The ATOMOΣ Transactional Programming Language

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

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

Abstract

Atomos is the first programming language with implicit transactions, strong atomicity, and a scalable multiprocessor implementation. Atomos is derived from Java, but replaces its synchronization and conditional waiting constructs with simpler transactional alternatives.

The Atomos `watch` statement allows programmers to specify fine-grained *watch sets* used with the Atomos `retry` conditional waiting statement for efficient transactional conflict-driven wakeup even in transactional memory systems with a limited number of transactional contexts. Atomos supports *open-nested* transactions, which are necessary for building both scalable application programs and virtual machine implementations.

The implementation of the Atomos scheduler demonstrates the use of open nesting within the virtual machine and introduces the concept of transactional memory *violation handlers* that allow programs to recover from data dependency violations without rolling back.

Atomos programming examples are given to demonstrate the usefulness of transactional programming primitives. Atomos and Java are compared through the use of several benchmarks. The results demonstrate both the improvements in parallel programming ease and parallel program performance provided by Atomos.

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 Transactional Memory, Conditional Synchronization, Java, Multiprocessor Architecture

1. Introduction

Processor vendors have exhausted their ability to improve single-thread performance using techniques such as simply increasing clock frequency [45, 2]. Hence they are turning *en masse* to single-chip multiprocessors (CMPs) as a realistic path toward scalable performance for server, embedded, desktop, and even laptop platforms [31, 28, 29, 8]. While parallelizing server applications is straightforward with CMPs, parallelizing most other 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 by increasing code complexity and hurting performance in the absence of contention. With such code, it is often easy to end up with code that is prone to deadlocks or priority inversion.

Transactional memory has been proposed as an abstraction to simplify parallel programming [24, 42, 23, 20]. Transactions eliminate locking completely by grouping sequences of object references into atomic and isolated execution units. They provide an easy-to-use technique for non-blocking synchronization over multiple objects, because the programmer can focus on determining where atomicity is necessary, and not its implementation details.

All transactional memory proposals need to detect *violations* of data dependencies between transactions. Violations occur when a transaction's *read set*, the set of all locations read during the transaction, intersects with another transaction's *write set*, the set of all locations written during the transaction. While this basic idea of violation detection is common to all proposals, there is much divergence in the details. We believe a transactional memory proposal should have certain key features: specifically, it should provide a *programming language* model with *implicit transactions*, *strong atomicity*, and demonstrate a scalable *multiprocessor* implementation. To understand why we consider these choices to be important, let us consider the alternatives to these features:

1. *explicit versus implicit:* Some proposals require an *explicit* step to make locations or objects part of a transaction, while other proposals make the memory operations' behavior *implicit* on the transactional state. Implicit transactions require either compiler or hardware support [1]. Older proposals often required explicit instructions or calls to treat specific locations or objects as transactional; however, most proposals now allow existing code to run both transactionally and non-transactionally based on the context. Requiring explicit transactional operations prevents a programmer from composing existing non-transactional code to create transactions. Programmers need to create and maintain transaction-aware versions of existing non-transactional code in order to reuse it.

PLDI '06 June 10–16, 2006, Ottawa, Ontario, Canada

2. *weak atomicity versus strong atomicity:* The atomicity criteria defines how transactional code interacts with non-transactional code. In proposals with *weak atomicity*, transactional isolation is only guaranteed between code running in transactions, which can lead to surprising and non-deterministic results if non-transactional code reads or writes data that is part of a transaction's read or write set. For example, non-transactional code may read uncommitted data from the transaction's write set and non-transactional writes to the transaction's read set may not cause violations. In proposals with *strong atomicity*, non-transactional code does not see the uncommitted state of transactions and updates to shared locations by non-transactional code violate transactions, if needed, to prevent data races.

   From a programming model point of view, strong atomicity makes it easier to reason about the correctness of programs because transactions truly appear atomic with respect to the rest of the program. However, most software implementations of transactional memory have only guaranteed weak atomicity as a concession to performance. Recently, some hardware and hybrid proposals that support unlimited transaction sizes have also only offered weak atomicity. The problem is that programs written for one atomicity model are not guaranteed to work on the other; for a transactional program to be truly portable, it has to be written with a specific atomicity model in mind, potentially hindering its reuse on other systems [6].

3. *library versus programming language:* Some proposals treat transactions simply as a library, while others integrate transactions into the syntax of the programming language. There are many issues with not properly integrating concurrency primitives with programming language semantics as shown in recent work on the Java Memory Model and threads in C and C++ [38, 7]. Clear semantics are necessary to allow modern optimizing compilers to generate safe yet efficient code for multi-processor systems as well as perform transactional memory specific optimizations [22, 1].

4. *uniprocessor versus multiprocessor:* Some proposals require a uniprocessor implementation for correctness, while others take advantage of multiprocessor scaling. Since trends indicate a move to multiprocessors, new programming languages should make it easy to exploit these resources. To properly evaluate transactional memory as an abstraction to simplify parallel programming, it is important for proposals to provide a multiprocessor implementation.

In this paper we introduce the Atomos transactional programming language, which is the first to include implicit transactions, strong atomicity, and a scalable multiprocessor implementation. Atomos is derived from Java, but replaces its synchronization and conditional waiting constructs with transactional alternatives.

The Atomos conditional waiting proposal is tailored to allow efficient implementation with the limited transactional contexts provided by hardware transactional memory. There have been several proposals from the software transactional memory community for conditional waiting primitives that take advantage of transactional conflict detection for efficient wakeup [20, 21]. By allowing programmers more control to specify their conditional dependencies, Atomos allows the general ideas of these earlier proposals to be applied in both hardware and software transactional memory environments.

Atomos supports *open-nested* transactions, which we found necessary for building both scalable application programs and virtual machine implementations. Open nesting allows a nested transaction to commit before its parent transaction [35, 37]. This allows for parent transactions to be isolated from possible contention points in a more general way than other proposals like *early release*, which only allows a program to remove a location from its read set to avoid violations [12].

In this paper we make the following specific contributions:

- We introduce Atomos, the first programming language with strongly atomic transactional memory and a scalable multiprocessor implementation.
- We introduce the `watch` and `retry` statements to allow fine-grained conditional waiting, which is more scalable than other coarse-grained proposals in hardware environments with limited transactional contexts.
- We introduce the `open` statement to create nested transactions that commit independently from their parent.
- We introduce the concept of violation handlers to transactional memory to allow virtual machine implementations to handle expected violations without rolling back.

In our evaluation, implicit transactions and strong atomicity are supported by the Transactional Coherence and Consistency (TCC) hardware transactional memory model [36]. The scalable implementation is built on the design of the underlying Jikes Research Virtual Machine (JikesRVM) and Transactional Coherence and Consistency protocol [4]. Using this environment, we evaluate the relative performance of Atomos and Java to demonstrate the value of programming with transactions. We show not only savings from the removal of lock overhead, but speedup from optimistic concurrency.

While JikesRVM and TCC are well suited to supporting Atomos, there is nothing about Atomos that fundamentally ties it to these systems. Atomos's toughest requirement on the underlying transactional memory system is strong atomicity, which lends itself more naturally toward a hardware transactional memory-based implementation. Although there has been recent research into strongly atomic software transactional memory systems, native code poses a further challenge to their use by Atomos. Typically these systems prohibit the calling of native code within transactions, significantly restricting the flexibility of the program. Atomos leverages the JikesRVM scheduler thread architecture in its implementation of conditional waiting, but the design could be adapted to other timer-based schedulers.

The rest of the paper is organized as follows. Section 2 discusses earlier work on integrating transactions and programming languages. Section 3 describes the Atomos programming language and provides motivating examples. Section 4 demonstrates how both fine-grained conditional waiting and loop speculation are implemented using the basic atomicity features of Atomos. In Section 5, we evaluate the usability of Atomos as well as the relative performance of transactional Atomos programs and lock-based Java programs. Finally, we conclude in Section 6.

2. Related Work

Any transactional programming language proposal builds on work in hardware and software transactional memory as well as on earlier work integrating database-style transactions into programming languages. As we review transactional memory-related work in Section 2.1 and Section 2.2, we will use the four properties from Section 1 to differentiate the various proposals. A summary of this is provided in Table 1, showing that Atomos is the first transactional programming language with implicit transactions, strong atomicity, and a multiprocessor implementation.

2.1 Hardware Transactional Memory

Hardware transactional memory variants have now been around for almost twenty years and have focused on multiprocessor im-

Name	Implicit Transactions	Strong Atomicity	Programming Language	Multi-Processor
Knight [30]	Yes	Yes	No	Yes
Herlihy & Moss [24]	No	No	No	Yes
TCC [18, 17]	Yes	Yes	No	Yes
UTM [5]	Yes	No	No	Yes
LTM [5]	Yes	Yes	No	Yes
VTM [39]	Yes	Yes	No	Yes
Shavit & Touitou [42]	No	No	No	Yes
Herlihy et al. [23]	No	No	No	Yes
Harris & Fraser [20]	Yes	No	Yes	Yes
Welc et al. [46]	Yes	No	Yes	Yes
Harris et al. [21]	Yes	Yes	Yes	No
AtomCaml [40]	Yes	Yes	Yes	No
X10 [11]	Yes	No	Yes	Yes
Fortress [3]	Yes	No	Yes	Yes
Chapel [12]	Yes	No	Yes	Yes
McRT-STM [41, 1]	Yes	No	Yes	Yes
Atomos	Yes	Yes	Yes	Yes

Table 1. Summary of transactional memory systems. The first section lists hardware transactional memory systems. The second section lists software transactional memory systems.

plementations. Knight first proposed using hardware to detect data races in parallel execution of implicit transactions found in mostly functional programming languages such as Lisp [30]. This proposal had two of the important features of transactional memory: implicit transactions and strong atomicity. However, the transaction granularity was not made visible to the programmer, with each store acting as a commit point and executing in sequential program order. Herlihy and Moss proposed transactional memory as a generalized version of load-linked and store-conditional, meant for replacing short critical sections [24].

Recent proposals such as TCC, UTM/LTM, and VTM have relieved earlier data size restrictions on transactions, allowing the development of continuous transactional models [18, 5, 39]. TCC provides implicit transactions, strong atomicity, and some features for speculation and transactional ordering [17]. UTM and LTM are related proposals that both provide implicit transactions. However, these proposals differ in atomicity, with UTM providing weak atomicity and LTM providing strong atomicity [6]. Finally, VTM provides implicit transactions and strong atomicity.

2.2 Software Transactional Memory

Shavit and Touitou first proposed a software-only approach to transactional memory, but it was limited to explicit static transactions with weak atomicity where the data set is known in advance, such as k-word compare-and-swap [42]. Herlihy et al. overcame this static limitation with their dynamic software transactional memory work, which offered explicit transactions, strong atomicity, and a Java library interface [23]. Harris and Fraser provide the first implicit software transactional memory with programming language support, allowing existing Java code to run as part of a transaction and providing an efficient implementation of Hoare's conditional critical regions (CCRs) [20]. Welc et al. provide transactional monitors in Java through JikesRVM compiler extensions, treating `Object.wait()` as a thread yield without committing [46]. Like Harris, this proposal has implicit transactions with weak atomicity. Unfortunately, the transactional interpretation of `Object.wait()` by Welc et al. can cause existing code such as barriers to fail [10].

Harris et al. later explored integrating software transactional memory with Concurrent Haskell, providing implicit transactions, strong atomicity, and the `orElse` construct for composing conditional waiting in a uniprocessor environment [21]. AtomCaml explicitly takes advantage of a uniprocessor-only implementation to achieve its implicit transactions with strong atomicity [40]. AtomCaml builds transactional conditional waiting entirely from language primitives without any additional implementation support. Recently X10, Fortress, and Chapel have been proposed, which all provide implicit transactions, weak atomicity, and programming language support for transactions in multiprocessor environments [11, 3, 12]. McRT-STM provides a multi-core runtime that supports software transactional memory for C++ via a library interface and for Java and C# through language extensions backed by a just-in-time (JIT) compiler with transactional memory-specific optimizations [41, 1].

2.3 Programming Languages with Durable Transactions

While most of the related work comes from the area of transactional memory, there has been a body of work integrating transactional persistence and programming languages. ARGUS was a programming language and system that used the concepts of guardians and actions to build distributed applications [33]. The Encina Transactional-C language provided syntax for building distributed systems including nested transactions, commit and abort handlers, as well as a "prepare" callback for participating in two-phase commit protocols [43]. Encina credits many of its ideas to work on Camelot and its Avalon programming language [14]. The SQL standards define how SQL can be embedded into a variety of programming languages [25]. There are also systems such as PJama that provide orthogonal persistence allowing objects to be saved transparently without explicit database code [27].

3. The Atomos Programming Language

The Atomos programming language is derived from Java by replacing locking and conditional waiting with transactional alternatives. The basic transactional semantics are then extended through open nesting to allow programs to communicate between uncommitted transactions. Finally, commit and abort callbacks are provided so programs can either defer non-transactional operations until commit or provide compensating operations on abort.

3.1 Transactional Memory with Closed Nesting

Transactions are defined by an `atomic` statement similar to other proposals [20, 11, 3, 12]. Because Atomos specifies strong atomicity, statements within an `atomic` statement appear to have a serialization with respect to other transactions as well as to reads and writes outside of transactions; reads outside of a transaction will not see any uncommitted data and writes outside a transaction can cause a transaction to roll back. Here is a simple example of an atomic update to a global counter:

```
atomic { counter++; }
```

Nested `atomic` statements follow closed-nesting semantics, meaning that the outermost `atomic` statement defines a transaction that subsumes the inner `atomic` statement. When a nested transaction commits, it merges its read and write sets with its parent transaction. When a transaction is violated, only it and its children need to be rolled back; the parent transaction can then restart the nested child.

The use of `atomic` statements conceptually replaces the use of `synchronized` statements. Studies show that this is what programmers usually mean by `synchronized` in Java applications [15]. However, this is not to say that programmers can blindly substitute one statement for the other, as it can affect the semantics of existing code [6]. Fortunately, this does not seem to be a common problem in practice [10].

```
public int get (){
    synchronized (this) {
        while (!available)
            wait();
        available = false;
        notifyAll();
        return contents;}}
public void put(int value){
    synchronized (this) {
        while (available)
            wait();
        contents  = value;
        available = true;
        notifyAll();}}
```

```
public int get() {
    atomic {
        if (!available) {
            watch available;
            retry;}
        available = false;
        return contents;}}
public void put (int value) {
    atomic {
        if (available) {
            watch available;
            retry;}
        contents  = value;
        available = true;}
```

Figure 1. Comparison of producer-consumer in Java (left) and Atomos (right). The Java version has an explicit loop to retest the condition where the Atomos rollback on `retry` implicitly forces the retesting of the condition. Java requires an explicit notification for wakeup where Atomos relies on the violation detection of the underlying transactional memory system.

```
synchronized (lock) {
    count++;
    if (count != thread_count)
        lock.wait();
    else
        lock.notifyAll();}
```

```
atomic {
    count++;}
atomic {
    if (count != thread_count) {
        watch count;
        retry;}}
```

Figure 2. Comparison of a barrier in Java (left) and Atomos (right). `count` is initialized to zero for new barriers. The Java version implicitly has two critical regions since the `wait` call releases the monitor. In Atomos, the two critical regions are explicit.

The concept of the `volatile` field modifier is replaced with an `atomic` field modifier as proposed in Chapel [12]. Fields marked with the `atomic` modifier are accessed as small closed nested transactions that may be top level or nested as specified above. This serves one purpose of `volatile`, since it restricts the reordering of access between top-level transactions. However, another purpose of transactions is to force visibility of updates between threads. This usage of `volatile` is discussed below in Section 3.3 on reducing isolation between transactions.

The initial proposal for Atomos prohibits starting threads inside of transactions. This eliminates the semantic questions of nested parallelism where multiple threads could run within a single transactional context. Other proposals are also exploring such semantics, but there does not seem to be a consensus about the practical usefulness or the exact semantics of this feature [34, 13].

3.2 Fine-Grained Conditional Waiting

Atomos conditional waiting follows the general approach of an efficient Conditional Critical Region (CCR) implementation: a program tests some condition and, finding it false, waits to restart until it has reason to believe the condition might now be true [20]. One nice semantic property of CCRs is that the waiting thread does not need to coordinate *a priori* with other threads that might be affecting the condition. Unfortunately, this lack of connection between waiting threads and modifying threads historically led to problematic performance for CCRs because either the condition was reevaluated too often, resulting in polling-like behavior, or not often enough, resulting in delayed wakeup. Transactional memory has made CCRs efficient by using the read set of the transaction as a tripwire to detect when to reevaluate the condition: when a violation of the read set is detected, another transaction has written a value that was read as part of evaluating the condition, so there is some reason to believe that the value of the condition might be true.

One problem with this approach is that it requires a transactional context with a potentially large read set to remain active to listen for violations even while the thread has logically yielded. While this can scale well for software transactional memory systems that maintain such structures in memory, it is problematic for hardware transactional memory systems with a limited number of hardware transactional contexts, typically one per processor.

Atomos refines this general approach by allowing a program to specify a *watch set* of locations of interest using the `watch` statement. After defining the watch set, the program calls `retry` to roll back and yield the processor. The implementation of the `retry` statement communicates the watch set to the scheduler, which then listens for violations on behalf of the now yielded thread. By rolling back, `retry` allows the waiting thread to truly yield both its processor resource and transactional context for use by other threads. Violation of the watch set and restart of waiting threads are transparent to the writing thread, which does not need to be concerned about a separate explicit notification step as is necessary in most lock-based conditional waiting schemes.

Figure 1 shows a simple producer-consumer example derived from Sun's Java Tutorial to compare and contrast Java's conditional variables with Atomos's `watch` and `retry` [9]. Although these two versions seem very similar, there are notable differences. First, note that Java requires an explicit association between a lock, in this case `this`, and the data it protects, in this case `available` and `contents`. Atomos transactions allow the program to simply declare what they want to appear atomic. Second, Java's `wait` releases the lock making side effects protected by the lock visible. Instead, Atomos's `retry` reverts the transaction back to a clean state. Third, because Atomos reverts back to the beginning of the transaction, the common Java mistake of using an `if` instead of a `while` in testing the condition is eliminated, since calling `retry` ensures that the program will always reevaluate the condition. Finally, Java requires an explicit notify to wake up waiting threads, where Atomos relies on implicit violation handling. This reduces the need for coordination and allows threads to wait for conditions without defining *a priori* conventions for notification.

To understand why this third difference is important, consider Figure 2 which shows an example derived from Java Grande's `SimpleBarrier` [26]. As noted above, Java `wait` will release the lock and make the updated `count` visible to other threads. However, re-

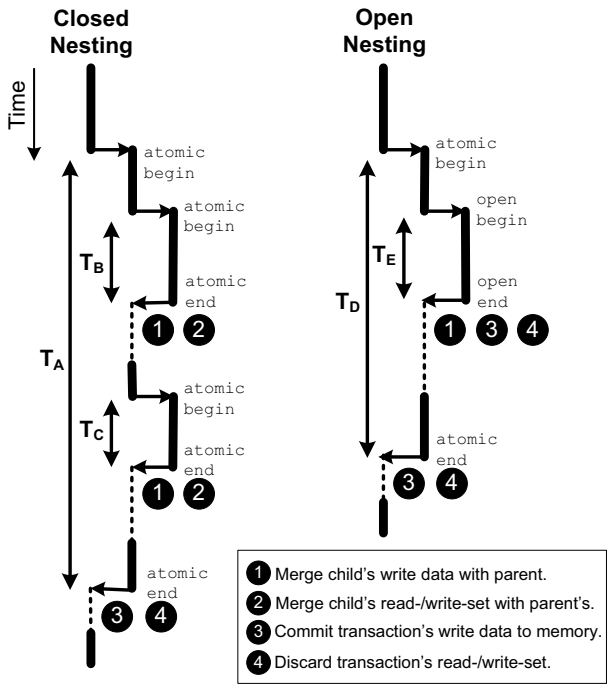

Figure 3. Timeline of three nested transactions: two closed-nested and one open-nested. Merging of data (❶ and ❸) and read-/write-sets (❷ and ❹) is noted at the end of each transaction.

placing Java `wait` with Atomos `retry` would cause the code to roll back, losing the `count` update. All threads would then think they were first to the barrier and the program will hang. To create a transactional version, we need to rewrite the original code into two transactions, one that updates the `count` and another that watches for other threads to reach the barrier. Code with side effects before conditional waiting is not uncommon in existing parallel code, as a similar example is given in Sun's Java Tutorial [9]. SPECjbb2000 also contains such an example with nested `synchronized` statements [44]. Fortunately, although they share a common pattern, such examples make up a very small fraction of the overall program and are generally easy to rewrite.

3.3 Open Nesting

Basic transactional behavior depends on the detection of conflicting read and write sets. However, in the world of databases, transactions often reduce their isolation from each other to gain better performance. Chapel [12] provides the early release construct, which is intended to prevent dependencies between transactions. Atomos takes a different approach by providing open nesting, which allows communication from within uncommitted transactions while still providing atomicity guarantees for updates made within the transaction.

The `open` statement allows a program to start an open-nested transaction. Where a closed-nested transaction merges its read and write set into its parent at commit time, an open-nested transaction commit always makes its changes globally visible immediately. For example, in Figure 3 when T_E commits, its updates are made visible to both other transactions in the system as well as its parent, T_D. In contrast, when closed-nested transactions such as T_B and T_C commit, their changes are only made available to their parent, T_A. Only when the parent T_A commits are changes from T_B and T_C made visible to the rest of the system. Like closed-nested transactions, the violation of an open-nested child does not roll back the parent, allowing the child to be resumed from its starting point, minimizing lost work.

Open nesting semantics can seem surprising at first, since changes from the parent transaction can be committed if they are also written by the open-nested child transaction, seemingly breaking the atomicity guarantees of the parent transaction. However, in the common usage of open-nested transactions, the write sets are typically disjoint. This can be enforced through standard object-oriented encapsulation techniques.

To see how open-nested transactions are useful to application programs, consider an example of generating unique identifiers for objects using a simple counter derived from SPECjbb2000 [44]. A program may want to create objects, obtain an unique identifier, and register the object atomically as sketched in the following code:

```
public static int generateID {
    atomic {
        return id++;}}
public static void createOrder (...) {
    atomic {
        Order order = new Order();
        order.setID(generateID());
        // finish initialization of order.
        // this could include creating more
        // objects which could mean more
        // calls to generateID.
        ...;
        orders.put(new Integer(order.getID()),
                   order);}}
```

However, doing so will mean that many, otherwise unrelated transactions will have conflicting accesses to the `id` variable, even though the rest of their operations may be non-conflicting. By changing `generateID` to use open nesting, the counter can be read, updated, and committed quickly, with any violation rollback limited to the open-nested transaction, and not the parent application transactions:

```
public static open int generateID {
    open {
        return id++;}}
```

Open-nested transactions can also be used for more general inter-transaction communication like that found with transaction synchronizers [34].

Open-nested transactions allow threads to communicate between transactions while minimizing the risk of violations. Their use has some similarity to `volatile` variables in that commit forces writes to be immediately visible even before the parent transaction commits. For this reason, we also allow an `open` field modifier for cases where a `volatile` field was used within a `synchronized` statement.

Once an open-nested transaction has committed, a rollback of one of its parent transactions is independent from the completed open-nested transaction. If the parent is restarted, the same open-nested transaction may be rerun. In the unique identifier example, this is harmless as it just means there might be some gaps in the identifier sequence. In other cases, another operation might be necessary to compensate for the effects of the open-nested transaction, as we will discuss in the next section.

Open-nested transactions are also very useful in virtual machine implementation where runtime code runs implicitly within the context of a program transaction. For example, the JikesRVM JIT compiler adds its own runtime code to methods as it compiles them for several purposes: a.) code that checks for requests from the scheduler to yield the processor for other threads to run, b.) code that checks for requests from the garbage collector to yield

the processor for the garbage collector to run, and c.) code that increments statistic counters, such as method invocation counters, that are used to guide adaptive recompilation. By using open-nested transactions, the runtime can check for these requests or increment these counters without causing the parent application transactions to roll back. We will discuss the use of open-nested transactions for virtual machine implementation further in Section 4.1, where we discuss how watch sets are communicated to the scheduler.

3.4 Transaction Handlers

In database programming, it is common to run code based on the outcome of a transaction. Transactional-C provided `onCommit` and `onAbort` handlers as part of the language syntax. Harris extended this notion to transactional memory with the `ContextListener` interface [19]. Harris introduces the notion of an `ExternalAction`, which can write state out of a transactional context using Java `Serialization` so that abort handlers can access state from the aborted transaction.

For Atomos, we feel that open-nested transactions fill the role of `ExternalAction` by providing a way to communicate state out of a transaction that might later be needed after rollback. We provide separate `CommitHandler` and `AbortHandler` interfaces so that one or the other may be registered independently:

```
public interface CommitHandler {
    public void onCommit();}
public interface AbortHandler {
    public void onAbort();}
```

Each nesting level can have its own handlers. When registered, a handler can be associated with any currently nested transaction and is run at the conclusion of that nested transaction in a new transactional context.

In database programming, transaction handlers are often used to integrate non-transactional operations. For example, if a file is uploaded to a temporary location, on commit it would be moved to a permanent location and on abort it would be deleted. In transactional memory programming, transactional handlers serve similar purposes. Transaction handlers can be used to buffer output or rewind input performed within transactions. Transaction handlers can be used to provide compensation for open-nested transactions. In our JIT example, 100% accurate counters were not required. If a method is marked as invoked and then rolls back, it is not necessary to decrement the counter. However, programs such as the SPECjbb2000 benchmark that keep global counters of allocated and deallocated objects want accurate results. An abort handler can be used to compensate the open transaction, should a parent transaction abort. Further details on handlers, including an I/O example, can be found in [35].

4. Implementing Transactional Features

In this section, we will detail two transactional programming constructs implemented in Atomos. The first is a discussion of our implementation of Atomos `watch` and `retry`, which demonstrates a use of open-nested transactions and violation handlers. The second is a demonstration of how loop speculation can be built with closed-nested transactions.

4.1 Implementing `retry`

Implementing violation-driven conditional waiting with hardware transactional memory is challenging because of the limited number of hardware transactional contexts for violation detection, as mentioned previously in Section 3.2. The problem is making sure someone is listening for the violation even after the waiting thread has yielded. The Atomos solution uses the existing scheduler thread of the underlying JikesRVM implementation to listen for violations on behalf of waiting threads.

We use a *violation handler* to communicate the watch set between the thread and the scheduler. A violation handler is a callback that allows a program to recover from violations instead of necessarily rolling back. Violation handlers are a more general form of abort handlers that allow complete control over what happens at a violation including whether or not to roll back, where to resume after the handler, and what program state will be available after the handler. Violation handlers run with violations blocked by default, allowing a handler to focus on handling one violation without having to worry about handler re-entrance. If a violation handler chooses to roll back the violated transactions, any pending violations are discarded. Otherwise the violation handler will be invoked with the pending violation after it completes.

Violation handlers are not a part of the Atomos programming language. They are the mechanism used to implement the higher-level Atomos `AbortHandler`, which is not allowed to prevent roll back. At the operating system level, violation handlers are implemented as synchronous signal handlers that run in the context of the violated transaction. The handler can choose where to resume the transaction or to roll back the transaction. Here we show a higher level `ViolationHandler` interface that is more restrictive than the general purpose signal handler:

```
public interface ViolationHandler {
    public boolean onViolation(
        Address violatedAddress);}
```

This restricted form of handler can either return true if it handled the violation and wants to simply resume the interrupted transaction or return false if it wants the transaction to roll back. The underlying lower-level violation signal handler takes care of calculating the address of where to resume.

Figure 4 sketches out the basic implementation of Atomos conditional waiting. The `watch` statement simply adds an address to a thread-local watch set. The `retry` implementation uses `open` to send this watch set to the scheduler thread, which is effectively listening for requests from the other threads using a violation handler. Once the thread is sure that the scheduler is listening on its watch set, it can roll back and yield the processor. The scheduler's violation handler serves three purposes. First, it watches for requests to read watch set addresses. Second, it handles requests to cancel watch requests when the thread is violated in the process of waiting. Finally, it handles violations to watch set addresses by ensuring that watching threads are rescheduled.

To consider a simple example, let us walk through the producer-consumer code from Figure 1. Suppose a consumer thread finds that there is no data available. It requests to watch `available`, which simply adds the address of `available` to a local list. When the thread calls `retry`, the thread uses an open-nested transaction to send the `available` address to the scheduler, which then reads it. The scheduler then uses its own open-nested transaction to acknowledge that it has read the address, so that the original thread can now roll back and yield. Later on, a producer makes some data available. Its write to `available` causes the scheduler to be violated. The scheduler finds that it had read the address of `available` on behalf of the original consumer thread, which it then reschedules for execution.

Figure 5 presents a sketch of code for the Atomos scheduler implementation with the code run by the waiting thread on the left and the code run by the scheduler on the right. The `watch` implementation simply adds the address to the thread local `watchSet` list. Note that the address never needs to be explicitly removed from the wait set because this transaction will be rolled back either by a violation from another thread or when the thread is suspending.

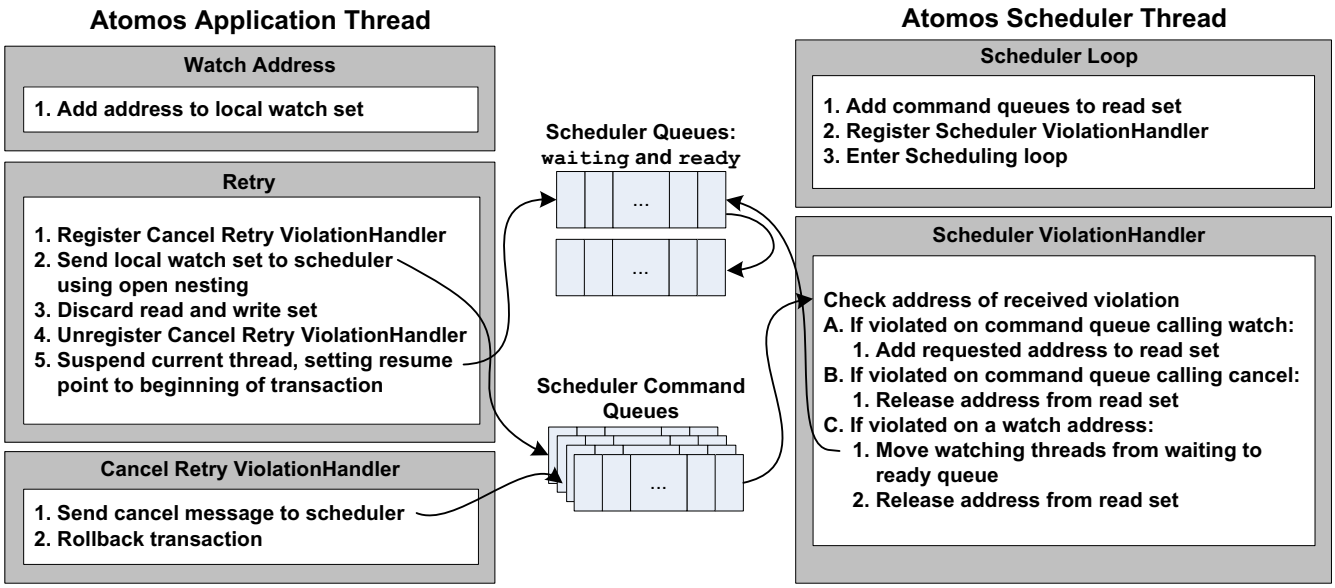

Figure 4. Conditional synchronization using open nesting and violation handlers. Waiting threads communicate their watch sets to the scheduler thread via scheduler command queues in shared memory that interrupt the scheduler loop using violations. The Cancel Retry ViolationHandler allows the waiting thread to perform a compensating transaction to undo the effects of its open-nested transactions in the event of rollback of the parent transactions.

```
// watch keyword adds an address to local wait set
void watch(Address a){
  VM_Thread.getCurrentThread().watchSet.add(a); }

// retry keyword implementation
void retry(){
  VM_Thread thread = VM_Thread.getCurrentThread();
  List watchSet = thread.watchSet;
  // register "cancel retry violation handler" to
  // cleanup scheduler if we violated before yield
  VM_Magic.registerViolationHandler(retryVH);
  for (int i=0,s=watchSet.size();i<s;i++){
      Address a=(Address)watchSet.get(i);
      open {
          // write address where scheduler is reading
          thread.schedulerAddress = a;
          // wakeup the scheduler violation handler
          thread.schedulerWatch = true; }
      // busy wait until we hear back
      open { if (thread.scheduleWatch) for(;;) ; }}
  // clear our read set to avoid violations
  // now that scheduler is listening for us
  VM_Magic.discardState();
  // safe to unregister now that read set cleared
  VM_Magic.unregisterViolationHandler(retryVH);
  // store resume context from checkpoint and yield
  thread.suspend(); }

// cancel retry violation handler (retryVH)
boolean onViolation(Address a){
  VM_Thread thread = VM_Thread.getCurrentThread();
    open {
      thread.schedulerCancel = true; }
    open { if (thread.schedulerCancel) for(;;) ; }
  return false; } // rollback transaction
```

```
// Scheduler violation handler
boolean onViolation(Address a){
  VM_Thread t = schedulerWatches.get(a);
  if (t != null) { // case A: watch request
    Address address;
    // read the next watch address
    open { address = t.schedulerAddress; }
    address.loadWord(); // load adds to read set
    open {
      // update address and thread mappings for below
      addressToThreads.get(address).add(t);
      threadToAddresses.get(t).add(address);
      // let the sender continue
      thread.schedulerWatch = false;
      return true; }} // never rollback transaction
  t = schedulerCancels.get(a);
  if (t != null) { // case B: retry cancel request
    open {
      List addresses = threadToAddresses.remove(t);
      for (int j=0, sj=addresses.size();j<sj;j++){
        Address a = (VM_Thread)addresses.get(j);
        Map threads = addressToThreads.get(address);
        threads.remove(t);
        if (threads.isEmpty()) {
            VM_Magic.releaseAddress(a); }}
      thread.schedulerCancel = false;
      return true; }} // never rollback transaction
  open { // notification for some thread?
    List threads = addressToThreads.remove(a);
    if (threads != null) { // case C: resume threads
      for (int i=0, si=threads.size();i<si;i++){
        VM_Thread t = (VM_Thread)threads.get(i);
        t.resume();
        List addresses = threadToAddresses.remove(t);
        for (int j=0, sj=addresses.size();j<sj;j++){
          Address a = (Address)threads.get(j);
          Map moreThreads = addressToThreads.get(a);
          moreThreads.remove(t);
          if (moreThreads.isEmpty()) {
              VM_Magic.releaseAddress(a); }}}}}
  return true; }  // never rollback transaction
```

Figure 5. Implementation details of Atomos `watch` and `retry` using violation handlers. Following the convention set by Figure 4, the code on the left runs in the waiting thread and the code on the right runs as part of the scheduler. The Scheduler ViolationHandler cases A, B, and C from Figure 4 are marked with comments in the Scheduler `onViolation` code.

The retry implementation needs to communicate the addresses from the watchSet to the scheduler thread so it can receive violations on behalf of the waiting thread after it suspends. To do this, the waiting thread uses open-nested transactions. However, the transaction could be violated while communicating its watch set. The scheduler would then be watching addresses for a thread that has already been rolled back. In order to keep consistent state between the the two threads, the waiting thread uses a violation handler to perform a compensating transaction to let the scheduler know to undo the previous effects of the waiting thread's open-nested transactions. In order to achieve this, it is important to register the retryVH violation handler before any communication with the scheduler. This violation handler is only unregistered after the thread's watchSet has been communicated and the read set of the waiting thread has been discarded to prevent violations.

In this example implementation of retry, schedulerAddress is used to communicate watchSet addresses to the scheduler and schedulerWatch is set to true to violate the scheduler thread, invoking its violation handler. After the first open transaction, a second open is used to listen for an acknowledgment from the scheduler so that the waiting thread has confirmation of the watchSet transfer before discarding state, ensuring that at least one of the two threads will receive the desired violations at any time.

Below the retry code is retryVH, the violation handler for the waiting thread. It uses the same technique to communicate with the scheduler. The violation handler returns false to indicate that the waiting thread should be rolled back, allowing the waiting thread to reevaluate its wait condition.

The right side of Figure 5, shows the onViolation code for the scheduler thread. It uses the schedulerWatches map to determine if this is a watch request from a waiting thread. The schedulerWatches and related schedulerCancels maps are established when the VM_Thread objects are created during virtual machine initialization; programming language threads are multiplexed over the VM_Thread instances. If the violation is from a known schedulerWatch address, the value in schedulerAddress field is added to the read set of the scheduler simply by loading from the address. The schedulerAddress value is read in an open-nested transaction to avoid adding the location of this field to the scheduler read set. The thread and address information is then used to update addressToThreads and threadToAddresses maps. The addressToThreads is used when a violation is received to determine the threads that have requested wakeup. The threadToAddresses map is used to track addresses to remove from the scheduler read set when there is a cancel request.

If the violation is instead from a schedulerCancel address, the scheduler needs to remove from its read set any addresses that it was watching solely for the requesting thread, being careful to remove the location only if it is not in the watch set of any other thread.

The final case in the scheduler code is to resume threads on a watch set violation. After resuming, the threads will then reevaluate their conditional waiting code. In addition, the watch sets of the resumed threads are removed from the scheduler read set as necessary, similar to the code in the cancel case.

The scheduler thread must be very careful in managing its read set or it will miss violations on behalf of other threads. The violation handler uses exclusively open-nested transactions to update addressToThreads and threadToAddresses and schedulerCommand. The scheduler main loop must also use only open-nested transactions as committing a close nested transaction will empty the carefully constructed read set. Fortunately, such complex uses of open nesting are generally confined to runtime system implementation and application uses are more straightforward as shown in the previous counter example.

4.2 Loop Speculation

The t_for loop speculation statement allows sequential loop-based code to be quickly converted to use transactional parallelism [17]. When used in ordered mode, it allows sequential program semantics to be preserved while running loop iterations in parallel. It also allows multiple loop iterations to be run in larger transactions to help reduce potential overheads caused by many small transactions, similar to loop unrolling.

Figure 6 gives a schematic view of running loop iterations in parallel using t_for. In Figure 6a, the sequential order is preserved by requiring that loop iterations commit in order, even when iteration length varies. Note that although the commit of iterations has to be delayed to preserve sequential order at the start of the t_for loop, the resulting pattern of staggered transactions typically leads to a pipelining effect where future iterations do not have to be delayed before committing. In Figure 6b, iterations are free to commit in any order, removing stall times from the ordering case.

In Atomos, statements like t_for can be implemented as library routines. For example, we can recast t_for as a Loop.run method that takes a LoopBody interface similar to the use of Thread and Runnable:

```
public class Loop {
    public static void run (
        boolean  ordered,
        int      chunk,
        List     list,
        LoopBody loopBody);}
public interface LoopBody {
    public void run (Object o);}
```

The ordered argument allows the caller to specify sequential program order if needed, otherwise transactions commit in "first come, first served" order. The chunk specifies how many loop iterations to run per transaction. The list argument specifies the data to iterate over. The loopBody specifies the method to apply to each loop element. Before we investigate the details of the implementation of Loop.run, let us consider an example of using loop speculation. Consider the following histogram example program:

```
void histogram(int[] A,int[] bin){
  for(int i=0; i<A.length; i++){
    bin[A[i]]++;}}
```

We can use a Loop.run routine to convert histogram into the following parallel program:

```
void histogram(int[] A,int[] bin){
  Loop.run(false,20,Arrays.asList(A),new LoopBody(){
    public void run(Object o){
      bin[A[((Integer)o).intValue()]]++;}}}
```

This version of histogram runs loop chunks of 20 loop iterations in parallel as unordered transactions.

Figure 7 shows a simple implementation of Loop.run. The unordered case is relatively straightforward with various chunks running atomically in parallel with each other. The ordered case is more interesting in its use of a loop within a closed-nested transaction to stall the commit until previous iterations have completed without rolling back the work of the current iterations. This general pattern can be used to build arbitrary, application-specific ordering patterns.

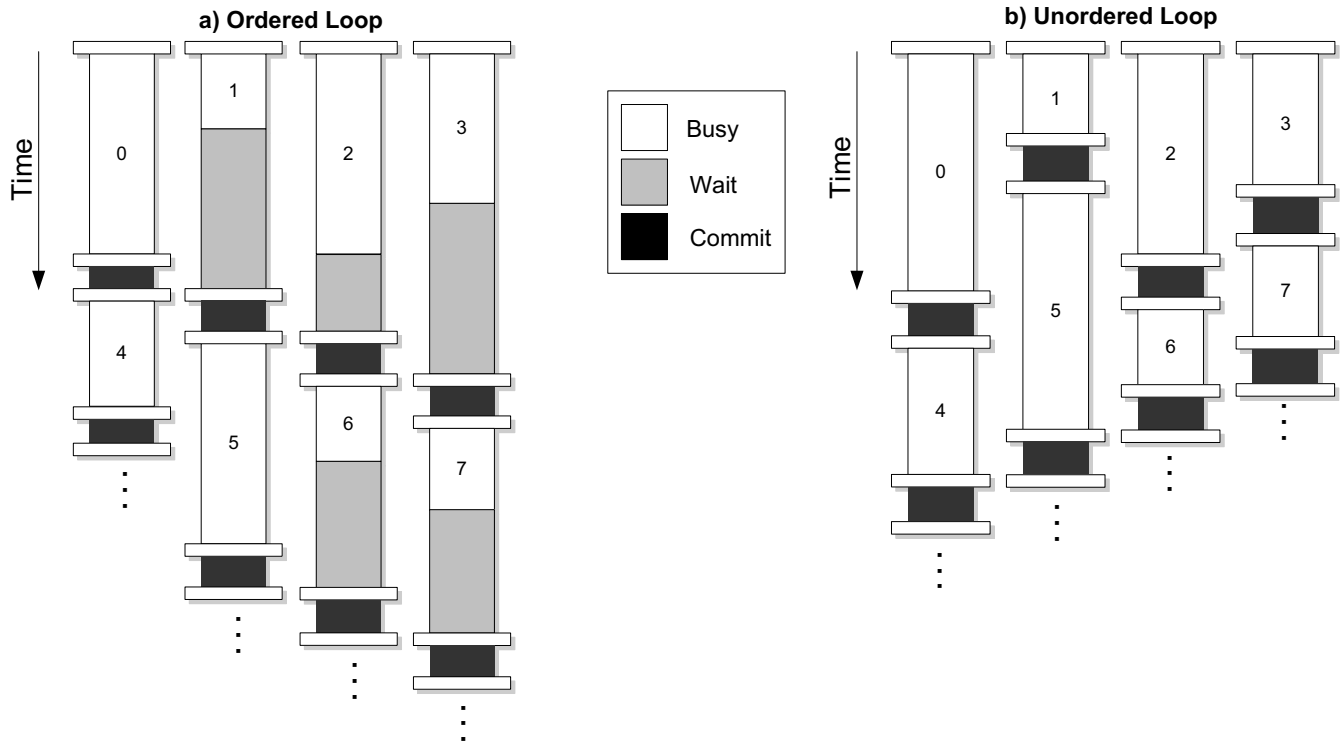

Figure 6. Ordered versus Unordered speculative loop execution timeline. The numbers indicate the loop iteration from sequential execution. In the ordered case on the left, iterations are stalled to preserve sequential semantics, setting up a cascading commit pattern that often reduces stalls in later iterations. In the unordered case on the right, iterations are free to commit in first come, first served order.

```
void run (boolean ordered, int chunk, List list, LoopBody loopBody){
    Thread[]  threads  = new Thread[cpus];
    boolean[] finished = new boolean[list.size()];                    // keep track as iterations finish
    for(int t=0; t<cpus; t++){
        threads[t] = new Thread(new Runnable(){
            public void run(){
                for(int i = t*chunk; i < list.size(); i+= cpus*chunk){
                    atomic {                                          // run each chunk atomically
                        for(int c=0; c<chunk; c++){
                            int iteration = i+c;
                            loopBody.run(list.get(iteration));
                            finished[iteration]=true; }               // mark when iterations complete
                        if (ordered){
                            if (i>0)
                                atomic {
                                    if (!finished[i-1])               // preserve ordering by stalling commit.
                                        for (;;) {                    // when the previous iteration updates
                                            ; }}}}}});                // finished, we restart the inner atomic
        threads[t].start();}
    for(int t=0; t<cpus; t++){
        threads[t].join();}}
```

Figure 7. `Loop.run` implementation. For the unordered case, we simply run and commit iterations as fast as possible. In the ordered case, we have to stall commit until the previous transaction finishes without losing the work of the loop body. This is performed by stalling in a nested `atomic` block at the end of loop iterations.

Code	Description	Source	Input	Lines	Java Synchronization
JikesRVM	Java virtual machine, v2.3.4	Alpern [4]	NA	307,212	131S 14V 15W 4N 11NA
GNU Classpath	Java class libraries, v0.18	FSF [16]	NA	848,165	424S 14V 41W 11N 30NA
SPECjbb2000	Java Business Benchmark	SPEC [44]	736 transactions	30,754	244S 0V 5W 3N 1NA
TestHashtable	multithreaded Map get/put	Harris [20]	4,000 `get` 4,000 `put`	398	4S 0V 2W 0N 2NA
TestWait	circular token passing	Harris [20]	1-16 tokens	367	4S 0V 2W 0N 2NA
TestHistogram	histogram of test scores	Hammond [17]	80,000 scores	331	5S 0V 2W 0N 2NA

Table 2. Summary of runtime and benchmark application sources. *Lines* is the total number of lines of Java source in the original program. *Java Synchronization* lists the uses of various Java constructs: S=`synchronized` V=`volatile` W=`wait` N=`notify` NA=`notifyAll`

5. Evaluation

In this section, we compare the performance of Atomos transactional programming to Java lock-based programming and evaluate the level of effort required to adapt existing programs to Atomos.

5.1 Benchmarks

To evaluate the Atomos programming language, we converted Java benchmarks from using locks to transactions, as summarized in Table 2. SPECjbb2000 provides a widely used Java server benchmark. Several micro-benchmarks are included from recent transactional memory work by Hammond [17] and Harris [20].

The general approach of executing Java parallel programs with transactional memory is to turn `synchronized` statements 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* or follow disciplined locking and nesting conventions for correct synchronization.

Fields with the `volatile` modifier were changed to use `atomic` to force communication. We did not find any places where it was necessary to use an `open` modifier to force a `volatile` to be treated as an open transaction within an `atomic` statement.

Uses of condition variables were then converted from using Java `Object.wait`, `Object.notify`, `Object.notifyAll` method calls to Atomos `watch` and `retry` statements. This step was the most manual, but as shown in Table 2, uses of `wait`, `notify`, `notifyAll` are very infrequent. In most places, converting the code was as straightforward as the producer-consumer example in Figure 1 and did not require rework like the barrier code from Figure 2.

5.2 Environment

We evaluate both Java programs and Atomos programs using the same base Jikes Research Virtual Machine, version 2.3.4, with the following changes. The scheduler was changed to pin threads to processors to avoiding migrating threads during transactions. Methods were compiled before the start of the main program execution. A one gigabyte heap was used which avoided garbage collection. The results focus on benchmark execution time, skipping virtual machine startup. The single-processor version with locks is used as the baseline for calculating speedup.

JikesRVM was run with an execution-driven simulator of a PowerPC CMP system that implements the TCC continuous transaction architecture for evaluating Atomos as well as MESI snoopy cache coherence for evaluating Java locking [36]. The simulator was extended with support for `atomic`, `open`, and violation handlers. 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 and buses is modeled. In particular, the simulator models the contention for the single data port in the L1 caches, which is used both for processor accesses and either commits for transactions or cache-to-cache transfers for MESI. Table 3 presents the main parameters for the simulated CMP architecture. The victim cache is used to hold recently evicted data from the L1 cache.

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 Transfer Latency	3 pipelined cycles 3 pipelined cycles
L2 Cache	8MB, 8-way, 16 cycles hit time
Main Memory	100 cycles latency up to 8 outstanding transfers

Table 3. 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.

5.3 Scaling 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.

Few manual changes were necessary to create an Atomos version of SPECjbb2000. After automatically changing `synchronized` statements to `atomic` statements, the use of condition variables for two different barriers was similar to Figure 2.

We ran using the default configuration that varies the number of threads and warehouses from 1 to 32, although we just measured 736 application-level transactions instead of a fixed amount of wall clock time.

Figure 8 shows the results from SPECjbb2000 [44]. Both the Atomos and Java versions show linear speedup because there is only a 1% chance of inter-warehouse orders causing contention between threads. Since SPECjbb2000 is intended to be an embarrassingly parallel application meant to measure implementation scalability, this is a validation that the Atomos implementation does not add any performance bottlenecks to the original Java implementation for well optimized parallel applications..

5.4 Avoiding Serialization in TestHashtable

Transactions allow optimistic speculation where locks would require pessimistic waiting. Instead of minimizing critical sections, programmers can use large-granularity transactions, which make it easier to reason about correctness without jeopardizing performance in low contention cases.

TestHashtable is a micro-benchmark that compares different `java.util.Map` implementations [20]. 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 number of processors and measure the speedup attained over the single processor case.

When running the Java version, we run the original synchronized `Hashtable`, a `HashMap` synchronized using the `Collec-`

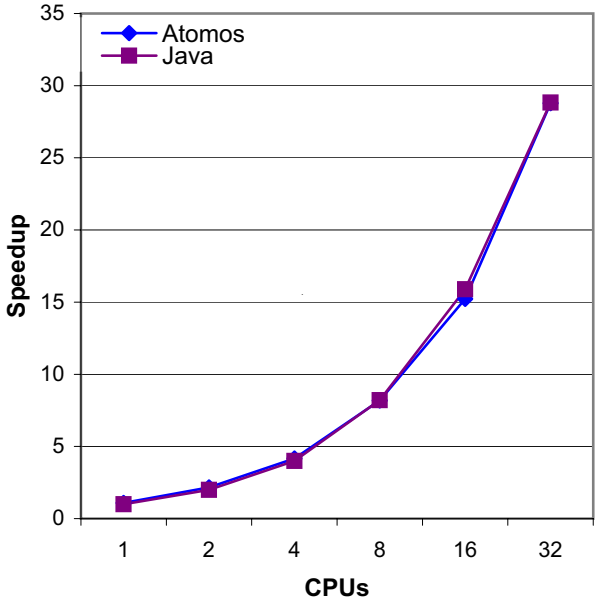

Figure 8. SPECjbb2000 shows linear scaling up to 32 CPUs, demonstrating that Atomos retains the basic scalability of the JikesRVM-based Java implementation from which it was derived.

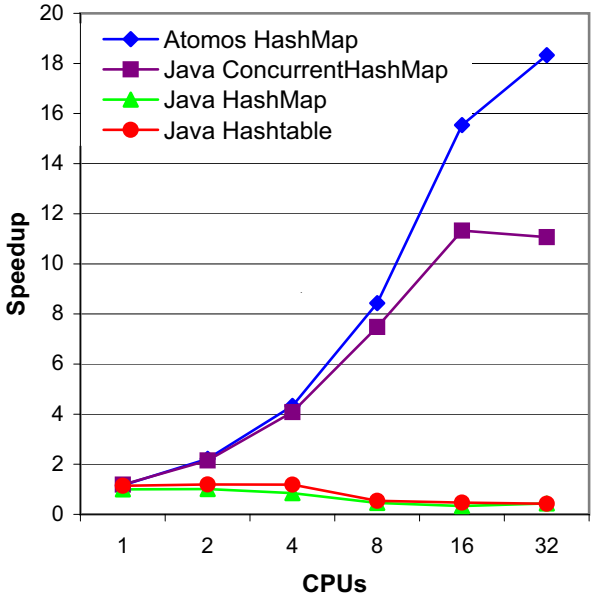

Figure 9. TestHashtable compares the scalability of various `Map` implementations. Coarse-grained locks prevent the scaling of `HashMap` and `HashTable` in Java. `ConcurrentHashMap` does scale to 16 CPUs with Java. However, Atomos is able to achieve scaling through 32 CPUs while using the simpler `HashMap` implementation.

`tions` class's `synchronizedMap` method, and a `ConcurrentHashMap` from `util.concurrent` Release 1.3.4 [32]. `Hashtable` and `HashMap` use a single mutex, while `ConcurrentHashMap` uses fine-grained locking to support concurrent access. When running the Atomos version, we run each `HashMap` operation within one transaction.

Figure 9 shows the results from TestHashtable. The results using Java for `Hashtable` and `HashMap` show the problems of scaling when using a simple critical section on traditional multiprocessors. The `synchronizedMap` version of `HashMap` as well as `Hashtable` actually slows down as more threads are added. While `ConcurrentHashMap` shows that fine-grained locking implementation is scalable up to 16 processors, this implementation, which required significant complexity, suffers a performance degradation at 32 processors. With Atomos, we can use the simple `HashMap`, which had the worst performance in Java, and with a single `atomic` statement, achieve better performance compared to the complex `ConcurrentHashMap` implementation.

5.5 Conditional Waiting in TestWait

One of the contributions of Atomos is fine-grained conditional waiting. Our implementation tries to minimize the number of transactional contexts required to support this and still achieve good performance. We present the results of a micro-benchmark that show that our implementation does not adversely impact the performance of applications that make use of conditional waiting.

`TestWait` is a micro-benchmark that focuses on producer-consumer performance [20]. 32 threads simultaneously operate on 32 shared queues. The queues are logically arranged in a ring. Each thread references two adjacent queues in this ring, treating one as an input queue and one as an output queue. Each thread repeatedly attempts to read a token from its input queue and place it in its output queue. For the queue implementation, we used `BoundedBuffer` from `util.concurrent` for Java and a `TransactionalBoundedBuffer` modified to use `watch` and `retry`. In our experiment we vary the number of *tokens*, not processors, from 1 to 32.

Figure 10 shows the results from TestWait. As the number of tokens increases, both Atomos and Java show similar speedups from 1 to 4 tokens, since both are paying similar costs for suspending and resuming threads. However, as the number of tokens approaches the number of processors something interesting happens. Recall that threads that are in the process of waiting but have not yet discarded their read set can be violated and rescheduled without paying the cost of the thread switch. Up until the point that the read set is discarded, a violation handler on the thread that has entered `retry` can cancel the process and simply restart the transaction without involving the scheduler. At 8 tokens, one quarter of the 32 processors have tokens at a time, so its very likely that even if a processor does not have a token, it might arrive while it is executing `watch` or the the watch request part of `retry`, allowing it to rollback and restart very quickly. At 16 tokens, this scenario becomes even more likely. At 32 tokens, this scenario becomes the norm. In the Java version, the mutual exclusion of the monitor keeps the condition from changing while being tested, meaning that if the condition fails, the thread will now wait, paying the full cost of switching to the wait queue and back to running even if the thread that could satisfy the condition is blocked waiting at the monitor at the time of wait.

5.6 `Loop.run` with TestHistogram

To evaluate the basic usefulness of `Loop.run`, we use the simple histogram example from 4.2, originally presented in Hammond [17]. Random numbers between 0 and 100 are counted in bins. Our version, TestHistogram, also provides a manually parallelized Java version. When running with Java, each bin has a separate lock to prevent concurrent updates. When running with Atomos, each update to a bin is one transaction.

Figure 11 shows the results from TestHistogram. While the Java version does exhibit scalability over the single processor baseline, the minimal amount of computation results in significant overhead

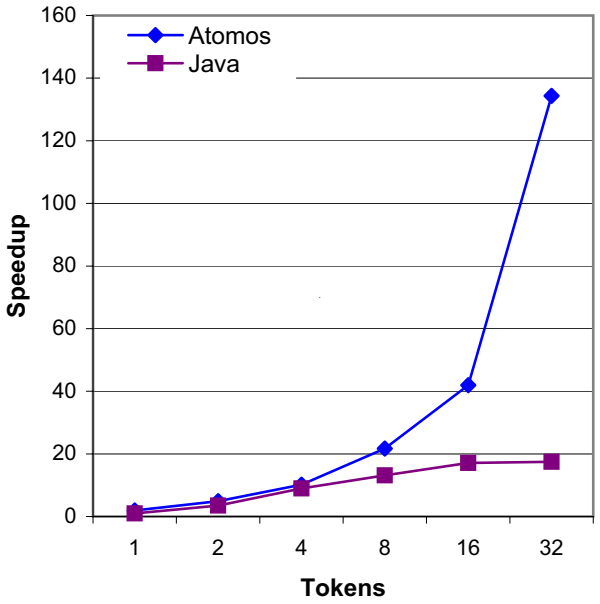

Figure 10. TestWait compares Java and Atomos conditional waiting implementation, through a token passing experiment run in all cases on 32 CPUs. As the number of simultaneously passed tokens increases, both Java and Atomos take less time to complete a fixed number of token passes. The Atomos performance starts to accelerate with 8 tokens being passed on 32 CPUs when Cancel Retry ViolationHandler from Figure 4 frequently prevents the thread waiting code from completing a context switch. When 32 tokens are passed between 32 CPUs, there is almost no chance that the Atomos version will have to perform a context switch.

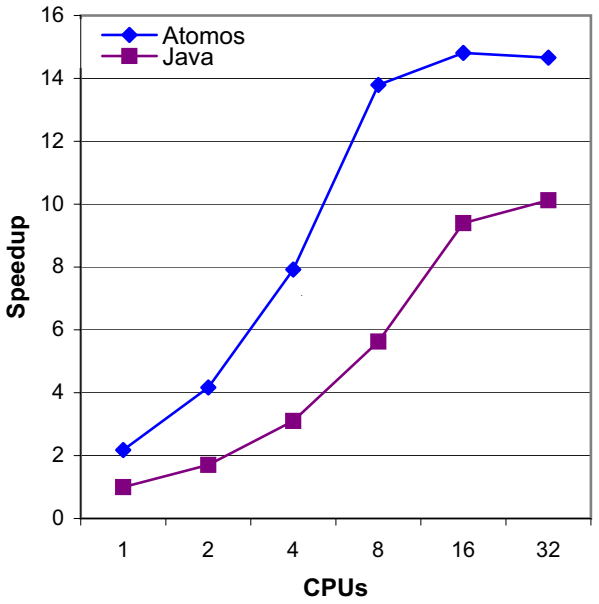

Figure 11. TestHistogram compares a Java locked-based parallel histogram program with a transactional Atomos version. Atomos performance starts out faster on 1 CPU because of the lack of Java lock overhead. With 32 CPUs updating 101 bins, contention becomes an issue for both languages. The Atomos version has violations, causing a slight performance degradation at 32 CPUs. The Java has a knee in the performance curve as lock contention starts to become a limiting factor.

for acquiring and releasing locks, which is shown by the difference in performance between Java and Atomos even in the single processor case. The Atomos version eliminates the overhead of locks and demonstrates scaling to 16 CPUs; transactions allow optimistic speculation where locks caused pessimistic waiting. However, at 32 CPUs, both versions start to suffer from the contention of 32 threads competing to update only 101 bins; Atomos starts to suffer from violation rollbacks and Java spends a higher percentage of time waiting to acquire locks.

6. Conclusions

The Atomos programming language simplifies writing parallel programs utilizing transactional memory. Atomos provides strong atomicity by default, while providing mechanisms to reduce isolation when necessary for features such as loop speculation. Atomos allows programs to specify watch sets for scalable conditional waiting. The Atomos virtual machine implementation uses violation handlers to recover from expected violations without necessarily rolling back. The superior performance of Atomos compared to Java and simplicity of coding demonstrates the value of programming with transactions.

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 and through the MARCO Focus Center for Circuit & System Solutions (C2S2), under contract 2003-CT-888. Brian D. Carlstrom is supported by an Intel Foundation PhD Fellowship.

References

[1] A.-R. Adl-Tabatabai, B. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2006. ACM Press.

[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] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[5] C. S. Ananian, K. Asanović, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)*, pages 316–327, San Franscisco, California, 2005.

[6] C. Blundell, E. C. Lewis, and M. M. K. Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, June 2005.

[7] H.-J. Boehm. Threads cannot be implemented as a library. In

PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pages 261–268, New York, NY, USA, 2005. ACM Press.

[8] *The Broadcom BCM1250 Multiprocessor*. Technical report, Broadcom Corporation, April 2002.

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

[10] B. D. Carlstrom, J. Chung, H. Chafi, A. McDonald, C. Cao Minh, L. Hammond, C. Kozyrakis, and K. Olukotun. Transactional Execution of Java Programs. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*. October 2005.

[11] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, 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 programming systems languages and applications*, pages 519–538, New York, NY, USA, 2005. ACM Press.

[12] Cray. *Chapel Specification*. February 2005.

[13] J. Danaher, I.-T. Lee, and C. Leiserson. The JCilk Language for Multithreaded Computing. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*. October 2005.

[14] J. L. Eppinger, L. B. Mummert, and A. Z. Spector, editors. *Camelot and Avalon: a distributed transaction facility*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1991.

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

[16] Free Software Foundation, *GNU Classpath 0.18*. `http://www.gnu.org/software/classpath/`, 2005.

[17] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (TCC). In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 1–13, New York, NY, USA, October 2004. ACM Press.

[18] 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.

[19] T. Harris. Exceptions and side-effects in atomic blocks. In *2004 PODC Workshop on Concurrency and Synchronization in Java Programs*, July 2004.

[20] 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.

[21] T. Harris, S. Marlow, S. Peyton-Jones, and M. 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, July 2005. ACM Press.

[22] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2006. ACM Press.

[23] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, July 2003. ACM Press.

[24] 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.

[25] International Organization for Standardization. *ISO/IEC 9075-5:1999: Information technology —- Database languages — SQL — Part 5: Host Language Bindings (SQL/Bindings)*. International Organization for Standardization, Geneva, Switzerland, 1999.

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

[27] M. Jordan and M. Atkinson. *Orthogonal Persistence for the Java Platform*. Technical report, Sun Microsystems, October 1999.

[28] 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.

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

[30] T. Knight. An architecture for mostly functional languages. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 105–112, New York, NY, USA, August 1986. ACM Press.

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

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

[33] B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. Program. Lang. Syst.*, 5(3):381–404, 1983.

[34] V. Luchangco and V. Marathe. Transaction Synchronizers. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*. October 2005.

[35] A. McDonald, J. Chung, B. D. Carlstrom, C. Cao Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural Semantics for Practical Transactional Memory. In *Proceedings of the 33rd International Symposium on Computer Architecture*, 2006.

[36] 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 *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 63–74, Washington, DC, USA, September 2005. IEEE Computer Society.

[37] E. Moss and T. Hosking. Nested Transactional Memory: Model and Preliminary Architecture Sketches. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*. October 2005.

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

[39] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 494–505, Washington, DC, USA, June 2005. IEEE Computer Society.

[40] M. F. Ringenburg and D. Grossman. Atomcaml: first-class atomicity via rollback. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 92–104, New York, NY, USA, 2005. ACM Press.

[41] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg. A high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, NY, USA, March 2006. ACM Press.

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

[43] M. Sherman. Architecture of the encina distributed transaction processing family. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 460–463, New York, NY, USA, 1993. ACM Press.

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

[45] 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.

[46] 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.