Transactional Execution of Java Programs

Brian D. Carlstrom, JaeWoong Chung, Hassan Chafi, Austen McDonald, Chi Cao Minh
Lance Hammond, Christos Kozyrakis, and Kunle Olukotun

Computer Systems Laboratory
Stanford University
{*bdc, jwchung, hchafi, austenmc, caominh, lance, kozyraki, kunle*}*@stanford.edu*

ABSTRACT

Parallel programming is difficult due to the complexity of dealing with conventional lock-based synchronization. To simplify parallel programming, there have been a number of proposals to support transactions directly in hardware and eliminate locks completely. Although hardware support for transactions has the potential to completely change the way parallel programs are written, initially transactions will be used to execute existing parallel programs. In this paper we investigate the implications of using transactions to execute existing parallel Java programs. Our results show that transactions can be used to support all aspects of Java multithreaded programs. Moreover, the conversion of a lock-based application into transactions is largely straightforward. The performance that these converted applications achieve is equal to or sometimes better than the original lock-based implementation.

Categories and Subject Descriptors

C.5.0 [**Computer Systems Implementation**]: General; D.1.3 [**Programming Techniques**]: Concurrent Programming – parallel programming; D.3.3 [**Programming Languages**]: Language Constructs and Features – concurrent programming structures

General Terms

Performance, Design, Languages

Keywords

Transactions, feedback optimization, multiprocessor architecture

1. INTRODUCTION

Processor vendors have exhausted their ability to improve single-thread performance using techniques such as simply increasing clock frequency [41, 2]. Hence they are turning *en masse* to single-chip multiprocessors (CMPs) as a realistic path towards scalable performance for server, embedded, and even desktop platforms [22, 20, 34, 5]. While parallelizing server applications is straightforward with CMPs, parallelizing existing desktop applications is much harder.

Traditional multithreaded programming focuses on using locks for mutual exclusion. By convention, access to shared data is coordinated through ownership of one or more locks. Typically a programmer will use one lock per data structure to keep the locking protocol simple. Unfortunately, such coarse-grained locking often leads to serialization on high-contention data structures. On the other hand, finer-grained locking can improve concurrency, but only by making code much more complex. With such code, it is often easy to end up in deadlock situations.

Transactional memory has been proposed as an abstraction to simplify parallel programming [16, 36, 17, 14]. Transactions eliminate locking completely by grouping sequences of object references into atomic execution units. They provide an easy-to-use technique for non-blocking synchronization over multiple objects, because the programmers can focus on determining where atomicity is necessary, and not the implementation details of synchronization. Although transactional memory has the potential to change the way parallel programs are written, initially transactions will be used to execute existing parallel programs. To understand issues associated with executing existing parallel programs using transactional memory, we investigate how Java parallel programs interact with hardware-supported transactional memory that provides continuous transactions. There have been several recent proposals that can support continuous transactions such as Transactional Coherence and Consistency (TCC), Unbounded Transactional Memory (UTM), and Virtual Transactional Memory (VTM) [13, 4, 33].

Our general approach of executing Java parallel programs with hardware transactional memory is to turn `synchronized` blocks into atomic transactions. Transactions provide strong atomicity semantics for all referenced objects, providing a natural replacement for critical sections defined by `synchronized`. The programmer does not have to identify shared objects *a priori* and follow disciplined locking and nesting conventions for correct synchronization. Studies show that this is what programmers usually mean by `synchronized` in Java applications [9]. Optimistic execution of transactions provides good parallel performance in the common case of non-conflicting object accesses, without the need for fine-grain locking mechanisms that further complicate correctness and introduce significant overhead.

The paper makes the following contributions:

- We show how hardware transactional memory can support all aspects of Java multithreaded programming, including synchronized blocks, native methods, synchronization on condition variables, and I/O statements. We show how existing Java applications can be run with few, if any, changes to the program.
- We demonstrate the results of running parallel Java programs, showing how our transactional execution matches or exceeds the performance of the original lock-based versions.

The rest of the paper is organized as follows. Section 2 provides an architectural overview of continuous transactional execution models. Section 3 describes running Java applications with transactions. In Section 4, we evaluate the performance of transactional Java programs. Section 5 discusses related work, and we conclude in Section 6.

2. CONTINUOUS TRANSACTIONAL ARCHITECTURES

The particular variant of transactional memory we consider in this paper is continuous transactions. Continuous transactional architectures provide hardware support for transactional memory in a shared-memory multiprocessor. Each thread consists of a sequence of transactions, where each transaction is a sequence of instructions guaranteed to execute and complete only as an atomic unit. As opposed to earlier hardware transactional memory proposals, there is no execution outside of transactions. The processors execute transactions, enforce atomicity, and maintain memory consistency and cache coherence only at transaction boundaries, when object updates are propagated to shared memory. By using the continuous transactional model, we hope to uncover a wider range of issues than would be found in a system that only uses transactions a fraction of the time.

Each processor follows a cycle of transaction execution, commit, and potential re-execution. Each transaction is executed speculatively, assuming it operates on data that is disjoint from transactions running concurrently on other processors. Writes are buffered locally, and do not update shared memory before the transaction commits. Transactions are considered indivisible if they are supporting a programmer defined notion of atomicity or divisible if they can be split as needed by the hardware. For example, if an indivisible transaction exceeds its processor's cache capacity, its write-set overflow can be managed by software buffering [16, 14] or lower levels of the memory hierarchy [10, 4, 33] whereas a divisible transaction can be split as needed by a commit.

When a transaction commits, it communicates its write-set to the rest of the system. Other processors determine when they have speculatively read data that has been modified by another transaction. These data dependency violations trigger re-execution to maintain atomicity, usually of the other processor's current transaction, but potentially of the committing processor. Hardware guarantees that transaction commits are seen by all processors in the same order, so commits are always globally serialized. Some systems allow programmer specified ordering of transaction commits to support such features as loop speculation and speculating through barriers. Systems that only support unordered transactions sometimes require transactions to commit from oldest to youngest to ensure forward progress which could lead to load balancing issues. Some systems simply favor older transactions when a data dependency violation is detected. Other systems simply allow transactions to commit on a first come, first served basis, addressing forward progress through other mechanisms.

Transactional buffering and commit provide atomicity of execution across threads at hardware speeds. A multithreaded language like Java can build on these mechanisms to provide non-blocking synchronization and mutual exclusion capabilities, as we explain in Section 3.

3. RUNNING JAVA WITH TRANSACTIONS

In this section we explain how the concurrency features of the Java programming language are mapped to a transactional execution model in order to run existing Java programs with a continuous transactional execution model. In the course of our explanation, we discuss how transactions interact with the Java Memory Model, the Java Native Interface, non-transactional operations, and exceptions.

3.1 Mapping Java to Transactions

In this section we discuss how various existing Java constructs are mapped into transactional concepts.

synchronized blocks

When running with a transactional memory model, `synchronized` blocks are used to mark indivisible transactions within a thread. Since continuous transactional hardware runs code in transactions at all times, a `synchronized` block defines three transaction regions: the divisible transactions before the block, the indivisible transaction within the block, and the divisible transactions after the block. Similarly, accesses to `volatile` fields are considered to be small indivisible transactions. As an example, consider the simple program shown in Figure 1. This program creates an indivisible transaction separated by other divisible transactions by commit points as shown.

When `synchronized` blocks are nested, either within a method or across method calls, only the outermost `synchronized` block defines an indivisible transaction. This is referred to as a closed nesting model [28]. As an example, consider the simple program shown in Figure 2. This program also creates a single indivisible transaction as shown.

Handling nested transactions is important for composability. It allows the atomicity needs of a caller and callee to be handled correctly without either method being aware of the other's implementation. The block structured style of `synchronized` is fundamental to this, as it ensures that transaction begins and ends are properly balanced and nested. Simply exposing a commit method to programmers would allow a library routine to commit arbitrarily, potentially breaking the atomicity requirements of a caller.

While nested transactions are necessary for composability, the run-time flattening into a single transaction simplifies execution, making the common case fast. The database community has explored alternatives, including nested transactions allowing partial rollback. Evaluating the need for and implementing such alternatives is beyond the scope of this paper.

Although `synchronized` blocks are used to mark indivisible transactions, the actual lock object specified is not

```
public static void main(String[] args){          a();       // divisible transactions
    a();                                         COMMIT();
    synchronized(x){                             b();       // indivisible transaction
        b();}                                    COMMIT();
    c();}                                        c();       // divisible transactions
                                                 COMMIT();
```

Figure 1: Converting a `synchronized` block into transactions.

```
public static void main(String[] args){          a();       // divisible transactions
    a();                                         COMMIT();
    synchronized(x){                             b1();      //
        b1();                                    b2();      // indivisible transaction
        synchronized(y){                         b3();      //
            b2();}                               COMMIT();
        b3();}                                   c();       // divisible transactions
    c();}                                        COMMIT();
```

Figure 2: Converting nested `synchronized` blocks into transactions.

used. That is because the continuous transactional hardware will detect *any* true data dependencies at runtime. In a purely transactional variant of Java, one can imagine replacing `synchronized` blocks with a simpler `atomic` syntax omitting the lock variable as is done in other systems [14, 3, 7]. In such a system, programmers would not need to create a mapping between shared data and the lock objects that protect them.

The fact that the lock variables are not used points to a key advantage of transactions over locks. In Java without transactions, there is not a direct link between a lock object and the data it protects. Even well intentioned programs can mistakenly synchronize on the wrong object when accessing data. With transactions, *all* data accesses are protected, guaranteeing atomic semantics in all cases. There is no reason for basic data structures to provide any synchronization, because the caller defines its own atomicity requirements. Hence, programmers can write data structures without resorting to complex fine grained locking schemes to minimize the length of critical sections.

For example, the evolution of Java collections shows how locking can complicate one of the most basic data structure classes: the hash table. Java's original `Hashtable` used `synchronized` to guarantee internal consistency, which is important in a sandbox environment. However, in JDK 1.2, a simpler non-locking `HashMap` was introduced, since most applications needed to avoid the overhead of the implicit locking of the original `Hashtable`. Recently, JDK 1.5 has complicated matters further by adding a `ConcurrentHashMap` that allows multiple concurrent readers and writers.

A transactional memory model eliminates the need for this kind of complexity in the common case. Consider the simple string interning example in Figure 3. With transactional execution, there is no need to use anything other than the original simple `Hashtable`. Concurrent reads happen in parallel due to speculation. Even non-conflicting concurrent writes happen in parallel. Only conflicting and concurrent reads and writes cause serialization and this is handled automatically by the system, not the programmer.

Traditionally, users of `synchronized` blocks are encouraged to make them as short as possible to minimize blocking other threads' access to critical sections. The consequences of making the critical section too large is that processors often spend more time waiting and less time on useful work. At worst, it can lead to complete serialization of work. Consider the example code in Figure 4. Because of the way this code is written with a `synchronized` block around the entire routine, a multithreaded web server becomes effectively single threaded, with all requests pessimistically blocked on the `sessions` lock even though the reads and writes are non-conflicting. Because a transactional system can simply optimistically speculate through the lock, it does not have this serialization problem.

wait, notify, notifyAll

When threads need exclusive access to a resource, they use `synchronized` blocks. When threads need to coordinate their work, they use `wait`, `notify`, and `notifyAll`. Typically, these condition variable methods are used for implementing producer-consumer patterns or barriers.

Consider the example of a simple producer-consumer usage of `wait` and `notifyAll` derived from [6] shown in Figure 5 and in Figure 6. This code works in a non-transactional system as follows. When a consumer tries to `get` the `contents`, it takes the lock on the container, checks for `contents` to be `available`, and calls `wait` if there is none, releasing the lock. After returning from `wait`, the caller has reacquired the lock but has to again check for `contents` to be `available` since another consumer may have taken it. Once the data is marked as taken, the consumer uses `notifyAll` to alert any blocking producers that there is now space to `put` a value. An analogous process happens for producers with `put`.

In a transactional system, the `get` method is `synchronized` so the method is run as an indivisible transaction. If there is no data `available` and we need to `wait`, we commit the transaction as if the `synchronized` block was closed. This is keeping analogous to the semantics of `wait` releasing the lock and making updates visible. When the thread returns from waiting, we start a new indivisible transaction.

Because we commit on `wait`, we also are committing state from any outer `synchronized` blocks, potentially breaking atomicity of nested locks. One alternative considered was using rollback, since that would preserve the atomicity of outer `synchronized` blocks and works for most producer-consumer examples. However, many commonly used pat-

```
String intern(){
    synchronized(map){
        Object o=map.get(this);
        if (o!=null){
            return (String)o;}
        map.put(this,this);
        return this;}}
```

Figure 3: A string interning example.

```
void handleRequest(String id, String command){
    synchronized(sessions){
        Session s=sessions.get(id);
        s.handle(command);
        sessions.put(id,s);}}
```

Figure 4: Synchronizing on a `Session` object.

```
synchronized int get(){
    while (available == false) wait();
    available = false;
    notifyAll();
    return contents;}
```

Figure 5: `get` code used by the consumer.

```
synchronized void put(int value){
    while (available == true) wait();
    contents = value;
    available = true;
    notifyAll();}
```

Figure 6: `put` code used by the producer.

terns for barriers would not work with rollback. Rollback prevents all communication, but the existing Java semantics of `wait` are to release a lock and make any changes visible. This loss of atomicity in outer `synchronized` blocks because of nesting is a common source of problems in existing Java programs as well [35].

Fortunately, the nesting of `wait` in `synchronized` blocks is rare, since it causes problems in existing Java programs as well. Java programmers are advised not to place calls to `wait` within nested `synchronized` blocks because when a thread waits, it retains all but one lock while it is asleep [35]. By their very nature, condition variables are used to coordinate the high-level interactions of threads, so it is rare for them to be used deeply in library routines. For example, a survey of the Apache Jakarta Tomcat web server version 5.5.8 does not reveal any `wait` calls nested within an outer `synchronized` block. Tomcat does have libraries for purposes such as producer-consumer queuing that include uses of `wait` on an object with a corresponding `synchronized` block on the same object, but they are used only for high-level dispatching and not called from `synchronized` blocks on other objects. A further example is in SPECjbb2000, which has one example of a barrier where committing works and rollback would fail.

The handling of `wait` is the thorniest problem in the transactional execution of Java programs. If we treat `wait` as a rollback, we have composable transactional semantics but existing programs will not run. If we treat `wait` as a commit, it is easy to come up with contrived programs that will not match the previous semantics. However we have never seen a benchmark or system that exhibits a problem treating `wait` as commit. In a programming language built with transactions in mind, the rollback semantics would make more sense; all of the problematic examples requiring commit could be rewritten to use rollback semantics.

3.2 Impact of Transactions on Java

In this section we discuss how transactions relate to several aspects of the Java programming language.

Java Memory Model

A new Java memory model was recently adopted to better support various shared memory consistency models [1, 30, 19]. The new model has been summarized in [14] as follows:

if a location is shared between threads, either:

(i). all accesses to it must be controlled by a given mutex, or

(ii). it must be marked as volatile.

These rules, while simplified, are an excellent match for transactional execution. The Java memory model ties communication to `synchronized` and `volatile` code. In our system, these same constructs are used to create indivisible transactions working with shared data.

Run-time violation detection can be used to detect programs that violate these rules, an advantage of continuous transactional execution with “always on” violation detection. For example, a divisible transaction might be violated, indicating that a location is shared between threads without being protected by a `synchronized` block or `volatile` keyword. Alternatively, an indivisible transaction can be violated by divisible transaction. These typically unexpected violations could be logged or additional extensions could allow the program to respond to exceptional cases.

Java Native Interface

The Java Native Interface (JNI) allows Java programs to call methods written in other programming languages, such as C [25]. Software transactional memory systems typically forbid most native methods within transactions [14]. This is because those systems only can control the code compiled by their JIT compiler and not any code within native libraries. While some specific runtime functions can be marked as safe or rewritten to be safe, this places a lot of runtime functionality in the non-transactional category.

This limitation of software transactional memory systems destroys the composability and portability of Java software components. Code within `synchronized` blocks cannot call methods without understanding their implementation. For example, an application using JDBC to access a database needs to know if the driver uses JNI. Even “100% Pure Java” drivers may use `java.*` class libraries that are supported by native code. Therefore, code carefully written to work on one Java virtual machine may not work on another system, since the language and library specifications only consider nativeness to be an implementation detail, and not part of the signature of a method.

On the other hand, hardware transactional memory systems do not suffer from this limitation, since they are source-language neutral. Transactions cover memory accesses both by Java *and* native code. This allows programmers to treat code that traverses both Java and native code together as a logical unit, without implementation-specific restrictions.

Non-transactional operations

Certain operations are inherently non-transactional, such as I/O operations. Many transactional memory systems simply consider it an error to perform a non-transactional operation within a transaction. This certainly would prevent correct execution of existing Java programs.

Continuous transactional architectures might seem to pose an extra challenge because of their "all transactions, all the time" nature. However, divisible transactions can handle non-transactional operations by implicitly committing before the operation so there is no memory state to be violated and cause a rollback. Since these code regions do not guarantee atomicity, it is legal to freely commit their results at any point. There are a variety of approaches to the more difficult case of non-transactional operations in indivisible transactions.

A simple approach is to allow only one indivisible transactions to run non-transactional operations at a time. Other transactions that do not need to run non-transactional operations may continue to execute in parallel, but cannot commit. Once the indivisible transactions that performed the non-transactional operation commit, other transactions may then commit or begin their own non-transactional operations. This approach generally works with current Java programs. In contrast, the other, more advanced approaches presented below require changes in applications, libraries, or the underlying runtime. In this study we have taken this simple approach.

Another approach to non-transactional operations is to require that the operations be moved outside of indivisible transactions. For example, an HTTP server can read a request from the network interface into memory, use transactions as it likes, producing a response buffered to memory, and finally write the response to the network after the transactions have committed. However, manually restructuring code in this way is undesirable. Modular systems are built on composable libraries. We need a general way to address non-transactional operations without disturbing the logical structure of code.

A more programmer-friendly way to implement this structure is with changes to the runtime system. Non-transactional operations within indivisible transactions are recorded at each call but only performed at commit time using transaction callbacks, a common solution in traditional database systems. For example, a logging interface for debugging typically prints messages as they are received. A transactional runtime implementation could buffer the messages in a thread local variable when running in an indivisible transactional after registering a commit callback. Only when the original, indivisible transaction commits is the callback called to write out the buffered messages. This is a relatively simple pattern that can generally be applied to library code. For convenience, a reusable wrapper class can be used for such common cases as buffering stream output. In a transactional runtime, this functionality could be built into the standard `BufferedOutputStream` and `BufferedWriter` classes, allowing most output code to work without modification. While commit handlers are useful for delay output, rollback handlers can be used for some cases of reverting input. However, with the combination of rollback handlers and transactional memory this is problematic because when the transaction rolls back, the registration of the handler itself could be rolled back. We will describe a possible solution to this issue in Section 5.

There are several classes of non-transactional operations. Most non-transactional operations are concerned with I/O. However, some operations, such as asking for the current time, may be deemed safe for use within transactions without being considered "non-transactional" operations. Another common non-transactional operation in our current model is thread creation. Although alternative and more complex models could support rolling back of thread creation, this is of little practical value. As in the above example of nesting barriers within existing transactions, it seems better to consider thread creation a high-level operation that rarely occurs within deep nesting of `synchronized` blocks.

Exceptions

We treat transactions and exceptions as orthogonal mechanisms. Most exceptions in practice are `IOExceptions` or `RuntimeExceptions`. Since we would not be able to guarantee the rollback of non-transactional operations, our exception handling follows the flow of control as expected by today's Java programmers. Figure 7 illustrates an example. If no exceptions are thrown, the code produces the transactions shown in Figure 8. On the other hand, if `b2()` throws an `IOException` and `e2()` throws a `RuntimeException`, the code is equivalent to Figure 9. This is exactly the same control flow as current Java exception handling.

4. PERFORMANCE EVALUATION

In this section we compare the performance of existing Java programs running on a traditional multi-processor using snoopy cache coherence and the same applications converted to run on a continuous transactional multi-processor. We discuss the Java virtual machine and simulator used for evaluation, describe the benchmarks performed, and discuss the results.

4.1 Environment

We evaluated programs using JikesRVM with the following changes. The scheduler was changed to pin threads to processors and not migrate threads between processors. Methods were compiled before the start of the main program execution. The garbage collector was disabled and a one gigabyte heap was used. When running transactionally, `synchronized` blocks and methods were run transactionally and `Object.wait()` was changed to perform a transaction commit. The results focus on benchmark execution time, skipping virtual machine startup.

JikesRVM was run with an execution-driven simulator of a PowerPC CMP system that implements the TCC continuous transaction architecture as well as MESI snoopy cache coherence for evaluating locking [27]. All instructions, except loads and stores, have a CPI of 1.0. The memory system models the timing of the L1 caches, the shared L2 cache, and buses. All contention and queuing for accesses to caches

```
try {
    a();
    synchronized(x){
        b1();
        b2();
        b3();}
    c();}
catch (IOException e) {
    d();
    synchronized(y){
        e1();
        e2();
        e3();}
    f();}
finally {
    g();}
```

Figure 7: A `try-catch` construct containing `synchronized` blocks.

```
a();        // try
COMMIT();
b1();       //
b2();       // try
b3();       //
COMMIT();
c();        // try
COMMIT();
g();        // finally
COMMIT();
```

Figure 8: Runtime transactions created by the non-exception case from code in Figure 7.

```
a();        // try
COMMIT();
b1();       //
b2();       // IOException
COMMIT();
d();        // catch
COMMIT();
e1();       //
e2();       // RuntimeException
COMMIT();
g();        // finally
COMMIT();
```

Figure 9: Runtime transactions created by the exception case from code in Figure 7.

Feature	Description
CPU	1–16 single-issue PowerPC cores
L1	64-KB, 32-byte cache line, 4-way associative, 1 cycle latency
Victim Cache	8 entries fully associative
Bus Width	16 bytes
Bus Arbitration	3 pipelined cycles
Transfer Latency	3 pipelined cycles
L2 Cache	8MB, 8-way, 16 cycles hit time
Main Memory	100 cycles latency, up to 8 outstanding transfers

Table 1: Parameters for the simulated CMP architecture. Bus width and latency parameters apply to both commit and refill buses. L2 hit time includes arbitration and bus transfer time.

and buses is modeled. In particular, the simulator models the contention for the single data port in the L1 caches, which is used for processor accesses and commits for transactions or cache-to-cache transfers for MESI. Table 1 presents the main parameters for the simulated CMP architecture. The victim cache is used for recently evicted data from the L1 cache.

4.2 Results

To evaluate running Java with hardware transactional memory, we ran a collection of benchmarks, as summarized in Table 2, using both locks and transactions. The single-processor version with locks is used as the baseline for calculating the percentage of normalized execution time, with a lower percentage indicating better performance. The execution time is broken down into five components. *Useful* time is for program instruction execution. *L1 Miss* time results from stalls on loads and stores. When running with locks, *Idle/Sync* time is due to the synchronization overhead. When running with transactions, *Commit* time is spent committing the write-set to shared memory, and *Violation* time is the time wasted from data dependency violations.

The micro-benchmarks are based on recent transactional memory papers from Hammond [11] and Harris [14]. For a server benchmark we included SPECjbb2000 [38], while for numerical benchmarks we included the multi-thread Java Grande kernels and applications [18]. For each benchmark, we provide a description of its conversion to transactions.

TestHistogram

`TestHistogram` is a micro-benchmark to demonstrate transactional programming from Hammond [11]. Random numbers between 0 and 100 are counted in bins. When running with locks, each bin has a separate lock to prevent concurrent updates. When running with transactions, each update to a bin is one transaction.

Figure 10 shows the results from `TestHistogram`. While the locking version does exhibit scalability over the single processor baseline, the minimal amount of computation results in significant overhead for acquiring and releasing locks, which dominates time spent in the application. The transactional version eliminates the overhead of locks and demonstrates scaling to 8 CPUs; transactions allow optimistic speculation while locks caused pessimistic waiting. However, at 16 CPUs the performance of the transactional version levels off, because data dependency violations start to become more frequent with this number of bins.

TestHashtable

`TestHashtable` is a micro-benchmark that compares different `java.util.Map` implementations. Multiple threads contend for access to a single `Map` instance. The threads run a mix of 50% `get` and 50% `put` operations. We vary the num-

Benchmark	Description	Source	Input
TestHistogram	create histogram of student test scores	Hammond [11]	80,000 scores
TestHashtable	multithreaded Map read and write	Harris [14]	4,000 `get()`, 4,000 `put()`
TestCompound	multithreaded Map value swaps	Harris [14]	8,000 swaps
SPECjbb2000	Java Business Benchmark	SPEC [38]	368 transactions
Series	Fourier series	Java Grande [18]	100 coefficients
LUFact	LU factorization	Java Grande [18]	500x500 matrix
Crypt	IDEA encryption	Java Grande [18]	300,000 bytes
SOR	sucessive over relaxation	Java Grande [18]	100x100 grid
SparseMatmult	sparse matrix multiplication	Java Grande [18]	5000x5000 matrix
MolDyn	N-body molecular dynamics	Java Grande [18]	256 particles, 10 iterations
MonteCarlo	financial simulation	Java Grande [18]	160 runs
RayTracer	3D ray tracer	Java Grande [18]	16x16 image

Table 2: Summary of benchmark applications

ber of processors and measure the speedup attained over the single processor case.

When running with locks, we run the original synchronized `Hashtable`, a `HashMap` synchronized using the `Collections` class's `synchronizedMap` method, and a `ConcurrentHashMap` from `util.concurrent` Release 1.3.4 [24]. `Hashtable` and `HashMap` use a single mutex, while `ConcurrentHashMap` uses finer grained locking to support concurrent access. When running with transactions, we run each `HashMap` operation within one transaction.

Figure 11 shows the results from `TestHashtable`. The results using locks for `Hashtable` (HT) and `HashMap` (HM) show the problems of scaling when using a simple critical section on traditional multi-processors. The `synchronizedMap` version of `HashMap` actually slows down as more threads are added while `Hashtable` only gets a very small improvement up to 8 processors and a slowdown at 16 processors. While `ConcurrentHashMap` (CHM Fine) shows that fine-grained locking implementation is scalable, this implementation requires significant complexity. With transactions, we can use the simple `HashMap` with the same critical region as `ConcurrentHashMap` and achieve similar performance.

TestCompound

`TestCompound` is a micro-benchmark that compares the same `Map` implementations as `TestHashtable`. Again the threads contend for access to a single object instance, but this time instead of performing a single atomic operation on the shared instance, they need to perform four operations to swap the values of two keys. We perform two experiments that demonstrate both low-contention and high-contention scenarios: the low-contention case uses a 256 element table and the high-contention case uses an 8 element table.

We use the same basic `Map` implementations as with `TestHashtable`. For `Hashtable` and `HashMap`, our critical section uses the `Map` instance as the lock. For `ConcurrentHashMap`, we use two variations of the benchmark representing coarse-grained locking and fine-grained locking. The coarse-grained locking variation uses the `Map` instance as a lock, as with `Hashtable` and `HashMap`. The fine-grained locking variation uses the keys of the values being swapped as locks, being careful to order the acquisition of the locks to avoid deadlock.

The left side of Figure 12 shows the results of `TestCompound` with low contention for locks. Again, running `Hashtable` and `HashMap` with simple locking show the problems of simple locking in traditional systems. Furthermore, the coarse-grained version of `ConcurrentHashMap` (CHM Coarse) demonstrates that simply getting programmers to use data structures designed for concurrent access is not sufficient to maximize application performance. With locking, the application programmer must understand how to use fine-grained locking in their own application to properly take advantage of such data structures. As we continue to increase the number of threads, these applications with coarse-grained locks do not continue to scale, and, as shown, often perform worse than the baseline case. Only fine-grained version of `ConcurrentHashMap` compares favorably with transactions. Transactions have a performance advantage due to speculation. More importantly, transactions are able to beat the performance of fine-grained locks using only the most straightforward code consisting of a single `synchronized` block and the unsynchronized `HashMap`. Hence, transactions allow programmers to write simple code focused on correctness that performs better than complex code focused on performance.

The right side of Figure 12 shows the results of `TestCompound` with high contention for locks. The locking version of `Hashtable`, `HashMap`, and coarse-grained `ConcurrentHashMap` all perform similarly to the low contention case. Fine-grained `ConcurrentHashMap` and transactional performance are both degraded from the low-contention case because of lock contention and data dependency violations. However, in this especially tough case, transactions manage to beat out locks by the largest margin seen in all of these results.

SPECjbb2000

`SPECjbb2000` is a server-side Java benchmark, focusing on business object manipulation. I/O is limited, with clients replaced by driver threads and database storage replaced with in-memory binary trees. The main loop iterates over five application transaction types: new orders, payments, order status, deliveries, and stock levels. New orders and payments are weighted to occur ten times more often than other transactions and the actual order of transactions is randomized. We ran using the default configuration that varies the number of threads and warehouses from 1 to 16, although we measured for a fixed number of 368 application-level transactions instead of a fixed amount of wall clock time. The first part of Figure 13 shows the results from `SPECjbb2000`. Both locking and transactional versions show linear speedup in all configurations because there is very little contention

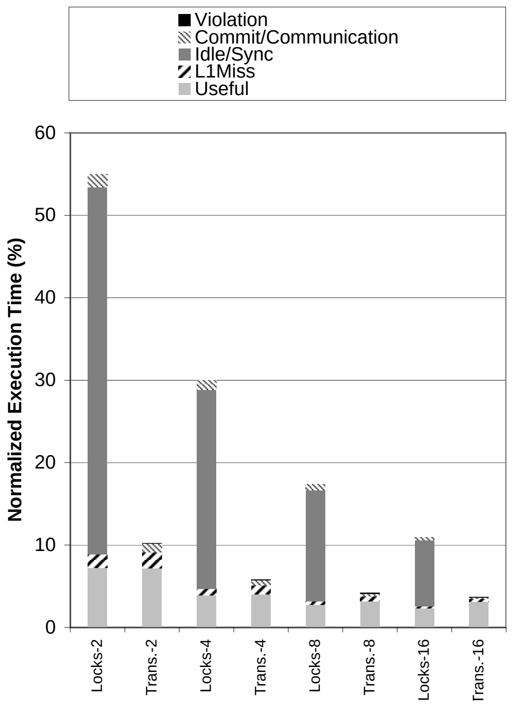

Figure 10: TestHistogram

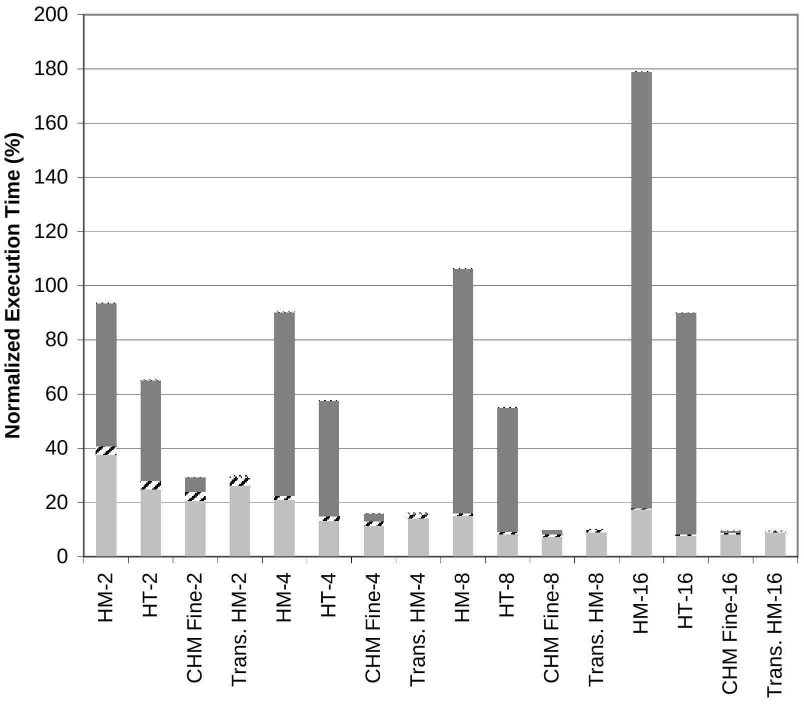

Figure 11: TestHashtable

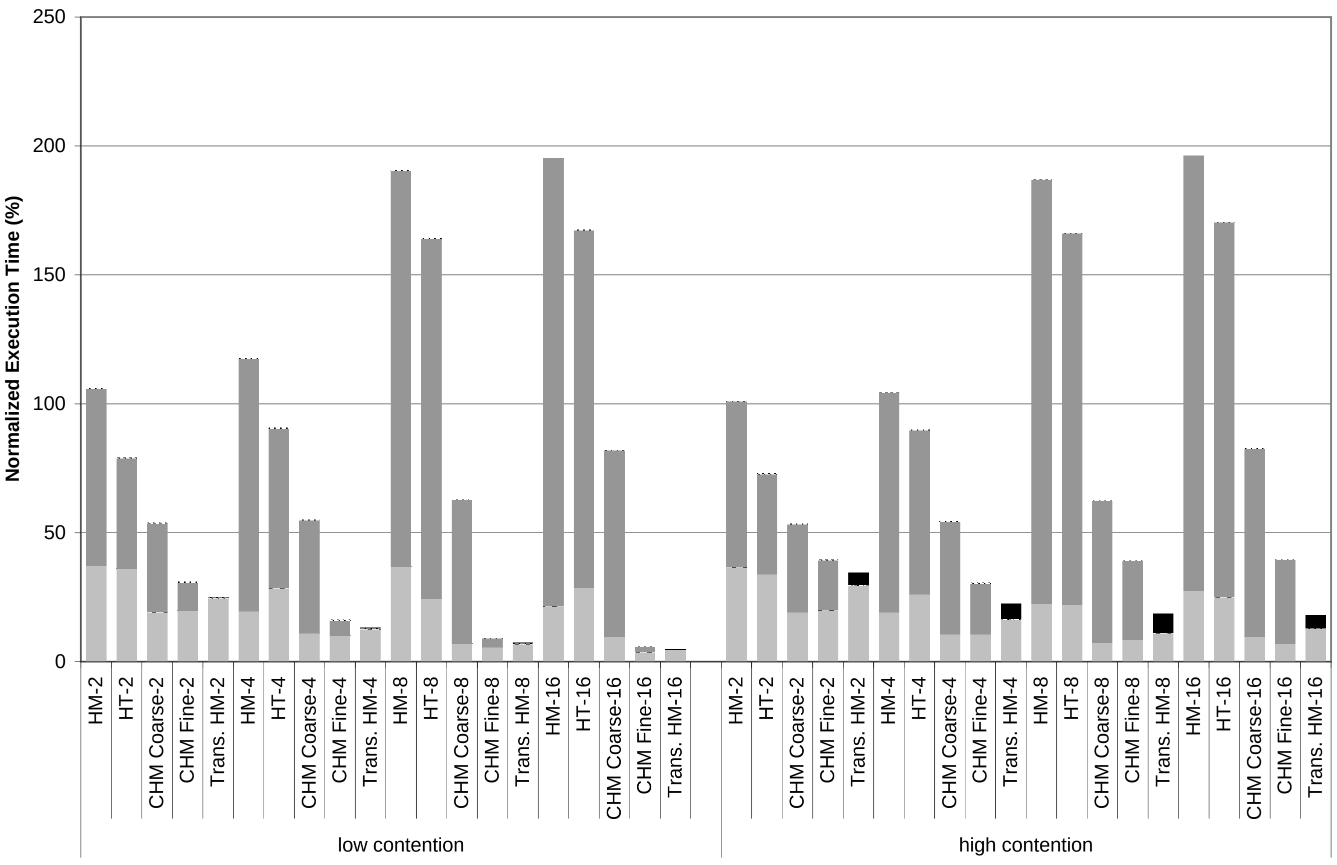

Figure 12: TestCompound

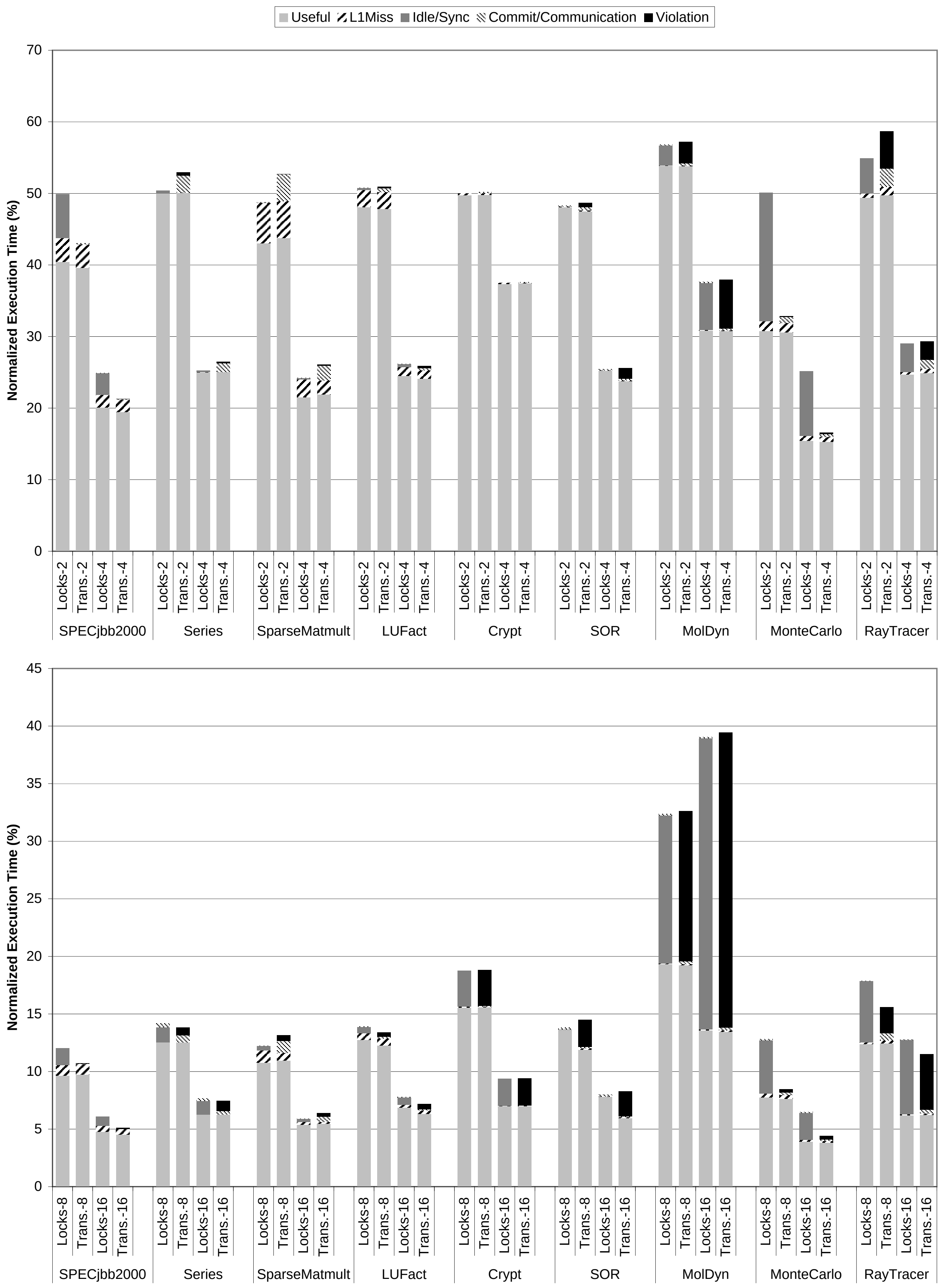

Figure 13: SPECjbb2000 and Java Grande benchmarks. `Series`, `SparseMatmult`, `LUFact`, `Crypt`, `SOR` **are Java Grande section 2 kernels.** `MolDyn`, `MonteCarlo`, `RayTracer` **are Java Grande section 3 applications.**

between warehouse threads. However, the locking version is slightly slower in all cases because it must still pay the locking overhead to protect itself from the 1% chance of an inter-warehouse order. Most importantly, the transactional version of SPECjbb2000 did not need any manual changes to achieve these results; automatically changing `synchronized` blocks to transactions and committing on `Object.wait()` was sufficient.

Java Grande

Java Grande provides a representative set of multithreaded kernels and applications. These benchmarks are often Java versions of benchmarks available in C or Fortran from suites such as SPEC CPU, SPLASH-2, and Linpack. We ran with the input sizes shown in Table 2.

The next five parts of Figure 13 show the results from the Java Grande section 2 kernel programs. These highly-tuned, lock-based kernels show comparable scaling when run with transactions. The transactional versions of `Series` and `SparseMatmult` initially are slightly slower because of commit overhead, but as the number of threads increases, lock contention becomes a factor for the version with locks. Transactions and locks perform equally well on `LUFact` and `Crypt`, with lock contention and violations having comparable cost as threads are added. The transactional version of `SOR` has similar scaling to the lock-based version, with only minimal relative slowdown due to violations.

The final three parts of Figure 13 show the results from the Java Grande section 3 application programs. `MolDyn` has limited scalability for both locks and transactions, with lock overhead being slightly less performance limiting than time lost to violations. `MonteCarlo` exhibits scaling with both locks and transactions, but similar to `TestHistogram`, transactions have better absolute performance because of the overhead cost of locking. `RayTracer` is similar to the kernels `Series` and `SparseMatmult`, with locks performing better with fewer processors and transactions performing better as the number of threads is increased.

Unlike SPECjbb2000, some minor source changes were necessary to achieve these results. `LUFact`, `MolDyn`, `RayTracer` all use a common `TournamentBarrier` class that does not use `synchronized` blocks. Each thread that enters the barrier writes to indicate its progress and then busy waits to continue. With transactions, it was necessary to place the write within a `synchronized` block before busy-waiting, for otherwise all threads loop forever, believing they are the only one to have reached the barrier. `SOR` has the same problem in its own barrier code; again adding a `synchronized` block around the assignment in its barrier code solved the problem. Note that these changes were necessary because the programs did not follow the Java Memory Model rules presented in Section 3.2. We did not find any programs that needed changes beyond those to comply with these rules.

5. RELATED WORK

Recent transactional hardware proposals build on earlier hardware transactional memory work as well as thread-level speculation (TLS). Java programming with transactions builds on earlier work on speculating through locks and transactional memory in general.

5.1 Hardware Transactional Memory

Knight first proposed using hardware to detect data races in the parallel execution of implicit transactions found in mostly functional programming languages such as Lisp [21]. Herlihy and Moss proposed transactional memory as a generalized version of load-linked and store-conditional, meant for replacing short critical sections [16]. Recent proposals such as TCC, UTM, and VTM have relieved earlier data size restrictions on transactions, allowing the development of continuous transactional models [13, 4, 33]. Thread-level Speculation (TLS) uses hardware support to allow speculative parallelization of sequential code [12, 23, 37, 39]. In particular, Chen and Olukotun proposed a system for Java that allowed automatic parallelization of loops using a profile driven JIT compiler on TLS hardware [8].

From the hardware perspective, these earlier systems generally layered speculative execution on top of a conventional cache coherence and consistency protocol. TLS systems allowed speculative threads to communicate results to other speculative threads continuously. In contrast, TCC completely replaces the underlying coherence and consistency protocol, and allows transactions to make their write state visible *only* at commit time, which should generally make it scalable to larger numbers of processors.

From a programming model perspective, earlier hardware transactional memory systems did not allow program control over transaction order. TLS systems allow only ordered execution [29]. TCC allows unordered and ordered transactions, including the use of both models simultaneously.

5.2 Speculating Through Locks

Rajwar and Goodman proposed Speculative Lock Elision (SLE) [32]. SLE speculates through lock acquisition, allowing concurrent execution using hardware support. If a data dependency is detected, all involved processors roll back. Later, Rajwar and Goodman extended SLE to create Transactional Lock Removal (TLR) which used timestamps to avoid rolling back all processors, giving priority to the oldest outstanding work [31].

Martínez and Torrellas proposed Speculative Synchronization [26] based on TLS hardware. It supports speculating not just through locks, but also barriers and flags. These systems have the concept of a *safe thread* that is non-speculative, and that thread has priority when data dependencies are detected, similar to TLR's use of the oldest timestamp.

SLE, TLR, and Speculative Synchronization build upon conventional cache coherence and consistency, such as early hardware transactional memory and TLS. As such, they do not detect run-time races outside of critical regions like continuous transactional hardware does. They always commit in order of transaction creation, providing no way for the programmer to specify the desire for unordered transactions.

5.3 Software Transactional Memory

Shavit and Touitou first proposed a software-only approach to transactional memory, but it was limited to static transactions where the data set is known in advance, such as k-word compare-and-swap [36]. Herlihy et al. overcame this static limitation with their dynamic software transactional memory work, which offered a Java interface through library calls [17]. Harris and Fraser provide language support for software transactional memory, allowing existing Java code to run as part of a transaction and providing an efficient implementation of Hoare's conditional critical regions (CCRs) [14]. Welc et al. provide transactional monitors in Java through JikesRVM compiler extensions, treating `Object.wait()` as a thread yield without committing [42].

Unlike Shavit and Touitou, later proposals support dynamic transactional memory. Unlike Herlihy et al. Harris and Fraser can run *unmodified* code within transactions. Unlike Harris and Fraser, hardware transactions can run both Java and native code within a transaction as well support non-transactional operations within atomic regions. Unlike Welc et al. our proposal can run code containing conditional variables such as barriers that require communication on `Object.wait()`.

Harris et al. later explored integrating software transactional memory with Concurrent Haskell [15]. Haskell is a mostly functional programming language where most memory operations do not need to be part of rollback state. Haskell's type system identifies I/O operations as part of signatures. This allows static checking for non-transactional operations with atomic regions. This work extends the earlier CCR work by allowing an atomic region to block on multiple complex conditions in a composable way.

Vitek et al. have created a calculus for Transaction Featherweight Java (TFJ) to study the semantics of adding transactions to a programming language. They provide soundness and serializability proofs for both an optimistic concurrency model and a two-phase locking protocol [40].

The advantages of hardware transactional memory over software-only versions are performance and language transparency. Software transactional systems encourage small critical sections because of the instruction overhead of maintaining each transaction's data dependencies. Also, running a program consisting entirely of transactions would be prohibitively slow with software transactional memory. Current transactional memory systems, both hardware and software, do not require the programmer to identify which loads and stores are to shared memory. However, for software transactional memory this requires compiler support, preventing transactional code from using mechanisms such as JNI for calling output from non-transactional compilers and hindering reuse of existing code.

6. CONCLUSIONS

Continuous transactional models promise to simplify writing of new parallel applications, but they can also be applied to parallelize today's programs. We have shown that conventional Java language constructs for identifying critical regions can be used to run these existing programs transactionally with minimal changes to applications. We have demonstrated that a continuous transactional system can deliver performance equal to or better than a lock-based implementation using the same programmer-defined critical regions. We also showed that simple, coarse-grained transactions can perform as well as fine-grained locking, demonstrating both the potential for simplified parallel programming and the increased performance of existing applications implemented using coarse-grained locking. The combination of good performance on conventional Java concurrency constructs, combined with the potential for simpler code that transactions offer, provides a foundation for easing the task of parallel software development.

7. ACKNOWLEDGMENTS

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) through the Department of the Interior National Business Center under grant number NBCH104009. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

Additional support was also available through NSF grant 0444470. Brian D. Carlstrom is supported by an Intel Foundation PhD Fellowship.

8. REFERENCES

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[2] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 248–259, 2000.

[3] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. *The Fortress Language Specification*. Sun Microsystems, 2005.

[4] S. Ananian, K. Asanovic, et al. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture*, February 2005.

[5] *The Broadcom BCM-1250 Multiprocessor*. Technical report, Broadcom Corporation, April 2002.

[6] M. Campione, K. Walrath, and A. Huml. *The Java Tutorial*. Addison-Wesley Professional, third edition, January 2000.

[7] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*. ACM Press, 2005.

[8] M. K. Chen and K. Olukotun. The Jrpm system for dynamically parallelizing java programs. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 434–445, June 2003.

[9] C. Flanagan. Atomicity in multithreaded software. In *Workshop on Transactional Systems*, April 2005.

[10] M. Garzaran, M. Prvulovic, et al. Tradeoffs in buffering multi-version memory state for speculative thread-level parallelization in multiprocessors. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture*, February 2003.

[11] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency. In *Proceedings of the 11th International Conference on Architecture Support for Programming Languages and Operating Systems*, Oct. 2004.

[12] L. Hammond, B. Hubbert, M. Siu, M. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra

CMP. *IEEE MICRO Magazine*, March–April 2000.

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

[14] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402. ACM Press, 2003.

[15] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the Principles and Practice of Parallel Programming*, July 2005.

[16] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 289–300, 1993.

[17] M. P. Herlihy, V. Luchangco, M. Moir, and W. M. Scherer. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, July 2003.

[18] Java Grande Forum, *Java Grande Benchmark Suite*. http://www.epcc.ed.ac.uk/javagrande/, 2000.

[19] *Java Specification Request (JSR) 133: Java Memory Model and Thread Specification*, September 2004.

[20] R. Kalla, B. Sinharoy, and J. Tendler. Simultaneous multi-threading implementation in POWER5. In *Conference Record of Hot Chips 15 Symposium*, Stanford, CA, August 2003.

[21] T. Knight. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, August 1986.

[22] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded Sparc processor. *IEEE MICRO Magazine*, 25(2):21–29, March-April 2005.

[23] V. Krishnan and J. Torrellas. A chip multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers, Special Issue on Multithreaded Architecture*, September 1999.

[24] D. Lea. *package util.concurrent*. http://gee.cs.oswego.edu/dl, May 2004.

[25] S. Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Company, Inc., 1999.

[26] J. Martinez and J. Torrellas. Speculative synchronization: Applying thread-level speculation to parallel applications. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

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

[28] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, March 1985.

[29] M. K. Prabhu and K. Olukotun. Using thread-level speculation to simplify manual parallelization. In *Proceedings of the Principles and Practice of Parallel Programming*, pages 1–12, 2003.

[30] W. Pugh. The Java memory model is fatally flawed. *Concurrency - Practice and Experience*, 12(6):445–455, 2000.

[31] R. Rajwar and J. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[32] R. Rajwar and J. R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 294–305. IEEE Computer Society, 2001.

[33] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proceedings of the 32nd International Symposium on Computer Architecture*, June 2005.

[34] R. Raman. UltraSparc Gemini: Dual CPU processor. In *Conference Record of Hot Chips 15 Symposium*, Palo Alto, CA, August 2003.

[35] B. Sandén. Coping with Java threads. *IEEE Computer*, 37(4):20–27, 2004.

[36] N. Shavit and S. Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, Ottawa, Canada, August 1995.

[37] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[38] Standard Performance Evaluation Corporation, *SPECjbb2000 Benchmark*. http://www.spec.org/jbb2000/, 2000.

[39] J. Steffan and T. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, Las Vegas, Nevada, 1998.

[40] J. Vitek, S. Jagannathan, A. Welc, and A. L. Hosking. A semantic framework for designer transactions. In D. A. Schmidt, editor, *Proceedings of the European Symposium on Programming*, volume 2986 of *Lecture Notes in Computer Science*, pages 249–263. Springer-Verlag, 2004.

[41] D. W. Wall. Limits of instruction-level parallelism. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 176–188. ACM Press, 1991.

[42] A. Welc, S. Jagannathan, and A. L. Hosking. Transactional monitors for concurrent objects. In M. Odersky, editor, *Proceedings of the European Conference on Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 519–542. Springer-Verlag, 2004.