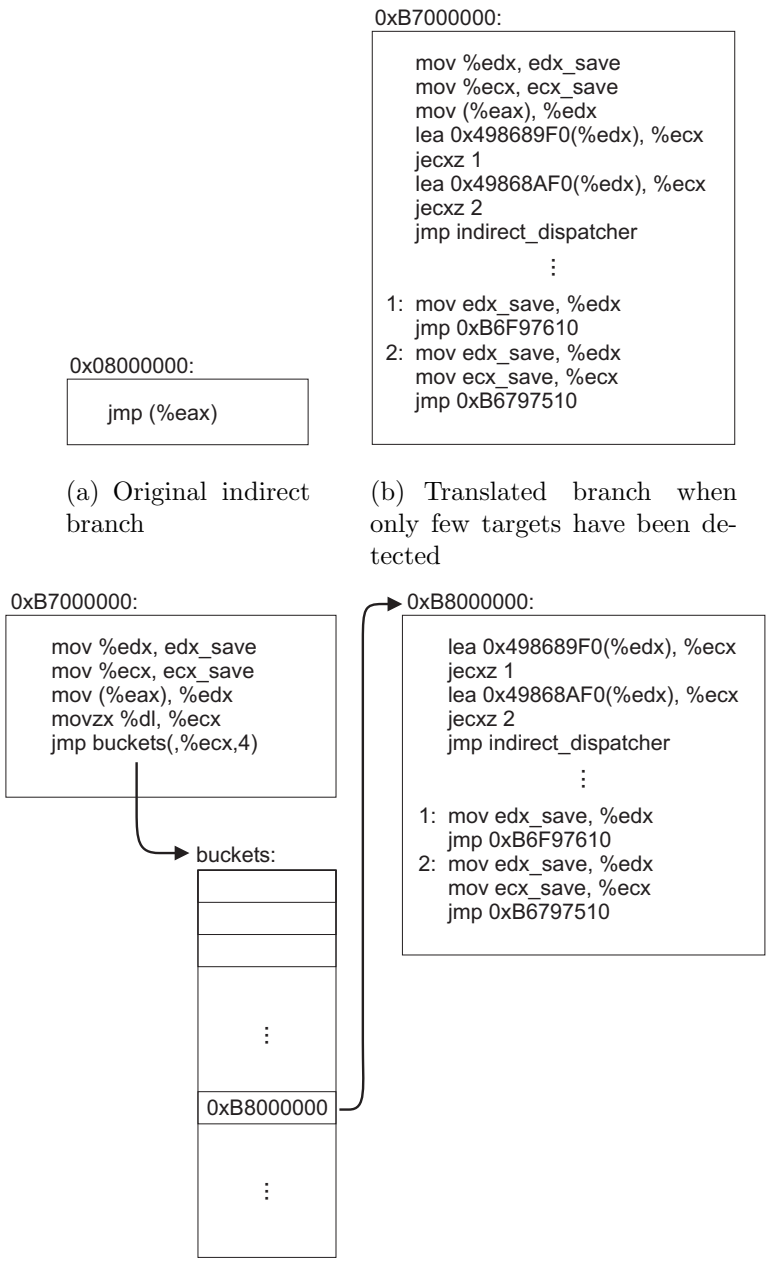
viously executed code through the current contents of the code cache. Many trade-offs exist when deciding trace sizes. JITing at a basic block granularity will reduce the size of the code cache, since only code that is guaranteed to execute is rewritten and cached. However, basic block JITing can introduce 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 terminated 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 (and therefore placed adjacent to one another). Of related schemes, this trace selection strategy is the most similar to that of HDTrans [40], except that HDTrans does not restrict the number of 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. Finally, if the instrumentation plugin does not specify basic block level instrumentation, Judo minimizes code duplication by allowing branches to target the mid-points of JITed traces.

When JITing a trace, all control instructions have to be modified to point to their equivalent code cache targets. Direct branches, such as the x86 `jcc`, `loopcc`, and `jmp` instructions, are modified so that control is redirected to the dispatcher. `Call` instructions are converted to `push` and `jmp` instructions. To preserve the contents of the stack,

the original non-instrumented return address is pushed to the stack. This serves two purposes. First, it enables Judo to detach itself at any time (in the event of an error) by returning control to the original non-instrumented code. Second, it ensures that any code that depends on the value of this return address will continue to function correctly. For example, call instructions can be used to push the value of the program counter to the stack so that it can be read. Any code depending on this method for retrieving the contents of the program counter will continue to function correctly since the original return address is still pushed to the stack. Indirect `jmp` and `call` instructions are modified in a similar fashion to their direct counterparts, however the address of the next instruction passed to the dispatcher is no longer a constant but rather calculated at runtime. Return instructions are handled like indirect jumps. Their branch targets are also runtime dependent and are obtained by popping the return address off the stack.

As an optimization, the Judo JIT attempts to directly link compiled traces whenever possible. Judo checks if the branch or fall-through target of a control instruction is already in the code cache, and if so, it emits a jump instruction to jump directly to the existing trace. Additionally, while JITing a new trace, Judo patches all previous control instructions that target the trace so that their subsequent invocations avoid the dispatcher. For indirect control instructions (such as the indirect branch pictured in Figure 3.2(a)), we apply *predicated indirect branch chaining*, a popular method of linking commonly-occurring indirect targets. Under this approach, a sequence of target address comparisons is built incrementally at runtime. Each comparison checks whether the current target address is equal to one observed previously. If the comparison succeeds, the code jumps to the code cache version of the target, otherwise, the next comparison is executed. If all comparisons fail, the dispatcher is called to locate the desired trace. Additionally, the dispatcher keeps track of the number of times that each target is taken. If a threshold is reached, a new comparison check is added to the sequence.



(a) Original indirect branch

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

(c) Translated branch when many targets have been detected

Figure 3.2: Predicated indirect branch chaining in Judo.

Judo performs the comparisons using the widely adopted `lea/jecxz` approach originally used in DynamoRIO [8] that does not affect the condition flags. Each comparison requires two registers which Judo frees-up by spilling the resident values to global vari-

ables (`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 hot targets are detected. However, unlike previous systems, Judo does not insert the new comparison instructions into dynamically allocated memory, which requires that the checks be linked together with jump instructions. Instead, the new instructions are emitted adjacent to one another in pre-allocated memory (as shown in Figure 3.2(b)), thus improving code cache locality and eliminating the cost of the extra jump instructions. Luk *et al.*, state that the dynamic chains are desirable because they allow new target comparisons to be inserted in any order [26]. We have been unable to verify this claim in our own experiments. In fact, we found that the linear pre-allocated memory approach outperforms predicated chains of dynamically allocated memory, even when elements are periodically reordered using information obtained from dynamic branch target profiling.

If a substantial number of unique hot targets are encountered for a specific branch, Judo reduces the number of comparisons required to quickly translate the target addresses by rewriting the predicated indirect chaining code into a local *executable hash table* lookup (shown in Figure 3.2(c)), which partly resembles the global *sieve* used in HDTrans [40]. Like the sieve approach, we use the `movzx` (move and zero-extend) instruction, which 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.

While compiling traces, the JIT will also emit any desired instrumentation, which is often as easy as inserting call instructions to the instrumentation routines into the traces. However, instrumentation routines may modify the state of the processor, and therefore, Judo must emit instructions that save and restore register and condition code states as well. As an optimization, Judo performs register and eflags (x86's condition code flags) liveness analysis to reduce the number of such instructions. Further, if the

instrumentation routine is small enough, the JIT will attempt to inline it into the trace. The following sections describe these optimizations in detail.

3.1.4 Register and Eflags Liveness Analysis

To eliminate redundant state-saving instructions, liveness analysis is performed on the target applications binary code to determine the minimum set of condition code flags (eflags) and registers that need to be saved. Liveness analysis proceeds by disassembling instructions following the desired position of instrumentation and checking whether registers or eflags are overwritten (i.e., *killed*) before being used as input. Those that are used before being killed are *live*, and are considered vulnerable if the instrumentation modifies them. If a direct control instruction is encountered, its target basic block is analyzed in the same fashion. Repeated analysis of the same basic block is avoided by entering the address of an already-analyzed basic block in a dedicated hash table. Indirect control instructions are treated conservatively and assumed to lead to a basic block that uses all registers and eflags. The set of vulnerable registers (including the eflags register) must be saved before, and restored after, the instrumentation routine.

When performing instrumentation with basic block granularity, the JIT is free to insert instrumentation anywhere in the basic block. In this case, the JIT will use the liveness analysis results to find the position where the least amount of state needs to be saved.

3.1.5 Instrumentation Inlining

For small instrumentation routines, Judo is able to automatically inline them directly into the JITed traces. While inlining, Judo can also specialize the instrumentation for any parameters that will not change at runtime. For example, when instrumenting the

direction of individual branches, the per branch counter address passed to an instrumentation routine will remain constant for all invocations of each branch and can therefore be propagated into the inlined routine.

To achieve these optimizations, standard compiler optimizations such as constant propagation, constant folding, copy propagation and dead-code elimination are applied. To reduce the complexity of the compiler code, we take the approach of Pin by using architecture-specific optimizations that operate directly on machine code. Inlining begins by placing all instrumentation instructions into a linked list so that they can be better manipulated. Next, Judo generates the static control flow graph—if any indirect jumps are encountered, inlining is aborted since Judo cannot determine the targets of these jumps. Judo propagates constant parameters by converting all move instructions that read constant parameters from the stack, into equivalent moves with immediates as their operands. If all accesses to a parameter are removed, the parameter is also removed and all stack accesses are modified to account for the change. Next, copy propagation is performed to eliminate any needless moves. Dead-code elimination is used to remove the remaining copies if all references to the copied register have been propagated out. Since Judo currently lacks a data-flow solver, these two steps are only performed if the instrumentation routine is composed of a single basic block. Finally, the specialized routine is laid out as a continuous sequence of instructions in memory. Since the sizes of basic blocks most likely will have changed, special care must be taken to patch up the relative branch target offsets of control instructions. Return instructions must also be either removed, or converted to relative jumps that point to the end of the inlined code. The resulting code is cached so that it can be reused if subsequent instrumentation inlining needs to be performed for the same instrumentation routine with the same parameter values.

3.1.6 Memory Allocator

Judo often needs to allocate dynamic memory while performing JIT compilation or analyzing code. Since memory allocators are often not reentrant, Judo must avoid using the target application's allocator as it might be operating on behalf of a thread currently executing its own memory allocation request. Doing so risks deadlock or a corrupt system state. Instead, Judo preallocates and manages its own memory with a custom memory allocator. We found that a simple, slightly optimized, implicit next-fit memory allocator was sufficient for our needs. If at any time Judo becomes low on preallocated memory, it sets a flag to flush its code cache and hash tables to free up more memory the next time the dispatcher is called.

3.1.7 Multi-threaded Considerations

Judo supports multi-threaded target applications through the use of private code caches and heaps. Despite the additional memory pressure, we believe that private code caches are desirable for applications that use modest thread counts since they enable the JIT to specialize instrumentation per thread and almost entirely avoid contention. Each processor also requires a private dispatcher so that it can save state to global memory without needing to check what thread it is running on.

3.2 Evaluation of Judo

In this section, we examine Judo's performance.

3.2.1 Experimental Framework and Benchmarks

We evaluate Judo by comparing it to DynamoRIO when running SpecINT2000 benchmarks. DynamoRIO dynamically detects and optimizes hot traces, making it one of the

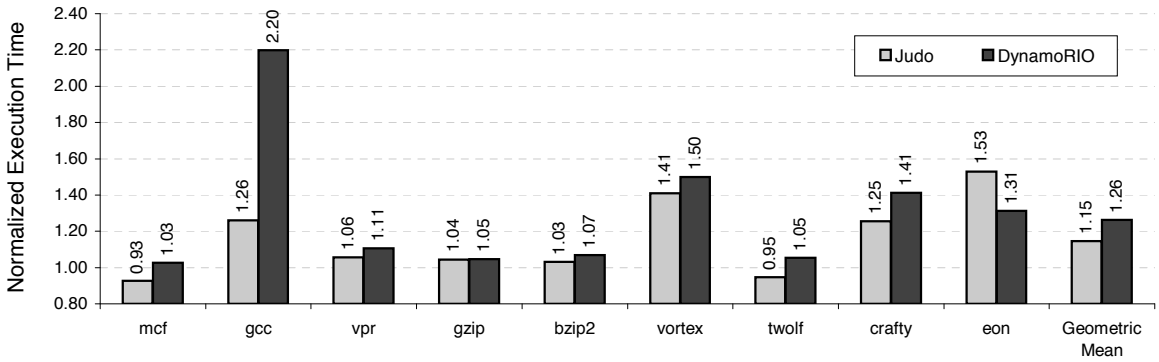


Figure 3.3: Dynamic binary rewriting performance comparison.

fastest DBR systems currently available. To demonstrate best-case performance, we executed all benchmarks without applying any instrumentation, thus we measure the cost of simply enabling instrumentation. Each of the benchmarks were compiled with gcc v3.3.6 at the `-O2` optimization level, and executed with the largest input file. All experiments were performed on an Intel Xeon 2.8GHz processor, containing a 12K μOps instruction trace cache, an 8KB L1 data cache, and 512MB L2 and 2MB L3 unified caches, executing with 4GB of main memory.

3.2.2 Performance

Figure 3.3 presents the performance comparison of the two dynamic binary rewriting systems. As is inevitable with any dynamic binary rewriting system, both Judo and DynamoRIO incur considerable overheads. On average, Judo incurs a 15% loss in performance over native non-DBR execution, while DynamoRIO suffers a drop of 26%.

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. Judo's careful trace selection, along with its novel use of an executable hash table for predicated indirect chaining, helps reduce the cost of executing the rewritten code.

In rare cases (mcf and twolf), Judo can speed up the execution of an application (by up to 7%). We attribute these speedups to 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 the increased available cache space due to their memory intensive nature.

3.3 Summary

This chapter described the design and implementation of the Judo dynamic binary rewriting framework. Judo reduces overheads by rewriting target applications at a trace granularity, and uses a novel optimization to reduce the cost of indirect branch target translation. Consequently, Judo incurs only modest slowdowns when executing SpecINT2000 benchmarks, and can outperform DynamoRIO, a highly optimized dynamic binary rewriting system. These promisingly-low overheads are what originally inspired us to build on Judo to develop JudoSTM.

Chapter 4

The Judo Software Transactional Memory System

In this chapter we present the design of JudoSTM, our dynamic binary rewriting software transactional memory system. JudoSTM uses the Judo DBR framework to automatically instrument transaction binaries with STM instrumentation. Since the instrumentation is applied to the application binary, JudoSTM supports calls to statically linked legacy libraries. Furthermore, since the instrumentation occurs at runtime, JudoSTM can instrument, and thus support calls to, dynamically linked libraries as well.

4.1 Overview of Desirable Features

In addition to the benefits described above, JudoSTM offers the following desirable features.

Privileged Transactions: Similar to the Unrestricted Transactional Memory design by Blundel *et al.* [6], JudoSTM supports a single transaction that can perform system calls. We call such a transaction 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 other non-privileged transactions abort should a conflict be detected. Since system calls escape the control of Judo's DBR framework, we cannot instrument their memory writes to verify the read-sets of other transactions. Therefore, JudoSTM cannot detect conflicts by comparing the version numbers (as used by [12]) of the data accessed by each transaction or by access timestamps (used in [11, 41]). Instead, JudoSTM detects conflicts by using value-based conflict detection, which detects conflicting operations between transactions by comparing the values of the memory locations accessed by each transaction, ensuring that they have not been modified since they were read during transaction execution. Since this strategy allows each transaction to verify its read-set without requiring any information from concurrent transactions, it can be used to detect conflicts even when caused by system calls.

Efficient Invisible-Readers: Invisible-readers have been shown to improve performance by reducing inter-processor communication [28, 37]. If a transaction executes using invisible-readers, it must perform expensive read-set validation after each and every memory load to prevent it from operating on inconsistent data. By executing with inconsistent data, a transaction may perform illegal loads or arithmetic operations, or, worse yet, make control-flow decisions that 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 [15] or with safe load instructions [12]; however, only a truly *sandboxed* transaction can execute safely when operating on inconsistent data. Since JudoSTM transforms *all* transactional code at runtime, it can prevent illegal control-flow which, when combined with trap handlers, is sufficient to safely support invisible-readers without the need for expensive incremental read-set validation.

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 [25]. 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 becomes 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 writing a `1`, assuming the lock is un-contended, and later free it by writing a `0`. When this transaction commits, the original value of the lock location (`0`) is overwritten with the final value (`0`), and therefore the modification is silent. 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.

Thanks to these benefits, JudoSTM permits developers to program transactional memory applications without imposing *any* restrictions on what can be placed within a transaction. To the best of our knowledge, JudoSTM is the first software transactional memory system to offer such freedom.

4.2 Design Decisions

JudoSTM's design is succinctly described as a write-buffering, blocking, lazy acquire, invisible-reader STM using word-granularity value-based conflict detection. The choice of operating at a word-granularity and using write-buffering stem from the decision to

use value-based conflict detection, while the use of blocking synchronization is required for supporting system calls. 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. However, value-based conflict detection can be expensive: unlike location based conflict detection, its overheads cannot be reduced by grouping multiple accesses to adjacent regions of memory to verify them simultaneously (strip and object-level granularity). Instead, a compare instruction must be executed for each word 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.

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. Since invisible-readers have already been shown to perform more efficiently than visible-readers even without sandboxing, we decided not to implement and evaluate a version of JudoSTM that uses visible-readers.

Write-Buffering: We chose to use the write-buffer rollback mechanism instead of an undo-log approach because of the write-buffer's compatibility with value-based conflict detection. While undo-logging has been shown to exhibit impressive results for low

contention applications [14, 41], it suffers significantly in high contention scenarios where many conflicts trigger expensive rollbacks [12]; under these conditions, write-buffering fairs better. Additionally, since write-buffering permits the use of lazy acquire, it can exhibit fewer false non-materializing conflicts than undo-logging. Finally, recent studies comparing the two have shown that write-buffer can remain competitive with undo-logging, even in scenarios with little contention [12].

Lazy Acquire: Judo implements lazy acquire in the hope of improving concurrency and limiting the number of rollbacks caused by false conflicts. Support for eager acquire is planned as future work.

Blocking: To support system calls, which cannot be rolled back, JudoSTM requires a blocking design. Unfortunately, blocking does reduce dependability, by allowing deadlock in the case of thread failure. Recent work by Ennals argues that this cost is worth the performance benefits of blocking STMs [14], and many recent STMs [12, 37, 41] have since implemented blocking designs.

4.3 Implementation

In this section, we describe how we built on the Judo DBR framework to implement JudoSTM.

4.3.1 System Overview

JudoSTM is implemented as a self-contained static library which can be linked with any application that desires transactional memory constructs. When creating transactions, the programmer writes regular `C` or `C++` code, without annotations or calls to an STM runtime. Furthermore, he is free to compile the program using the compiler of his choice.

Once compiled, transactions will not execute correctly unless they are dynamically instrumented by JudoSTM at runtime. To achieve this, transactions use the `judostm_start()` and `judostm_stop()` functions to switch between native and dynamically instrumented execution, when they start and finish. Since non-transactional code need not be instrumented, it does not have to execute through the Judo DBR system, and thus, is not affected by DBR overhead.

When a thread first enters `judostm_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.

JudoSTM introduces a number of changes to the Judo software architecture (illustrated in Figure 4.1). Since Judo does not yet perform full specialization when inlining instrumentation routines consisting of more than one basic block, we chose to directly augment the Judo JIT compiler with the JudoSTM instrumentation plugin. The plugin specifies the changes that are needed to transform the transaction so that it executes through a write-buffer, saving its reads, and later validating and committing what it saved. For instrumentation that is expected to execute often, the plugin directly emits the instructions into the transaction instruction stream; otherwise, it emits call or jump instructions to redirect control into an STM runtime library, which contains the bulk of the STM logic.

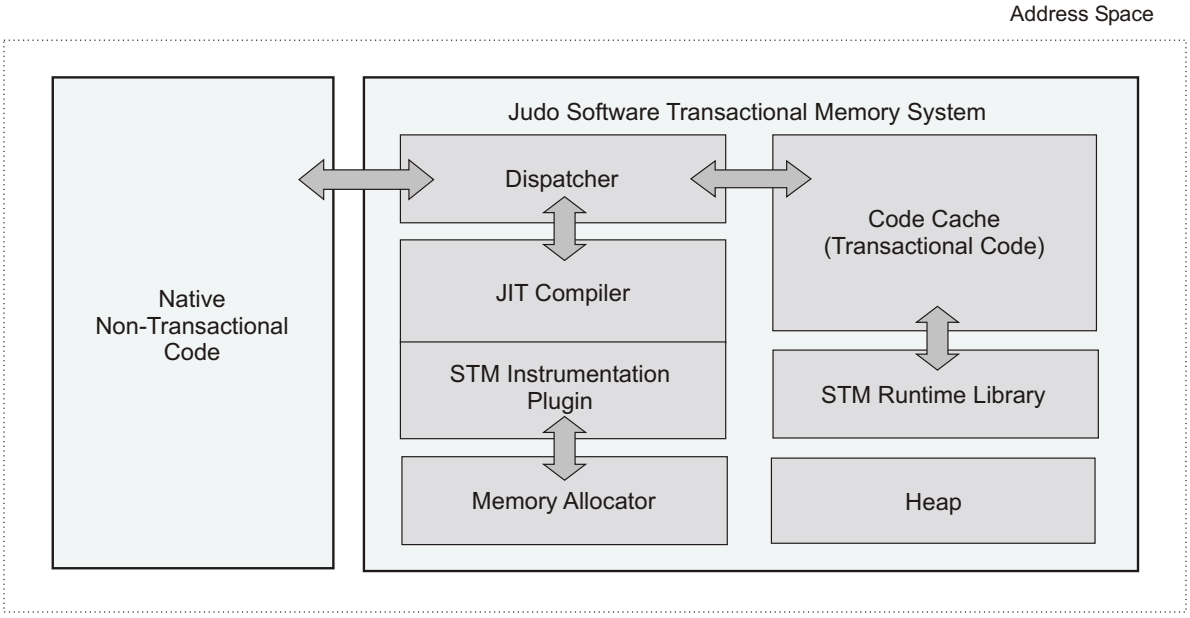


Figure 4.1: JudoSTM software architecture.

4.3.2 Defining a Transaction

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 4.2. 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_start` and before a call to `judostm_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

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

Figure 4.2: Definition of JudoSTM's atomic macro used to specify transactions.

support, for example by using a comprehensive checkpointing algorithm such as the one presented in [41].

4.3.3 Read and Write-buffering

JudoSTM instruments transaction code to execute through both a *read-buffer* and *write-buffer*. The read-buffer is used to store the initial values read by the transaction during its execution so that they can be verified as unchanged during commit, while the write-buffer isolates the transaction's modifications from others until validated. While a transaction is executing, all accesses to previously-written or read locations must be redirected into one of the two buffers. For locations that have been written, a quick lookup in the write-buffer ensures that a subsequent read operates on correct data. The need for the same level of indirection for locations that have been previously read seems unnecessary at first, but it is required to prevent a transaction from reading a value that is not used to validate the read-set. The problem is illustrated by the following example.

Consider the situation presented in Figure 4.3 where a transaction (Transaction 1) is executing concurrently with a second privileged transaction that has made a system call (Transaction 2)—since Transaction 2 is privileged its writes are not buffered. Assume that both transactions are operating on the same global variable `x`, which is initially set to `0`. First Transaction 1 reads `x` and copies it to the read-buffer for eventual read-set validation. Next Transaction 2 writes 100 into `x`, and Transaction 1 reads it again

Assume that initially `x==0`

```
// Transaction 1          // Transaction 2 (privileged)
atomic {                  ...
  t1=x;                   ...
  ...                     x=100;
  t2=x;                   ...
  ...                     x=0;
}                         ...
```

Figure 4.3: Read-buffering motivational example. Unless Transaction 1 performs read-buffering, the conflict created by Transaction 2 may go undetected.

directly from memory. If Transaction 2 modifies `x` back to its original value before Transaction 1 performs read-set validation, Transaction 1 will fail to detect the conflict. To solve this problem, the second read is instead redirected to the read-buffer—this way Transaction 1 will still commit but will have executed with a consistent view of `x` (i.e., the original value of `x` before Transaction 2 modified it). Therefore JudoSTM must introduce this level of indirection for all memory accesses that may conflict. While we used a privileged transaction to demonstrate this problem, it can be reproduced with non-privileged transactions as well.

To redirect reads and writes into the buffers, JudoSTM rewrites all instructions that access global memory so that they perform a lookup in both buffers before executing. JudoSTM identifies such instructions using a simple heuristic which 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 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 [42], which we plan to investigate in future work.

To quickly lookup an address in a buffer, we use a linear-probed open-address hash

```
// Original instruction
add    (%eax,%ebx,4),$1
```

```
// Instrumented instruction
lea    (%eax,%ebx,4), %edx
movzx  %dx, %ecx
cmp    0x119A20A0(,%ecx,8), %edx
jnz    probe_more
mov    0x119A20A4(,%ecx,8), %edx
add    (%edx),$1
```

Figure 4.4: Example of read/write-buffer lookup instrumentation.

table. JudoSTM inlines the first portion of the lookup directly into the transaction code so that a hit in the hash table requires as little as five extra instructions. Figure 4.4 presents the instrumentation code, in assembly, for an `add` instruction that operates on an effective address specified by the `%eax` and `%ebx` registers. The instrumentation starts by computing this effective address using the load effective address instruction (`lea`). Next, the move short and zero-extend instruction performs a mod 64K[1] to compute an initial lookup index, which is used by a compare instruction to check the address against what is in the table. If the addresses match, a move instruction loads the buffered address into a register, which is subsequently used to specify the effective address for the original add instruction. If the comparison fails, control is redirected to another basic block which calls the STM runtime library to probe the hash table some more, and save the value in the buffer if the probing fails.

To minimize the number of registers required to perform the lookup, JudoSTM specifies the effective address for the compare and move instructions with x86's scaled index addressing mode, using a static per-transaction hash table base address, and an index scaled by 8 bytes (each entry in the table stores a 4 byte address and the corresponding 4 byte buffered address). Unfortunately, since this addressing mode supports a maximum scaling of 8 bytes, this approach prevents JudoSTM from storing additional information

[1]For transactions with working sets that are greater than 64K, JudoSTM creates a hash function out of a `mov` and an `and` instruction.

in the table. Most importantly, JudoSTM must distinguish between memory locations that have been read from ones that have been written, so that it can determine whether the location should be read or write-buffered, or both. As a solution, JudoSTM maintains two hash tables: a read and a write table, which are used separately by read-only instructions and ones that write. JudoSTM subsequently uses the following algorithm: If a read instruction fails to find its address in the read hash table, the address is saved in the read-buffer, and the read hash table is updated to hold an entry pointing to the read-buffer. If a write instruction fails to to find its address in the write hash table, the address is saved in the write-buffer and both read and write hash tables are updated to point to the write-buffer.

If the `%ecx`, `%edx`, or eflags (condition flags) registers are live after the instrumented instruction, Judo additionally emits instructions to save and restore the registers to per-transaction private memory. Rather than using the `pushf` and `popf` instructions for saving and restoring the eflags register, which stalls the processor pipeline, we follow the approach taken in [7] and emit `lahf`, `seto/sahf`, `add` instructions which can save and restore only the arithmetic flags without requiring a pipeline flush.

For instructions that operate on a sub word granularity, JudoSTM emits instrumentation that first truncates the target address to one that is 4 byte aligned, performs the lookup, and finally adds back the truncated bytes. This ensures that overlapping accesses of different granularities to the same word are redirected to the same word in the write/read-buffer.

Finally, 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 [41], and can make regular stack instructions—that use frame or stack

pointers—access stale 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. Finally, we currently do not support sharing of stack local data between threads; however, such sharing can be detected using a page protection mechanism [42].

4.3.4 Sandboxing

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.

Since JudoSTM does not insert read-set validations after each memory read, transactional threads that have read inconsistent data can sometimes enter infinite loops. To handle such unfortunate cases, 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[2].

4.3.5 Commit

JudoSTM implements two different commit modes: coarse and fine-grained. In coarse-grained commit, a single global lock is used to synchronize transaction validation and

[2]We found that 256 worked well in practice as it did not require an extra compare instruction. A full study in search of the optimal value is beyond the scope of this dissertation

Assume that initially `x==0` and `y==1` and that `x + y` must always equal `1`

```
// Transaction 1    // Transaction 2    // Transaction 3
...                 ...                 atomic {
...                 ...                   t1=x;
atomic {            ...                   ...
  x=1;              ...                   ...
  y=0;              ...                   ...
}                   ...                   ...
...                 ...                   t2=y;
...                 atomic {              ...
...                   x=0;                ...
...                   y=1;                ...
...                 }                   }
```

Figure 4.5: Potential for incorrect transaction commit. Transaction 3 may fail to detect the conflict if it validates its read-set while Transaction 2 is committing its write-buffer.

commit to enforce that transactions atomically either succeed or abort. When a transaction is ready to commit, it must acquire the commit lock, verify its read-buffer, and if successful, commit its write-buffer before finally releasing the lock. While read-only transactions do not need to acquire the lock to commit, they must ensure that they do not validate their read-buffers against memory that is in an inconsistent or intermediate state, which can occur while another transaction is busy committing its write-buffer. Figure 4.5 presents an example where such a case can lead to an incorrect validation. Assume that `x` and `y` are initially set to `0` and `1`, respectively, and that the following invariant must always be true: `x + y == 1`. Transaction 3 starts by reading the value of `x` (`0`), saving it into `t1`. Next, Transaction 1 swaps the values of `x` and `y` such that they now equal `1` and `0`, respectively. Transaction 3 reads the value of `y` (`0`) but does not notice the inconsistent state of memory since it is executing using invisible reads, without performing any incremental validation. Transaction 2 starts executing, swapping the values of `x` and `y` again, and begins to commit the changes. If Transaction 3 validates its read-buffer while Transaction 2 is in the middle of committing its write-buffer, after

the store to `x` but before the store to `y`, Transaction 3 will compare its read-buffer (of two zeros) to the two zeros temporarily present in memory, and thus incorrectly validate itself. Forcing Transaction 3 to acquire the commit lock can prevent this problem at the cost of increased contention. Alternatively, to reduce contention, it is sufficient for Transaction 3 to repeatedly validate its read-buffer stopping only once it is certain that the lock was not held by anyone during the coarse of its validation. JudoSTM achieves this by versioning the commit lock, storing a commit version number in the upper 31 bits of the lock word. When a writing transaction commits its write-buffer, it releases the lock by incrementing the upper 31 bits and setting the lower bit to `0`. Thus, read-only transactions need only check that the lock is not acquired before proceeding with validation, and that it did not change after validation. Additional, load 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 strips and associates a lock with each of them. To commit, a transaction acquires the locks associated with the strips that it will write to. Similarly, a transaction verifies that the locks associated with the strip being validated are not acquired by other transactions during read-set validation. To prevent deadlock, we follow the common approach of relying on bounded spin-wait times, rather than sorting the locks to maintain an order to the acquisitions. Our fine-grained commit algorithm is most similar to that of the one described by Harris *et al.* in their STM implementation for Haskell [18]; however, rather than associating a lock with each variable, we associate each lock with a strip. Additionally, we verify that the locks have not be acquired during a read-set validation by comparing the sum of the lock version numbers before and after a validation rather than individually comparing each of the locks.

Finally, in the event that a validation fails, JudoSTM resolves the conflict differently

depending on which commit variant is used. For coarse-grained commit, JudoSTM reruns 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 fine-grained commit, JudoSTM must acquire all locks to execute a transaction natively. Given the number of locks, this is a costly endeavor and is therefore only performed after 1000 failed attempts[3].

4.3.6 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.

4.3.7 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

[3]We found that a large number like this performed well for the micro-benchmarks we tested; however, further experimentation is necessary to find a robust and efficient contention heuristic that works for a greater set of benchmarks

```
...
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 4.6: Example of emitted transaction-instance-specific read-set validation code.

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 4.6) 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, which instructs the processor to fetch and speculatively execute the fall-through basic block. 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 and are emitted each time a new read address is discovered. We call the result an *executable read-buffer*.

```
...
movl $0x0, 0x80B10CA4
movl $0x80B10CFC, 0x80B10BCC
movl $0x80B10CA4, 0x80B10BB8
movl $0x54, 0x80B10AB0
ret
```

Figure 4.7: Example of emitted transaction-instance-specific commit code.

We use a similar approach for commit code for which we emit a straight-line sequence of `mov` instructions (as in Figure 4.7) 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, resulting in an *executable write-buffer*.

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 expected to be quickly amortized for large read and write-sets. Furthermore, JudoSTM prefetches these instructions while waiting for the commit lock by executing the validation code, and by executing an incorrectly hinted branch instruction that targets the start of the commit code but is never taken.

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.

4.3.8 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, since 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 [23], garbage collectors, or quiescing [12]. 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"[4] 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 a 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 dissertation we use the Hoard highly scalable parallel memory allocator [4]. Since JudoSTM eliminates all `lock` prefixes in the re-written code, the JITed version of Hoard becomes a highly-efficient optimistic transactional memory allocator.

4.4 Summary

In this chapter we presented the design of JudoSTM, our dynamic binary rewriting software transactional memory system. JudoSTM instruments transactions with STM

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

instrumentation at runtime, while they execute, and thus supports transactions with calls to static and dynamic pre-compiled libraries. Furthermore, to the best of our knowledge JudoSTM is the first *software* TM that supports system calls within transactions. As a result, JudoSTM lets developers program transactional memory applications without imposing any restrictions on what can be placed within a transaction.

Chapter 5

Evaluation

In this chapter we provide a preliminary comparison of the performance of JudoSTM to both a conventional lock-based execution and also to the RSTM 2.0 system [28] on a set of micro-benchmarks. The purpose of this evaluation is to show that JudoSTM can be competitive with an existing library-based STM.

5.1 STM Feature Comparison

Table 5.1 outlines the most significant difference between the RSTM and JudoSTM systems. RSTM operates on an object-granularity requiring the use of object-oriented `C++` code. Additionally, since RSTM offers no compiler support, programmers must manually rewrite their code to include calls to the STM runtime library. JudoSTM, on the other hand, operates at a word-level granularity and can automatically instrument arbitrary `C` and `C++` code specified within `atomic` blocks. RSTM provides obstruction-free synchronization while JudoSTM must be blocking due to its support for system calls. Unfortunately, RSTM's use of obstruction-freedom affects the conclusions that can be drawn from this study since their is an additional cost to supporting non-blocking syn-

	RSTM	JudoSTM
Language	C++	C/C++
Programming Model	Library API, rewrite code	atomic {...}
Conflict Detection	Object-level location-based	Word-level value-based
Synchronization	Obstruction-free	Blocking
Commit	Object-cloning & pointer-switching	Executable write-buffer
Memory Allocation	Custom	"Hoard" scalable parallel allocator

Table 5.1: Comparison of RSTM and JudoSTM features.

chronization; however, at the time of this evaluation, we were unable to obtain a suitable copy of a blocking STM that could execute on x86. Finally, RSTM's lack of system call support requires that it use a custom memory allocator, while JudoSTM allows the use of any allocator, regardless of whether it performs system calls. For this study, we chose the Hoard scalable memory allocator.

5.2 Experimental Framework

We measure the performance of 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 10 test runs. In all experiments involving RSTM, the default settings were used: `Polka` contention manager, eager acquire, invisible-readers, and the epoch-based RSTM memory allocator. 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 updated. 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.

While micro-benchmarks offer only a partial picture of the STM's performance, they

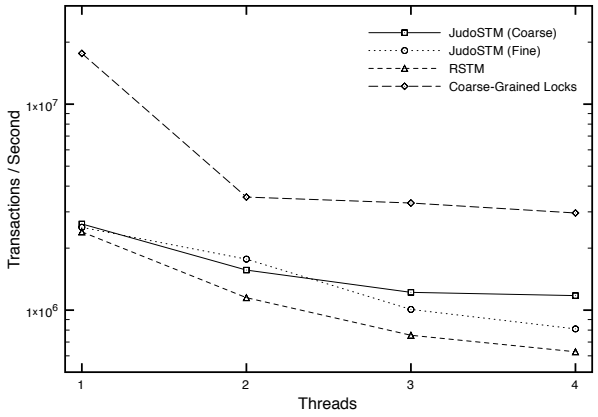
are a good starting point for validating the feasibility of this work. Additionally, since transactional memory is still in its infancy, few representative benchmarks exist. A comprehensive evaluation of the JudoSTM system is left as work for the future, when TM benchmark suites become readily available.

5.3 Performance

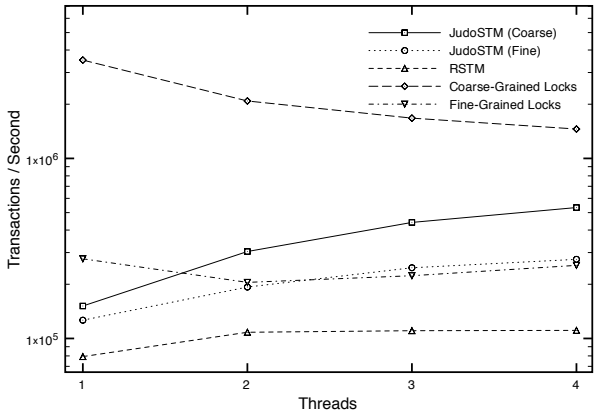
Figure 5.1 and Table 5.2 present the throughput results for each of the four micro-benchmarks. Note that the y-axes are in a log-scale. In every scenario JudoSTM performs competitively with RSTM.

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. Little can be done to improve performance as this micro-benchmark produces the worst-case access patterns for optimistic concurrency.

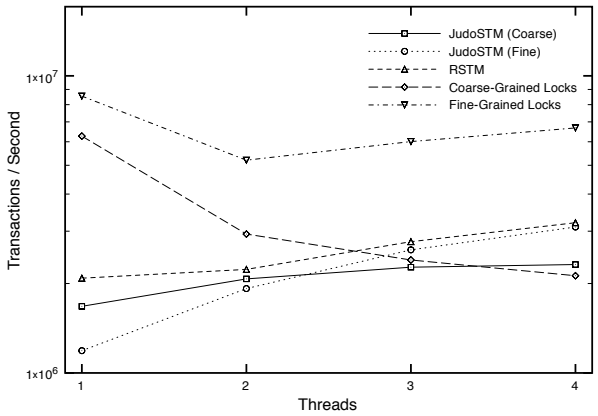
LinkedList: In LinkedList, the coarse-grained lock outperforms all other synchronization methods by a fair margin, though its performance degrades with the addition of threads. JudoSTM's coarse grained implementation, on the other hand, scales close to linearly, obtaining a throughput of 3.5x its single threaded performance at four threads. LinkedList exhibits a relatively large number of conflicts due to the constant number of node insertions and deletes which interfere with other transactions searching for their nodes. As a result, JudoSTM benefits from the coarse-grained contention heuristic which causes the transaction to execute natively after a rollback, thus reducing synchronization overhead and guaranteeing progress while still allowing other transactions to execute con-



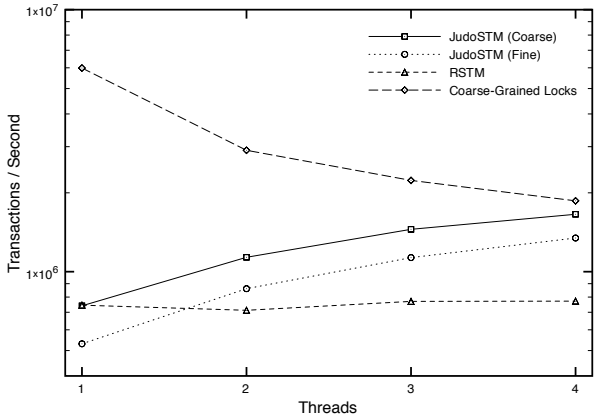
(a) Counter



(b) LinkedList



(c) HashTable



(d) RBTree

Figure 5.1: Benchmark performance comparisons (Note the log scale).

currently. JudoSTM's fine-grained commit implementation performs less well, achieving a throughput improvement of 2.2x over its single threaded performance for four threads. In this case, the cost of enabling parallel commits does not pay off since the benchmark seldom sees two concurrent threads reading and writing to disjoint data. Regardless, it is still able to outperform RSTM at all thread counts, as well as fine-grained locking at three and four threads.

It is interesting to note that unlike in other benchmarks, RSTM achieves a lower

	Threads	RSTM	Coarse-Grained Locking	Fine-Grained Locking	Judo (Fine)	Judo (Course)
Counter	1	2398757	17635467	N/A	2524132	2620580
	2	1149535	3539219	N/A	1768498	1565075
	3	755196	3311706	N/A	1007939	1219980
	4	627860	2962575	N/A	810889	1176341
LinkedList	1	79307	3517271	276963	126326	151462
	2	108247	2084858	204884	192982	304033
	3	110438	1674203	223107	246973	441079
	4	110869	1456085	254688	275080	533479
HashTable	1	2084262	6269103	8546620	1188223	1675981
	2	2231088	2933995	5201553	1923082	2072826
	3	2765749	2399992	6008633	2596054	2269789
	4	3203580	2124495	6679233	3099031	2317034
RBTree	1	745066	5988545	N/A	530518	740911
	2	712254	2906314	N/A	861225	1134857
	3	770287	2230442	N/A	1131581	1451182
	4	771991	1863915	N/A	1344401	1656253

Table 5.2: Benchmark performance comparisons. Results presented in number of transactions completed per second.

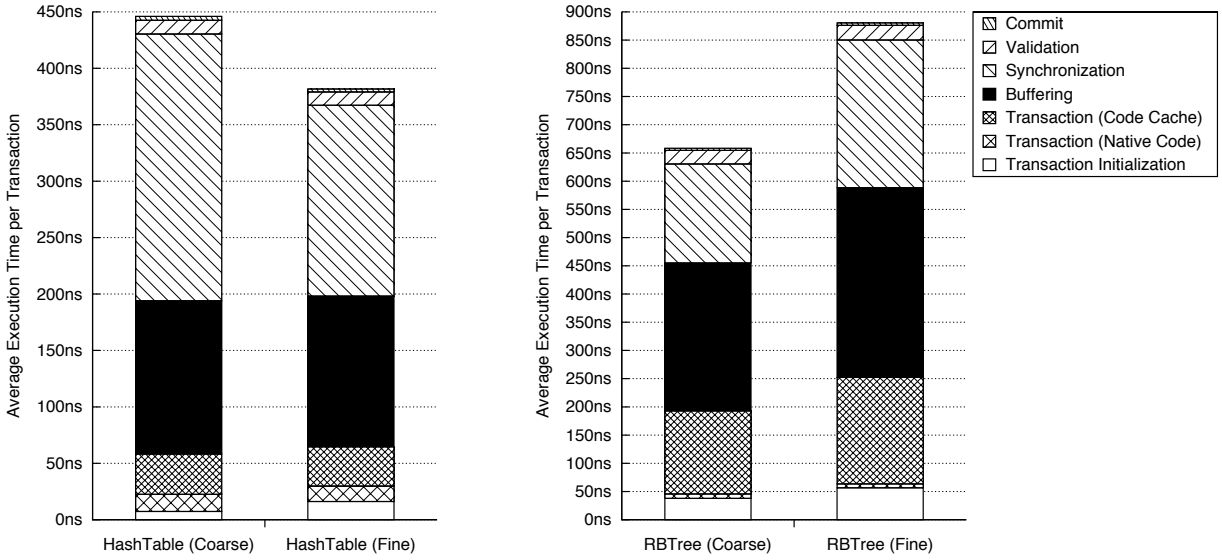
single threaded performance than both versions of JudoSTM for LinkedList. This is despite the fact that the JudoSTM transactions are instrumented at runtime, with more expensive word-granularity instrumentation. The lower performance can be attributed to the large amount of read operations performed by the transactions ($O(n)$), which are required to find a desired node in the linked list. Since RSTM verifies its read-set after each read, its invisible reader strategy causes high overhead for this benchmark. JudoSTM, on the other hand, does not need to incrementally validate its read-set, and can therefore obtain better single threaded performance.

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 JudoSTM's fine-grained commit, which performs up to 34% better than coarse-grained commit. At four threads, JudoSTM's fine-grained commit achieves a speedup of 2.6x over its single threaded performance, beating coarse-grained locking by 46% while performing similarly to RSTM despite having significantly more single threaded overhead. Fine-grained locking outperforms all other synchronization techniques, with an initial drop in performance at two threads followed by a steady increase.

RBTree: In RBTree, both versions of JudoSTM scale fairly well, with four processor speedups of 2.2x and 2.5x. Coarse-grained commit on the other hand, starts strong on a single thread but exhibits a rapid degradation in performance as threads are added. At four threads, the coarse-grained version of JudoSTM is within 13% of the performance of coarse-grained locking and appears on track to pass coarse-grained locking if more processors were added. In contrast RSTM fails to scale with more threads added, performing over 2x slower than the coarse-grained version of JudoSTM at four threads. This is despite starting with virtually the same performance at one thread.

5.4 Examining Execution

To obtain insight into the various overheads incurred by the JudoSTM instrumentation, we used the `oprofile` sampling-based system-wide profiling tool to determine the average execution breakdown of a transaction. The results were used to discover and identify potential performance bottlenecks which lead to a number of improvements. To provide `oprofile` with the symbols it needs for our dynamically generated code, we statically defined large functions in assembly and used the space to store dynamically emitted code.



(a) HashTable (b) RBTree

Figure 5.2: Execution breakdown on 4 processors.

Figure 5.2 shows the results of our profiling for both the fine and coarse-grained commit variations of JudoSTM when running either HASHTABLE or RBTREE. In all cases, JudoSTM spends the majority of its time buffering reads and writes, and synchronizing commits with other transactions. As a result of 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*). Due to 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.

While it is unlikely that the synchronization overheads can be reduced further, the execution breakdown results show that substantial performance improvements can be made by reducing the buffering overhead. We expect that a significant amount of this overhead can be attributed to cost of maintaining two hash tables (both read and write hash tables) when inserting new entries into the write-buffer. We plan to examine alternatives in future work.

5.5 Summary

In this chapter we presented a preliminary evaluation of the JudoSTM system. We described the significant differences between JudoSTM and RSTM, after which we compared the performance of the two systems on a number of micro-benchmarks and demonstrated that JudoSTM can perform comparably. Additionally, we examined the execution breakdown of JudoSTM where we discovered that the majority of JudoSTM's overheads can be attributed to buffering and synchronization.

Chapter 6

Conclusions and Future Work

As transactional memory systems move closer to mainstream use, we must make them easier to integrate into typical programming environments. Hence, it is important that software transactional memory systems (STMs) support arbitrary `C` and `C++` code, as well as library functions that may themselves contain system calls and locking code. We have presented JudoSTM, an 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; is programmed using only a simple `atomic{}` macro; and allows the programmer to use the compiler of his choice. JudoSTM supports several desirable features including sandboxing, efficient invisible-readers, 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 a compelling alternative.

6.1 Contributions

This dissertation makes the following contributions:

1. Presents a novel STM system based on dynamic binary rewriting that supports both statically and dynamically linked arbitrary C and C++ code;

2. Demonstrates the feasibility of such an approach by comparing a prototype to the Rochester Software Transactional Memory (RSTM) system;

3. Proposes 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;

4. Introduces 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; and

5. Presents the Judo and JudoSTM frameworks for use in future research.

6.2 Future Work

There are many avenues that may be pursued for building on top of this work. Optimizing JudoSTM's performance and evaluating it on real benchmarks are evident choices; however, the following two extensions are also of particular interest.

6.2.1 Support for Strong Atomicity

JudoSTM does not currently provide *strong atomicity*, and therefore programmers must be cautious not to write or call functions that operate on shared memory, concurrently

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, non-transaction conflicts introduced after a transaction validates its read-set but, before or during write-set commit, will go unnoticed. Furthermore, since JudoSTM cannot commit its buffered changes instantaneously, non-transactional threads may observe inconsistent state. Finally, while JudoSTM does correctly support privatization when using a single commit lock, JudoSTM's distributed lock version cannot correctly protect non-transactional accesses to privatized memory from accesses within concurrent transactions.

Extending JudoSTM to provide a stronger level of safety is interesting work for the future. Strong atomicity is especially compelling in the presence of code containing legacy locks as it allows legacy lock-protected code to execute safely in parallel with the same code rewritten for optimistic execution within a transaction. Unlike most STMs, JudoSTM can be extended to provide strong atomicity without requiring that non-transactional threads be rewritten with STM instrumentation. Instead, each non-transactional thread can quiesce while a transaction commits so that the transaction's changes appear to occur instantly. Quiescing also prevents conflicts caused by non-atomic code from occurring after a transaction validates its read-set, thus enabling transactions to safely detect conflicts caused by arbitrary non-transactional code.

6.2.2 Application to Hybrid Transactional Memory Systems

JudoSTM is particularly well suited for use in a hybrid transactional memory system composed of a best-effort HTM and a backup unbounded STM that re-executes failed transactions which run out of resources on the HTM. JudoSTM's ability to execute arbitrary machine code, regardless of how it was compiled or linked, matches that of an HTM.

Furthermore, since JudoSTM uses value-based conflict detection, it can detect conflicts caused by hardware transactions without requiring explicit communication between the HTM and STM. Such communication incurs overhead on transactions executing in hardware, thus limiting performance. As a result, JudoSTM can be easily used to augment a simple best-effort HTM to support large transactions and system calls without affecting the performance of the small common-case transactions that execute on the HTM. We plan to investigate such an extension in the future.

Bibliography

[1] Bowen Alpern, Dick Attanasio, John Barton, Michael Burke, Perry Cheng, Jong-Deok Choi, Anthony Cocchi, Stephen Fink, David Grove, Michael Hind, Susan Flynn Hummel, Derek Lieber, Vassily Litvinov, Ton Ngo, Mark Mergen, Vivek Sarkar, Mauricio Serrano, Janice Shepherd, Stephen Smith, VC Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeno virtual machine. 39(1), 2000.

[2] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture*, pages 316–327. Feb 2005.

[3] Fabrice Bellard. QEMU: a fast and portable dynamic translator. In *Proc. of USENIX 2005 Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[4] Emery Berger, Kathryn McKinley, Robert Blumofe, and Paul Wilson. Hoard: A scalable memory allocator for multithreaded applications. Technical report.

[5] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2), Nov 2006.

[6] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Unrestricted trans-

actional 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.

[7] Derek Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. Ph.d. thesis, Massachusetts Institute of Technology, Cambridge, MA, September 2004.

[8] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, Washington, DC, USA, 2003.

[9] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. Jun 2007.

[10] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. Jun 2007.

[11] O. Shalev D. Dice and N. Shavit. Transactional locking ii. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, 2006.

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

[13] Chen Ding, Xipeng Shen, Kirk Kelsey, Chris Tice, Ruke Huang, and Chengliang Zhang. Software behavior oriented parallelization. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*.

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

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

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

[17] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *ISCA '04: Proceedings of the 31st annual International Symposium on Computer Architecture*, pages 102–113, June 2004.

[18] Tim Harris, Simon Marlow, and Simon Peyton Jones. Haskell on a shared-memory multiprocessor. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 49–61, New York, NY, USA, 2005. ACM Press.

[19] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM Press.

[20] Maurice Herlihy, Victor Luchangco, Mark Moir, and III William N. Scherer. Software transactional memory for dynamic-sized data structures. pages 92–101, Jul 2003.

[21] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM Press.

[22] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[23] Richard L. Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin C. Hertzberg. Mcrt-malloc: a scalable transactional memory allocator. In *Proceedings of the International Symposium on Memory management*, New York, NY, USA, 2006.

[24] 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.

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

[26] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geof f Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim 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.

[27] Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Adaptive software transactional memory. In *Proceedings of the International Symposium on Distributed Computing*, Cracow, Poland, Sep 2005.

[28] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the overhead of software transactional memory. Technical Report TR 893, Computer Science Department, University of Rochester, Mar 2006.

[29] Austen McDonald, JaeWoong Chung, Brian D. Carlstrom, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural semantics for practical transactional memory. *SIGARCH Comput. Archit. News*, 34(2):53–65, 2006.

[30] Austen McDonald, JaeWoong Chung, Hassan Chafi, Chi Cao Minh, Brian D. Carlstrom, Lance Hammond, Christos Kozyrakis, and Kunle Olukotun. Characterization of tcc on chip-multiprocessors. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*. Sept 2005.

[31] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. Logtm: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265. Feb 2006.

[32] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2):1–23, October 2003.

[33] Marek Olszewski, Keir Mierle, Adam Czajkowski, and Angela Demke Brown. Jit instrumentation—a novel approach to dynamically instrument operating systems. In *EuroSys 2007*, Mar 2007.

[34] Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpot^{TM} server compiler. In *USENIX Java Virtual Machine Research and Technology Symposium*, pages 1–12, April 2001.

[35] Christopher J. F. Pickett, Clark Verbrugge, and Allan Kielstra. libspmt: A library for speculative multithreading. Technical Report SABLE-TR-2007-1, Sable Research Group, School of Computer Science, McGill University, Montréal, Québec, Canada, 2007.

[36] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 494–505. IEEE Computer Society, Jun 2005.

[37] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin 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.

[38] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in stm. *SIGPLAN Not.*, 42(6):78–88, 2007.

[39] Arrvindh Shriraman, Virendra Marathe, Sandhya Dwarkadas, Michael L. Scott, David Eisenstat, Christopher Heriot, William N, Scherer III, and Michael F. Spear. Hardware acceleration of software transactional memory. Technical Report 887, Department of Computer Science, University of Rochester, Dec 2005.

[40] Swaroop Sridhar, Jonathan S. Shapiro, Eric Northup, and Prashanth 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.

[41] Cheng Wang, Wei-Yu Chen, Youfeng Wu, Bratin Saha, and Ali-Reza 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.

[42] Victor Ying, Cheng Wang, and Youfeng 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.