Atomicity via Source-to-Source Translation

Benjamin Hindman Dan Grossman
University of Washington
b@cs.washington.edu djg@cs.washington.edu

Abstract

We present an implementation and evaluation of atomicity (also known as software transactions) for a dialect of Java. Our implementation is fundamentally different from prior work in three respects: (1) It is entirely a source-to-source translation, producing Java source code that can be compiled by any Java compiler and run on any Java Virtual Machine. (2) It can enforce "strong" atomicity without assuming special hardware or a uniprocessor. (3) The implementation uses locks rather than optimistic concurrency, but it cannot deadlock and requires inter-thread communication only when there is data contention.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Concurrent programming structures

General Terms Languages

Keywords Atomicity, Transactional Memory, Concurrent Programming, Java

1. Introduction

1.1 Atomicity: Definition and Prior Approaches

Multithreaded programs using shared memory, mutual-exclusion locks, and condition variables are notoriously difficult to write correctly. Avoiding races and deadlocks requires cumbersome and error-prone idioms. Yet for an increasing number of applications on an increasing number of platforms, parallelism is important for performance (to exploit multiple processors) and isolation (by running separate tasks with separate threads).

To make shared-memory multithreaded programming easier, many researchers have argued for *atomicity*, also known as *software transactions*. The software-engineering advantages of atomicity are numerous and not the focus of this paper. We note simply that it lets programmers write mutually-exclusive critical sections that access any number of objects, without risking deadlock or relying on other threads to obey a locking protocol.

Atomicity can complement or replace existing synchronization mechanisms with the statement form `atomic { s }` where s is a (nearly arbitrary) statement. Semantically, so-called *strong atomicity* means s must execute *as though* there is no interleaved computation, i.e., no other threads are running. (The implementation, of course, need not actually stop other threads provided it preserves the semantics.) Furthermore, a language should also ensure fair scheduling: long transactions must not starve other threads. Strong atomicity ensures sequential reasoning is sound within a transaction; no other thread can interfere or observe intermediate results.

The less rigorous *weak atomicity* provides a no-interleaving guarantee only among transactions, not between transactions and other computation. To achieve atomicity, programmers (perhaps with the aid of a type system or analysis tool) must use other means to ensure there are no conflicting memory accesses between transactional and non-transactional code.

The conventional wisdom followed in the many existing atomicity implementations (see Section 7) is that a quality multiprocessor implementation requires optimistic concurrency protocols implemented in hardware or low-level software libraries (perhaps with compiler back-end support). Moreover, strong atomicity is generally considered too performance limiting without hardware support since it can require reads and writes not in transactions to synchronize with other threads.

In this work, we take the view that a primary implementation consideration is avoiding unnecessary synchronization (for uncontended memory), and doing so correctly and efficiently requires neither low-level techniques nor optimistic protocols.

1.2 Source-To-Source Translation To Java

Our prototype atomicity implementation (for Java) takes the novel approach of modifying neither a compiler nor a (virtual) machine. Rather, we perform source-to-source translation: We take a program written with `atomic` and produce a regular Java source program. Together with a few classes we wrote in Java (our "runtime system" for atomicity), we can compile this Java code with any compiler and run it on any virtual machine.

As such, we have demonstrated that implementing atomicity can be kept quite separate from other concerns. Our translation occasionally uses particular Java features, but our approach should apply to other high-level languages. That is, we do not "pull any Java tricks" that restrict the applicability of lock-based atomicity or source-to-source translation.

Section 6 informally evaluates the use of source-to-source translation for this type of research study.

1.3 Transactions via Rollback and Locking

Instead of optimistic concurrency protocols, hardware cache coherence, or a restricted thread scheduler, we use automatically-managed locks to manage contention and have transactions roll-back memory if they hold the lock for data needed by another thread.

A basic atomicity implementation using locks in this way is surprisingly simple:

1. Let every object "lock itself" in the sense that every object has a field holding the `Thread` object that currently "owns" (i.e., may access) its fields. Let `null` indicate the "lock is available." Assume access to these "current-holder" fields is synchronized

MSPC'06 October 22, 2006, San Jose, CA, USA

via primitive means (e.g., compare-and-swap operations). Note these "locks" are *not* Java's built-in locks.

2. Require all code to acquire an object's lock (i.e., set the lock-field to the current thread) before reading or writing any other field of the object. (For weak-atomicity, require the lock only in a transaction.)
3. If a necessary lock is unavailable, "inform" the current holder it must be "released soon."
4. Require atomic-block execution to log all writes by storing the object-field written to and the overwritten value in a thread-local data structure.
5. Require threads to "poll" for locks to release. If in an atomic block, "rollback" (abort) the transaction before releasing a lock.

Parts (1), (2), and (5) ensure no execution of an atomic block reads inconsistent values or reveals inconsistent writes. Parts (3), (4), and (5) prevent deadlock because no thread holds onto a requested lock forever. The ability to rollback is crucial for preventing starvation. Exponential back-off can avoid livelock (any contention-management scheme would suffice). Note we could change "lock granularity" (making it finer by locking fields separately or coarser by locking multiple objects together) without sacrificing correctness. Similarly, it is correct to release a lock at any point outside an atomic block (reacquiring it as necessary).

1.4 Contributions

This work presents a lock-based source-level implementation of atomicity (with strong or weak semantics). The most novel contributions are:

- A prototype demonstrating that atomicity does not require special virtual-machine or hardware support. This indicates that even with such support (which could improve performance or simplify parts of the implementation), transactions can be largely decoupled from the rest of the memory system.
- A correct design for software-transactional memory that uses a lock-based approach and requires inter-thread communication only when there is data contention. Strong atomicity does not treat every memory access as a "little transaction."
- Preliminary experimental results showing reasonable performance for weak atomicity, much worse performance for naive strong atomicity, but some hope that simple whole-program optimization can recover much of the difference.

2. Basic Approach

This section describes our implementation assuming the entire program consists of user-defined classes containing constructors, instance methods, and field accesses. Our implementation supports other language features such as arrays; native code; the standard library; static fields, methods, and initializers; exceptions; etc.

Our implementation, built with the Polyglot extensible compiler [25], takes a Java program with `atomic` statements and produces a Java program that can be compiled and run by any Java implementation. The basic approach in this section obeys separate compilation: we can compile each class as independently as a Java compiler.

Figures 1 and 2 contain examples demonstrating how the features of our translation fit together. For simplicity, they elide access modifiers (e.g., `public`) and package names. The examples show substantial code bloat because they do little actual computation and do not incorporate optimizations (Section 4). Also, a just-in-time compiler can eliminate much of the introduced code.

2.1 Acquiring and Releasing Locks

We add one field to every object and several fields to each thread. As a source-to-source translation, we cannot change the definition of `Object` or `Thread`, but we can create subclasses `AObject` and `AThread` respectively. Any class extending `Object` or `Thread` has its `extends` clause changed or added appropriately. In this way, we can add fields to (almost) every user-defined class.

`AObject` has one field, `currentHolder`, of type `AThread`. This field indicates which thread may currently access the object's fields; `null` means no thread may. This field is the conceptual lock for the object, but it clearly is not a Java lock. The constructors for `AObject` initialize `currentHolder` to the current-thread; this policy choice does not affect correctness.

Every field access is preceded by checking that the field's object's `currentHolder` field is the current-thread. If not, we must first "acquire the lock." If `currentHolder` is `null` we write the current-thread in it and proceed (synchronization of `currentHolder` is discussed below). Else we add the object to the current-holder's "locks to release" set (the `AThread` instance field `lks_to_release` holds this set) and we later retry acquiring the lock. In summary, *ignoring synchronization*, the algorithm to acquire the lock for an object is roughly this method of `AObject`:

```
0. void acq_mylk() {
1.   AThread me = (AThread)Thread.currentThread();
2.   if(currentHolder != me)
3.     while(true) {
4.       if(currentHolder == null) {
5.         currentHolder = me;
6.         break;
7.       }
8.       currentHolder.lks_to_release.add(this);
9.       AThread.check_release();
10.      Thread.yield();
11.    }
12. }
```

A field access `e.f` where `f` is defined in class `C` is rewritten as `C.get_f(e)` where `get_f`[1] is a static method the translation of `C` generates. It acquires the lock then returns the field's contents. Similarly, `x.f=v` becomes `C.set_f(x,v)`.

Every thread *polls* its `lks_to_release` field. To ensure polling, we add a call to `AThread.check_release` to the start of each method body and loop.[2] The thread releases a lock by setting `currentHolder` to `null`. When in an atomic block, we roll back (see Section 2.3) before releasing any locks.

A key feature of our system is that threads do not release a lock until another thread requests it. This policy avoids unnecessary synchronization (see below), but does require polling throughout the program.

2.2 Synchronization of Locks

The above scheme ensures field accesses use locks and locks always become available eventually. Rolling back a transaction before releasing locks ensures atomicity. However, the actual implementation is more complicated: The `currentHolder` and `lks_to_release` fields are thread-shared so accessing them requires synchronization.

For `lks_to_release`, each `AThread` has its own monitor that every thread acquires before accessing `lks_to_release`. Deadlock is impossible because no thread acquires another lock

[1] Throughout this paper, we ignore name-mangling. For example, we actually use `__aj_get_f` in case there is a user-defined `get_f` method.

[2] We omit checks where it is obviously sound. For example, a method body containing no calls needs no check on entry.

```
class A {
 int x;
 A() { x=10; }
}
```

```
class A extends AObject {
 int x;
 static int get_x(A o) {
   o.acq_mylk(); return o.x;
 }
 static int set_x(A o, int v) {
    o.acq_mylk(); return o.x = v;
 }
 static int set_atomic_x(A o, int v) {
   o.acq_mylk();
   ((AThread)Thread.currentThread())
      .log(o,undo_x,o.x);
   return o.x = v;
 }
 static UndoInteger undo_x = new UndoInteger() {
   void undo(Object o,long v){((A)o).x = (int)v;}
 };
 A() {
  super();
  set_x(this,10);
 }
 A(DummyArg x) {
   super(x);
   set_atomic_x(this, 10);
 }
}
```

Figure 1. A simple class before (above) and after (below) translation.

while holding one of these locks. This synchronization would be a bottleneck if we actually incurred it on every loop iteration and method call. Instead, `AThread.check_release` just decrements a thread-local counter and checks `lks_to_release` only when the counter reaches zero. We then reset the counter to a constant `POLL_FREQUENCY`. This constant trades off responsiveness for communication. Section 5 measures the effect of varying it.

For `currentHolder`, every object has a Java monitor that is held for lines 4–8 of the lock-acquire code above and when releasing the lock. We could use the object itself (i.e., synchronize on `this` in the code), but if user code also synchronizes on the object, we could introduce deadlock. So we conservatively use a separate array of monitors for controlling access to `currentHolder` fields, using `System.identityHashcode` on an object to index into this array. (An alternative is to use `java.atomic.compareAndSet` for accessing `currentHolder` fields.)

Most importantly, the common case of a thread accessing a field of an object for which it already holds the lock does *not* require synchronization: The condition on line 2 of the lock-acquire code is true, so we neither hash nor acquire a monitor. In particular, thread-local data will never incur Java synchronization and this does not require any static analysis. Rather, we incur only the overhead of checking that `x.currentHolder==me` always holds.

This algorithm is correct under Java's Memory Model [20], i.e., we are *not* assuming sequential consistency: Although the read on line 2 above is not synchronized, the condition can be true only if the same thread has already performed a synchronized write to the `currentHolder` field. That is, we exploit that threads only ever write their own thread-id into `currentHolder` fields.

```
class B extends C {
 A a;
 void f() { a = new A(); }
 int g() { return a.x;  }
 B(int i) {
  super(); f();
  atomic { ++i; f(); a.x = a.x+i; }
 }
}
```

```
class B extends C {
 A a;
 static A get_a(B o) {
   o.acq_mylk(); return o.a;
 }
 static A set_a(B o, A v) {
   o.acq_mylk(); return o.a = v;
 }
 static A set_atomic_a(B o, A v) {
   o.acq_mylk();
   ((AThread)Thread.currentThread())
      .log(o,undo_a,o.a);
   return o.a = v;
 }
 static UndoObject undo_a = new UndoObject() {
   void undo(Object o,Object v){((B)o).a = (A)v;}
 };
 void f() {
  ((AThread)Thread.currentThread()).check_release();
  set_a(this, new A());
 }
 void __aj_f(){
  ((AThread)Thread.currentThread()).check_release();
  set_atomic_a(this,new A(DummyArg.single));
 }
 int g() { return A.get_x(get_a(this)); }
 int __aj_g() { return A.get_x(get_a(this)); }
 B(int i) {
  super(); f();
  AThread me = (AThread)Thread.currentThread();
  int __i = i;
  boolean done = false;
  me.start_atomic();
  while(!done) {
    done = true;
    try {
      ++i; __aj_f();
      A.set_atomic_x(get_a(this),
                     A.get_x(get_a(this))+i);
    } catch (RollBack e) {
      done = false;
      i = __i;
      me.sleep_after_rollback();
    } finally { if(done) me.end_atomic(); }
  }
 }
 B(DummyArg x, int i) {
  super(x); __aj_f();
  ++i; __aj_f();
  A.set_atomic_x(get_a(this),
                 A.get_x(get_a(this))+i);
 }
}
```

Figure 2. A class before (above) and after (below) translation, which uses the class in Figure 1.

2.3 Logging and Rollback

Correctness demands an atomic block not release locks for any objects it has accessed. To avoid deadlock, it suffices to release locks when requested, but to first undo all assignments to memory. This section discusses how we undo field assignments; Section 2.4 discusses local variables.

While executing a transaction, assignment to field `f` of object `x` of type `C` calls `C.set_atomic_f(x,v)` instead of `C.set_f(x,v)`. The former is like the latter except it also calls the current thread's `log` method. *Conceptually*, it passes `x` (the "container" object), `f` (the "field name"), and `x.f` before the assignment (the "old value"). A thread-local data structure holds log entries in a *conceptual* stack. To rollback, one pops elements off the stack, assigning each old value back to the field of the container object.

The logging implementation realizes this concept via some cleverness to minimize per-assignment-in-transaction cost and obey Java's type system. (The latter would be no concern were the log implemented in the virtual machine.) Relevant issues are:

- Field names are not first-class; we cannot pass them. (We could use reflection but have chosen not to.)
- Container objects could be any subtype of `AObject`.
- Previous values could be any type.
- Logging field-assignments should not cause memory allocation.

To begin, each `AThread` has four fields that together encode the log for fields whose types are subtypes of `Object`:

```
int           obj_log_index;
Object[]      obj_log_containers;
UndoObject[]  obj_log_undoers;
Object[]      obj_log_oldvalues;
```

The i^{th} log-entry is in the i^{th} index of the three arrays and `obj_log_index` holds the current log size. If the arrays fill, we double their size (see Section 2.5 for avoiding this). Hence we typically do no memory allocation, but never do more than $O(log\ n)$ allocations for a transaction that does n field assignments.

The `obj_log_containers` and `obj_log_oldvalues` arrays hold the container-objects and previous-values. More interestingly, `obj_log_undoers` holds call-back objects that the roll-back code uses. The `UndoObject` class is just:

```
abstract public class UndoObject {
  abstract public void undo(Object container,
                            Object old);
}
```

and the roll-back code is just:

```
  for(int i = obj_log_index; i >= 0; i--)
    obj_log_undoers[i].undo(obj_log_containers[i],
                            obj_log_oldvalues[i]);
```

It just remains for the caller to `log` to pass an appropriate instance of `UndoObject`. For example, if assigning to field `f` of type `D` of an instance of class `C`, the body of `undo` is the assignment `((C)container).f = (D)old;`. These downcasts execute only if we rollback a transaction; the call to `log` upcasts the container and previous-value to `Object`.

For every field declaration, we have one (anonymous) subclass for its undoer held in a static field of the field's class. Continuing our example, class `C` would have this declaration:

```
private static undo_f = new UndoObject () {
  public void undo(Object container, Object old) {
    ((C)container).f = (D)old;
  }};
```

Hence we have one new class for every field, but no per-instance or per-assignment memory allocation. Moreover, two log entries are for the same field of the same object if and only if the container and undoer are the same object (i.e., pointer-equal).

For fields with primitive types, the log described so far would incur the overhead of boxing the old-values. Instead, we use two other logs, one for integral types (the old-values array has type `long[]`) and one for `float` and `double` (the old-values array has type `double[]`). We use different abstract "undo" classes, which have `undo` methods that may perform narrowing conversions.

When a transaction commits, it is not necessary to empty the logs (the next transaction can just reset the indices to 0). However, leaving objects in the container and old-value arrays can cause space leaks. Section 5 reports the cost of "nulling-out" these array entries to avoid potential leaks. (Another option would be to reallocate the arrays for every transaction.) Note we cannot use weak-arrays because during a transaction a log could hold the last reference to an object that will be live if we rollback.

2.4 Translation of atomic

We create two versions of each method `m` in a source program: We call `m` while not executing a transaction (so a write to field `x` in `m` uses `set_x`) and `__aj_m` while executing a transaction (so a write uses `set_atomic_x`). Similarly, calls in `m` are to other "non-atomic" methods and calls in `__aj_m` are to "atomic methods." In this way, we know at each program point whether we are in a transaction or not, so there is no run-time overhead for deciding if we need to log. (Prior work [19] and our experience indicate that whole-program analysis can remove most of this code duplication because most methods are never used within a transaction.)

The obvious exception to the description above is that the non-atomic version of a method containing `atomic{`s`}` uses atomic methods in s. (In the atomic version we omit the `atomic` to implement a "flattened semantics" for nested atomic-blocks.) Translating `atomic{`s`}` also involves logging local variables, catching an exception indicating a rollback occurred, and looping until the transaction succeeds. For example, `atomic{ m(); ++i; }` becomes:

```
AThread me = (AThread)Thread.currentThread();
int __i = i; // log original value
boolean done = false; // loop guard
me.start_atomic(); // initialize logs
while(!done) {
  done = true;
  try { __aj_m(); ++i; }
  catch (RollBack e) {
    // locks were released and field-writes undone
    done = false;
    i = __i; // rollback local
    me.sleep_after_rollback(); // back-off
  } finally { if(done) me.end_atomic(); }
}
```

Methods like `__aj_m` do not log local variables; they are simply popped off the stack if a rollback occurs. Similarly, writes to fields of `this` in the atomic version of constructors need not be logged since the new object will be unreachable (garbage) after rollback.

However, to create atomic and non-atomic versions of each constructor we cannot follow our instance-method approach of giving the atomic version a different name. Therefore, we have atomic constructors take a dummy argument of an otherwise unused type and callers pass a (globally-shared) object of this type.

2.5 Details

We now discuss some less critical details relevant to the translation.

In-place update Translating `f().x += 3` to `C_set_x(f(),C_get_x(f())+3)` is incorrect because the latter calls `f` twice. For such expressions (including `(f().x)++`) we generate a helper method to do the update and pass `f()` to it.

Log-duplicates If the same field of the same object is set multiple times in the same transaction, only the first needs logging. As in prior work [27], we assume most atomic blocks have few writes, so it is faster not to detect duplicates. To avoid pathological situations, we detect and remove duplicates once the log arrays fill using a simple $O(n^2)$ approach: Two entries are duplicates if the containers and undoers are the same object. If after duplicate removal the arrays are still over half full, we create new arrays twice as long.

Thread Objects Threads must be a subclass of `Thread`, so Java's single inheritance means threads cannot be a subclass of `AObject`. Therefore, we also have a `currentHolder` field and `acq_mylk` method in `AThread`. We also seem to assume every `Thread` is an `AThread` lest `(AThread)Thread.currentThread()` fail. We have additional support for threads the virtual machine creates.

Final fields We read `final` fields directly; there is no need for getters, setters, and undoers. Similarly, for `final` local variables accessed within an atomic-block, we need not and must not do rollback. What remains is initializing a `final` field in a constructor (or instance initializer). In the atomic-constructor (the one taking a `DummyArg`) we can just do the initialization; if the transaction aborts the new object will be garbage anyway. In the non-atomic constructor, we statically disallow a final-field initialization to be lexically within an atomic block since rollback would not be possible.

Field and instance initializers Given a field initialization like `T x=f();` we cannot generate atomic and non-atomic versions like we do for methods and constructors. So here and only here we incur a run-time test to determine if the running thread is in a transaction.

Static getter/setter methods Why have we made the getter and setter methods `static`, with calls like `get_x(e)`, rather than calls like `e.get_x()`? Because fields and static methods have the same lookup rules, but instance methods use dynamic dispatch, which would be incorrect if a subclass reuses a field name.

Lock Stealing So far, we assume a thread holding access to an object will eventually release access when another thread requests it. However, threads that have terminated or are blocked (due to legacy synchronization or waiting for a lock to be released) will not do so. We did not pursue having threads release all objects they hold on termination because we do not want the space and time overheads of maintaining the set of objects held. (At the very least we would need to use weak references for this set.)

Instead, the thread requesting an object checks if the holding thread is blocked (or no longer alive). If so, it is safe to set the object's `currentHolder` to `null` "on behalf" of the dead thread, using appropriate synchronization.[3]

Catch/Finally Catch and finally blocks in user code must not "intercept" a `Rollback` exception. As necessary, we rewrite such blocks so that during rollback they immediately rethrow `Rollback`.

2.6 Summary

To review our source-to-source translation:

- For every field, there are 3 new methods (getters and setters) and one new field holding an anonymous inner class (the undoer).
- For every method and constructor, there are two versions (atomic and non-atomic).
- Every object has a `currentHolder` field. A global array of monitors synchronizes access to such fields.
- Every loop and method has a check to see if the running thread must release ownership of an object.
- Every `Thread` is an `AThread` holding thread-local data such as the logs for rollback.

[3] This synchronization requires modifying a flag before and after a thread blocks itself, but this is straightforward with source-to-source translation.

3. Other Language Features

We need to "scale up" our basic approach to support interaction with other language features in modern languages like Java. Some features we can support fully (arrays, static fields, other concurrency primitives, exceptions, and inner classes). For other features we have had to limit (but not forbid) their use (reflection, native code, finalizers) or relax Java's semantics (class loading). In general, we can identify three causes of such limitations:

1. Source-to-source translation: We add fields, methods, calls, etc. to programs. If these additions are visible (e.g., via reflection), then translation could change a program's meaning. In principle one could enrich the translation to hide the changes (e.g., by rewriting all uses of reflection), but we have not done so.
2. Irreversible virtual-machine actions: We must be able to abort a transaction and rollback to an equivalent pre-transaction state. But certain actions in Java are both visible and not undoable (e.g., loading a class with static initializers or creating an object with a finalizer). Virtual-machine support would avoid these limitations, but the practical impact is probably small.
3. Unavailable code: All code must obey our translation's invariants, but we cannot change native code nor certain library classes whose definition is assumed by the virtual machine.

We have chosen to make these limitations cause run-time exceptions (e.g., if a native call occurs in a transaction) rather than to relax our atomicity guarantees, but this is an easily changed policy.

Space constraints preclude discussing how we support these extensions, but an extended version [18] includes the details.

4. Optimizations

Our source-to-source translation introduces several sources of overhead that we could hope to ameliorate with compile-time analysis on the translated source program. Our experiments (Section 5) indicate that most overheads (e.g., polling, logging, and adding space to every object) are relatively small, so we consider the one that is not: The getters and setters for field (and array) accesses amount to read- and write-barriers on heap accesses.

Figure 3 summarizes the barrier overheads without optimization: Under strong atomicity, every access requires "owning" (i.e., having the `currentHolder` be the running thread) the object. Under weak atomicity, only accesses within transactions require "owning" the object; other accesses can read/write the field directly. As discussed in Section 7, much prior work provides only weak atomicity. Doing so produces faster multithreaded code, and in the limit case that a program does not use transactions, weak atomicity suffers no read or write barriers.

A variety of compile-time analyses could safely remove barriers from non-transactional code while preserving strong atomicity, such as thread-escape analysis for establishing that an object is thread-local. We implemented a novel analysis that is complementary to escape-analysis: We can remove a barrier if we can prove that the accessed object could never be accessed within a transaction. Although pointer-analysis would improve precision, we can remove many barriers with only class-based information. After all, if no atomic block ever accesses some field `f`, then no access of field

	weak atomicity	strong atomicity
non-atomic read	none	own
non-atomic write	none	own
atomic read	own	own
atomic write	own+log	own+log

Figure 3. What reads and writes do under different semantics: "own" means get exclusive access; "log" means log the old value.

`f` needs a barrier. In the limit, a program without atomic blocks will be optimized back to exactly the weak-atomicity implementation.

Therefore, our implementation supports "optimized strong atomicity" by using a linear-time whole-program analysis to remove barriers on accesses that cannot conflict with an access within a transaction. Details of the analysis are available [18].

5. Experiments

We view our current prototype as a proof-of-concept that one can implement atomicity for a modern programming language on top of existing hardware and virtual machines. Some performance parameters and large parts of the design space remain unexplored (see Section 8). Nonetheless, we have run our translator on small benchmarks to evaluate overall performance. We conclude that our approach is sometimes but not always competitive with lock-based code, and it provides a reasonable platform for evaluating ongoing research.

Section 5.1 describes the benchmarks and platforms. Section 5.2 presents overall performance results. Section 5.3 presents additional results from modifying parameters such as threads' polling frequency. Section 5.4 summarizes our results.

5.1 Benchmarks and Platforms

We have investigated four small programs. For each, we changed uses of `synchronized` to uses of `atomic` and manually verified that the programs are correct with either strong or weak atomicity.

- `tsp` solves a traveling salesperson problem. It has been used in previous concurrency studies [31, 8]. Threads share partially completed work and the best-answer-so-far via shared memory, but there is parallelism as they search independently. All data is pre-allocated (after the threads are spawned there are no uses of `new`). In the original Java, locking is coarse: all thread-shared data is immutable or guarded by one of two locks, and nontrivial work is done while holding locks. The original program also has benign data races: Code reads the "shortest tour found so far" without synchronization; this is correct because the value only decreases, so stale values lead only to useless work.
- `crypt` is an embarrassingly parallel program in the JavaGrande suite (`www.epcc.ed.ac.uk/javagrande/`) that does not need synchronization (threads operate over disjoint data). Therefore it is useful for measuring the slowdown of our translation for sequential code and the cost of unnecessary barriers in our unoptimized translation.
- `synchBench` is a small benchmark in the JavaGrande suite originally designed to measure the cost of Java's `synchronized` construct. We can similarly measure the cost of heavily contended atomic blocks where the body of the `atomic` does very little work (essentially increment a thread-shared counter).
- `hashtable` is our implementation of a benchmark described in previous work [13] in which parallel threads access a shared hashtable with a mix of insert and lookup operations (16% inserts). We keep the table sparsely populated enough that it is never resized. All threads share a hashtable-object which has an array for which each operation accesses an index of the array. The Java version uses one lock for the whole table; we have not had time to experiment with a lock-based concurrent hashtable.

All experiments used the Java HotSpot VM and Runtime Environment (build 1.5.0_06-b06) with the `-server` option. This option favors long-running programs, so we "warmed up" by running each program until we saw consistent timing data and then took the average of twenty runs. This methodology has two caveats:

- Without "warm up," all data had larger variance (the lock-based more than the atomic code), and the slowdown for atomic was much lower. That is, "warmed up" results are worse for our translation.
- While most runs have times near the average, occasionally runs take twice as long or longer. We believe unfortunate thread preemptions is to blame. These outliers exist for lock-based code and atomic code but are more common for atomic code.

We ran experiments on three machines, all running Linux 2.6.12. Our uniprocessor is a 2.8GHz Intel Pentium 4 with a 512Kb cache and 1GB RAM. Our two-processor machine has 2 Intel Xeon 3.22GHz processors with 2MB caches and 3GB RAM. Our eight-processor machine (which we use for most of our results) is a Dell Poweredge Server with 8 Intel Xeon 3.16GHz processors with 1 MB caches and 8GB RAM.

5.2 Overall Performance

We can measure the slowdown of the atomic versions of the benchmarks relative to the original Java programs. The latter are compiled directly by `javac`, i.e., they are not translated by us. For the atomic versions, we consider three settings: (1) Strong atomicity without optimization, (2) Strong atomicity with optimization, and (3) Weak atomicity. With optimization, our translator takes the whole program and passes specialized Java files to `javac`.

Figure 4 shows results for our benchmark programs for each semantics, various numbers of threads, and various machines. Figure 5 shows parallel speedups for `tsp` and `crypt`. The other benchmarks do not parallelize (even the Java versions), so synchronization overhead causes significant slow down with more processors. The slowdown for atomic is actually less than for locks (see [18]).

The `tsp` program shows significant slowdown compared to lock-based code, even with weak atomicity. This application has larger atomic blocks than other benchmarks, and we observed that rollbacks are not uncommon (on the order of tens of rollbacks per second on the eight-processor machine). We also believe the slowdown results from threads not releasing ownership of objects until another thread requests them, which is a bad match for the work-sharing style of the application.

Nonetheless, `tsp` shows our optimization has value: We recover about half the performance gap between strong and weak atomicity even though the optimization still cannot allow the benign data races. Unfortunately, the performance of strong atomicity (with or without optimization) does not scale with the number of processors for this benchmark. The weak-atomicity version shows some speed-up with the number of processors, though the Java version shows more (and neither is close to linear). We conclude that while removing barriers significantly speeds up sequential execution, we do not remove enough to achieve much parallelism for `tsp`, a fairly complicated benchmark with large atomic sections.

`crypt` has no synchronization (locks in the Java version or `atomic` in our version). Hence this is the ideal case for our optimization: It removes all barriers (so optimized and weak atomicity are identical) whereas the unoptimized version is essentially sequential because all threads contend for the same arrays. (See the

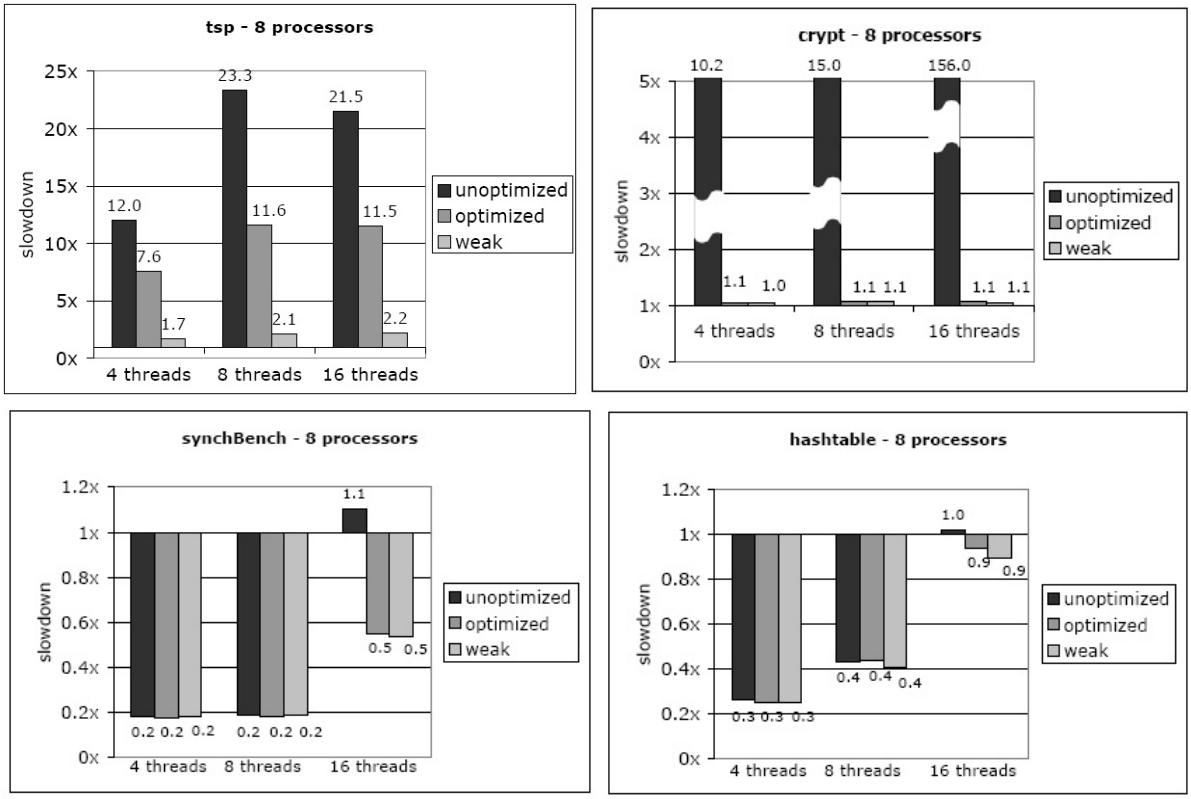

Figure 4. Benchmark results with 8-processors for various implementations and thread-counts. Slowdown is relative to the locking versions.

discussion for `hashtable` for how to avoid this.) Moreover, the overhead after removing barriers (polling for locks to release, space increases, etc.) is minimal; we run only 10% slower than the Java version and preserve parallel speedup (super-linear on 2 processors and almost 5x on 8 processors).

`synchBench` is designed to measure the cost of acquiring and releasing locks. All threads contend for the same data, so neither the lock nor the atomic version have parallelism. The Java program synchronizes on every iteration of every thread's inner-loop whereas our program synchronizes less often due to the polling frequency. Therefore, we run several times *faster* (slowdown of 0.2x is speedup of 5x). However, while the data when the thread-count does not exceed the processor-count is reliable, with 16 threads and 8 processors, run times for any semantics vary by several factors.

`hashtable` also exhibits no parallelism for any version, but the work done in critical sections is a larger than for `synchBench`. There are two reasons the atomic version cannot exploit parallelism using our implementation. First, all hashtable operations use the same hashtable object. Second, all hashtable operations use the same array contained in the hashtable object. The hashtable object is immutable, so a read-only analysis or reader-writer locks for `currentHolder` (i.e., allowing concurrent reads) would fix the first problem. For the array, our locking is too coarse; it is important to allow concurrent access to disjoint indices [2].

5.3 Sensitivity to Parameters

The results presented so far incorporate some tuning of implementation parameters. We now demonstrate that these parameters do involve trade-offs but that performance does not require "getting them exactly right."

Polling Frequency Recall that on every loop and non-leaf method call, a thread calls `check_release` but this method usually only decrements a thread-local counter. We can vary how often it actually checks for locks to release. If too low, threads wait too long for other threads. If too high, too much time is spent on synchronized access to thread-shared data. Results in the previous section used a polling frequency of 1024. As Figure 7 shows, for `tsp` (which has some parallelism and some contention), the best polling frequency is neither too small nor too large. For the other benchmarks, polling frequency is not important. In particular, for `crypt` (which has no inter-thread communication), the cost of checking for locks to release is extremely low because the virtual machine special-cases synchronizing on a monitor that is never held by another thread.

Back-Off Policy We can also adjust the back-off policy for when a thread aborts a transaction. To make livelock unlikely, we use exponential backoff, i.e., wait time $b * c^n$ where n is how many times a transaction has aborted. We can adjust b and c, though the HotSpot VM does not support sleeping less than one millisecond.[4]

Results in the previous section used $b = 1ms$ and $c = 1.1$. For `tsp` these small values produced the best results; significantly larger values could lead to additional slowdown by about a factor of two. The other benchmarks rollback too rarely for the exact parameters to matter except for very large values (e.g., $b = 100ms$) a single unfortunate rollback can produce very bad performance.

We also noticed that while a thread in `tsp` sleeps after aborting a transaction, it is common for other threads to "steal" locks it owns (see Section 2.5). The reason is this sequence of events when two threads are executing code that uses many of the same objects:

[4] The virtual machine implements `Thread`'s `sleep(long millis,long nanos)` method by rounding to the nearest number of milliseconds.

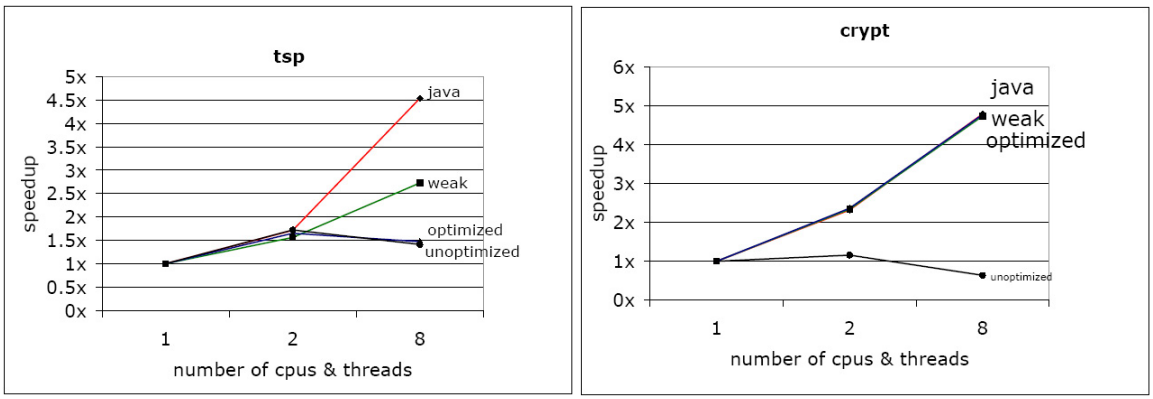

Figure 5. Parallel speedup: Results for 2 threads on a 2-processor machine and 8 threads on an 8-processor machine, relative to 1 thread on a 1-processor machine (results for 1-processor are normalized to 1).

Benchmark (thread-count)	default	synchronize on `this`	no current-thread sharing	`null`-out logs
tsp (4)	0.72s	0.69s (0.96x)	0.98s (1.36x)	0.73s (1.02x)
tsp (8)	0.71s	0.73s (1.02x)	0.95s (1.33x)	0.74s (1.04x)
crypt (4)	2.94s	2.93s (0.99x)	2.89s (0.98x)	2.94s (1.00x)
crypt (8)	2.80s	2.76s (0.98x)	2.75s (0.98x)	2.78s (0.99x)
synchBench (4)	1000K	953K (1.05x)	903K (1.11x)	784K (1.28x)
synchBench (8)	499K	503K (.99x)	419K (1.19 x)	353K (1.41x)

Figure 6. Effect of changing current-holder synchronization, current-thread sharing, and log destruction on our 8-processor machine. Times for `tsp` and `crypt` are absolute running times in seconds (low is good) with slowdown relative to "default" in parentheses. Times for `syncBench` are "operations per second" (high is good), with slowdown relative to "default" in parentheses. "Default" is the configuration used for other experiments: optimized strong atomicity, separate monitors for current-holder instead of `this`, with current-thread sharing, without writing `null` in log entries when atomic commits. Subsequent columns change one of these policies at a time.

- Thread 1 blocks to wait for object 1, owned by thread 2.
- Thread 2 rolls back a transaction, releases object 1, and sleeps.
- Thread 1 acquires object 1 and then "steals" other objects thread 2 had accessed and still owns.

If objects with this sort of locality shared a `currentHolder` (i.e., ownership had coarser granularity), performance would improve.

Current-Holder Synchronization Using an array of monitors for synchronizing access to `currentHolder` fields requires a level of indirection and computing hashcodes. This array is necessary only to avoid deadlock if the program still has `synchronized` statements. Our benchmarks do not, so we can synchronize directly on the object whose `currentHolder` field we are accessing. The third column in Figure 6 shows the slowdown (i.e., speedup when numbers are less than 1) with this change. The very small improvement suggests that the level of indirection and hashcode computation is well-optimized by the underlying virtual-machine.

Current-Thread Sharing Results presented so far include a simple translation-time optimization: Methods that use the current-thread more than once store the result of `Thread.currentThread()` in a local variable. To make this more useful, we also use versions of the getter methods that take the current-thread as an argument. The fourth column in Figure 6 shows that disabling this common-subexpression elimination slows down `tsp` and `synchBench` 11–36%, which suggests we should implement even more aggressive sharing.

Log Destruction When an atomic block completes, we just set the log indices back to 0 to prepare for the next `atomic` block.

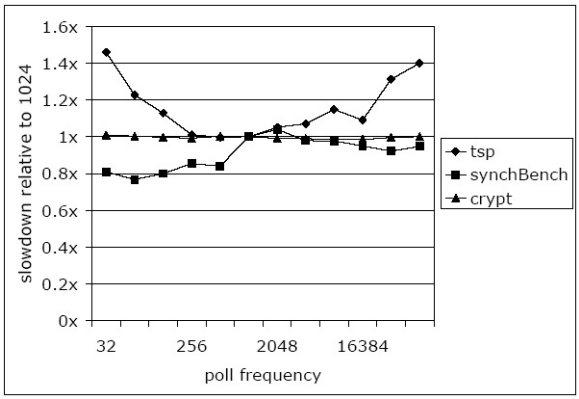

Figure 7. Effect of polling frequency with optimized strong atomicity, 8 processors, 8 threads. The x-axis has powers of 2 from 32 to 64K.

Hence the logs can leak space if they hold references to otherwise unreachable objects. For our benchmarks, the number of objects is too small for these potential leaks to matter, but a safer approach is to write `null` in the log entries when an atomic block commits. (Doing so does not change the asymptotic running time of transactions.) As the fifth column in Figure 6 shows, the slowdown from taking the time to write these `null` values is noticeable only for `synchBench`, in which we have no computation except very short atomic blocks (so we are adding a significant amount of work).

5.4 Summary

With weak atomicity, our performance for small benchmarks is surprisingly good considering the work we add and the lack of virtual-machine support. With strong atomicity, the results are less impressive. Optimizing away barriers improves performance, but too much unnecessary synchronization remains to achieve much parallelism. Tuning parameters can affect results, but not dramatically.

6. Effect of Source-to-Source Translation

With the benefit of hindsight, we discuss advantages and disadvantages of performing transactional-memory research via source-to-source-translation. Many of the conclusions apply broadly to research on memory systems for modern programming systems.

The most obvious advantages of source-to-source translation are rapid prototyping (making it easy to experiment with many settings and implementations), ease of programming (e.g., writing the "run-time" for our system in Java source code), and portability (i.e, working with any Java implementation). For memory studies where contention and synchronization are the primary factors, working at the source level is perfectly fine assuming lower levels do not introduce false sharing.

Probably more important but less obvious are the performance and correctness advantages of "reusing" an unchanged Java compiler. We know we have not broken or slowed down another memory-system component (e.g., the garbage collector) because we are completely decoupled. Put another way, we neither had to reimplement Java from scratch nor modify a complicated optimizing implementation; both would surely lead to bugs. Other minor advantages include the debugging advantages of producing source code that must type-check and for which sophisticated tools exist. For example, we used Java PathFinder [30] to check simple correctness properties of our run-time system.

The most obvious disadvantages of source-to-source translation are relying on an uncontrollable back-end for performance (e.g., method-inlining), sacrificing opportunities to share overhead (e.g., combining our polling with heap-limit checks), and having to compromise semantics when features are simply unavailable (e.g., rolling back class-loading).

Much less obvious when we began is the extra work required to produce source code that passes Java's semantic analysis. For example, the result of translating `atomic{`s`; return 1;}` is not something that Java's compile-time checker deems to "always return" (even though it does), so we have to insert a second (unreachable) return statement. A good compiler framework like Polyglot helps with these issues, but it really is work that should not be necessary. Another practical disadvantage is having to deal with arrays and unchangeable standard-library classes. For example, we cannot add `currentHolder` fields to arrays, even though this would be straightforward with virtual-machine support.

7. Related Work

Language design and implementation for transactional memory is a very active research area [29], particularly in the programming-language [6, 2, 16, 27, 9] and architecture [24, 26, 3] communities, but we believe our source-to-source translation and lock-based implementation are novel. This section briefly describes other language designs, atomicity implementations, and systems using similar implementation techniques.

Language Design We currently provide only the simplest language construct for software transactions. Prior work has provided conditional critical regions [13], better support for external actions [12, 27], alternative composition [14, 2], open transactions [6], and nested transactions [2]. Most systems let a transaction abort explicitly. Nothing in our approach precludes these features. Next-generation languages Fortress, Chapel, and X10 have transactions, but implementations are not yet available.

Implementation Approach Other languages guaranteeing atomicity and fair-scheduling employ one or more of: special-purpose hardware [11, 26, 3, 6], optimistic concurrency protocols for software transactional memory [28, 13, 14, 16, 2], limiting execution to one processor [27, 19], or not updating shared memory until a transaction completes [13, 32]. Software approaches that require exclusive ownership for writes to shared memory are closest to our approach, but they still use version-number techniques for reads [16, 2, 7]. That is, one can view these approaches as locking on write but using optimistic concurrency for reads. This approach allows read-parallelism (as would reader-writer locks) but in theory can waste more computation in transactions that abort.

Other implementations use a library [22, 21, 10, 17, 7] rather than a language extension. Like source-to-source translation, this avoids changing a compiler, making it easier to experiment with different techniques and parameters. Unlike a language extension, programmers must manually wrap access to transactional memory in library calls, which can require manual code duplication for functions and libraries used inside and outside transactions. It is unclear how to enforce strong atomicity via a library.

Pessimistic Atomicity The "pessimistic atomic sections" in Autolocker [23] share the most implementation ideas with our work, but there are substantial differences. In Autolocker, a C programmer uses `atomic` and also annotates data with what lock (if any) guards access to it. A whole-program analysis then determines if it can implement `atomic` by acquiring locks such that deadlock is impossible. Salient differences with our system include: (1) Autolocker provides weak atomicity. (2) Autolocker does not provide even weak atomicity if the programmer wrongly indicates that data accessed within a transaction does not need a lock. (3) Autolocker does not provide rollback or fairness: A transaction that does not terminate will hold locks forever, which can starve other threads.

Strong vs. Weak Atomicity Weak and strong atomicity are semantically incomparable (see [4], which also coined the terms), and it is widely believed that strong atomicity is better for software-engineering but worse for performance. We believe we are the first to investigate the performance of strong atomicity without assuming novel hardware [3, 6], a uniprocessor [27, 19], or a purely functional language that segregates mutable memory that can participate in transactions [15]. One could provide strong atomicity in a weak atomicity system in other ways, such as using a sound data-race detector [1, 5] or treating every memory operation as "its own little transaction." Our implementation is *not* equivalent to the latter because outside transactions we require ownership but not logging.

8. Conclusions and Future Work

We have presented a prototype for software transactions implemented in terms of locks. We have shown the approach can apply to a full object-oriented language and that (at least for small benchmarks) whole-program optimization can ameliorate some of the costs of strong atomicity. As a source-to-source translation written with an extensible compiler, our implementation provides an unprecedented level of portability and serves as an easy-to-use starting point for us and others in ongoing research.

Looking forward, many design parameters remain uninvestigated. We would like to use our system to study the following:

- Ownership granularity: Our prototype groups ownership of all an object's fields or array's indices by using exactly one `currentHolder` field for each object. Other granularities, both

finer (e.g., each array index) and coarser (e.g., entire data structures), can improve performance in some situations.

- Advantage of virtual-machine support: We would like to determine which aspects of the translation benefit most performance-wise when we implement them beneath the Java layer.
- More barrier removal: We believe extending our barrier-removal optimizations with alias analysis and escape analysis will improve the performance of strong atomicity even further.
- Early lock releasing: We currently do not release ownership of an object until another thread requests it, but any release-point outside a transaction is sound. For contended objects, an "early release" could improve performance.
- Reader-writer locks: Extending the notion of `currentHolder` to allow parallel readers is straightforward in principle, but it remains to investigate if the extra complexity helps performance.

Longer term, we expect hybrid approaches, dynamically adjusting between our lock-based approach for less contended data and optimistic approaches for more contended data, may prove best.

Acknowledgments: A Microsoft gift and a University of Washington Royalty Research Fund grant supported this work.

References

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems*, 28(2), 2006.

[2] 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 *ACM Conference on Programming Language Design and Implementation*, June 2006.

[3] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *11th International Symposium on High-Performance Computer Architecture*, Feb. 2005.

[4] C. Blundell, E. C. Lewis, and M. M. K. Martin. Deconstructing transactions: The subtleties of atomicity. In *4th Workshop on Duplicating, Deconstructing, and Debunking*, June 2005.

[5] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Nov. 2002.

[6] B. D. Carlstrom, J. Chung, A. McDonald, H. Chafi, C. Kozyrakis, and K. Olukotun. The Atomos transactional programming language. In *ACM Conference on Programming Language Design and Implementation*, June 2006.

[7] R. Ennals. Software transactional memory should not be obstruction-free, 2005. http://www.cambridge.intel-research.net/ rennals/notlockfree.pdf.

[8] C. Flanagan, S. N. Freund, and M. Lifshin. Type inference for atomicity. In *2005 ACM International Workshop on Types in Language Design and Implementation*, Jan. 2005.

[9] D. Grossman. Software transactions are to concurrency as garbage collection is to memory management. Technical Report 2006-04-01, Univ. of Washington Dept. of Comp. Sci. & Engr., Feb. 2006.

[10] R. Guerraoui, M. Herlihy, and S. Pochon. Polymorphic contention management. In *19th International Symposium on Distributed Computing*, Sept. 2005.

[11] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (tcc). In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

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

[13] T. Harris and K. Fraser. Language support for lightweight transactions. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2003.

[14] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In *ACM Symposium on Principles and Practice of Parallel Programming*, 2005.

[15] T. Harris, S. Marlow, and S. Peyton Jones. Haskell on a shared-memory multiprocessor. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, Sept. 2005.

[16] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *ACM Conference on Programming Language Design and Implementation*, June 2006.

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

[18] B. Hindman and D. Grossman. Strong atomicity for Java without virtual-machine support. Technical Report 2006-05-01, Univ. of Washington Dept. of Computer Science & Engineering, May 2006.

[19] J. Manson, J. Baker, A. Cunei, S. Jagannathan, M. Prochazka, B. Xin, and J. Vitek. Preemptible atomic regions for real-time Java. In *26th IEEE Real-Time Systems Symposium*, Dec. 2005.

[20] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *32nd ACM Symposium on Principles of Programming Languages*, Jan. 2005.

[21] V. J. Marathe, W. N. S. III, and M. L. Scott. Adaptive software transactional memory. In *19th International Symposium on Distributed Computing*, Sept. 2005.

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

[23] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. In *33rd ACM Symposium on Principles of Programming Languages*, 2006.

[24] A. McDonald, J. Chung, B. D. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. In *33rd International Symposium on Computer Architecture*, June 2006.

[25] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *12th International Conference on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, Apr. 2003.

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

[27] M. F. Ringenburg and D. Grossman. AtomCaml: First-class atomicity via rollback. In *10th ACM International Conference on Functional Programming*, Sept. 2005.

[28] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, Special Issue(10), 1997.

[29] TRANSACT: 1st ACM SIGPLAN workshop on languages, compilers, and hardware support for transactional computing, June 2006. http://www.cs.purdue.edu/homes/jv/events/TRANSACT/.

[30] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2), Apr. 2003.

[31] C. von Praun and T. R. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *ACM Conference on Programming Language Design and Implementation*, 2003.

[32] A. Welc, S. Jagannathan, and A. L. Hosking. Transactional monitors for concurrent objects. In *European Conference on Object-Oriented Programming*, 2004.