Strong Atomicity for Java Without Virtual-Machine Support

Benjamin Hindman Dan Grossman
b@cs.washington.edu djg@cs.washington.edu
Department of Computer Science & Engineering
University of Washington
Seattle, WA 98195

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 transformation, 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 software-transactional-memory, but it cannot deadlock and requires inter-thread communication only when there is data contention.

We evaluate our approach by qualitatively considering its overhead, quantitatively measuring the performance impact on benchmarks, and considering several difficult interactions with other language features. We conclude that it is possible to support "strong" atomicity and separate compilation, but simple whole-program optimizations improve performance significantly. In particular, a simple and effective analysis can identify fields that are never accessed in an atomic block.

Categories and Subject Descriptors

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

General Terms

Languages

Keywords

Atomicity, Transactions, Concurrent Programming, Java

1. INTRODUCTION

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*. Atomicity can complement or replace existing synchronization mechanisms with the statement form `atomic { s }` where s is a statement. Semantically, it 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.

The software-engineering advantages of atomicity are numerous and not the focus of this paper. We note simply that `atomic` lets programmers achieve mutual-exclusion while accessing any number of objects, without risking deadlock or relying on other threads to obey a locking protocol.

The obvious rub is that language implementations must assume the considerable burden of implementing `atomic` well. We believe this burden is well worth the benefit, much as the benefits of automatic memory management outweigh the burden of language implementations providing good garbage collectors [15]. Implementation techniques for transactions in general-purpose programming languages is a relatively new but extremely active area [18, 19, 16, 28, 22, 27, 5, 21, 2, 10] to which this paper contributes.

1.1 Strong vs. Weak Semantics

The definition of atomicity given above is "strong" atomicity: Transactions must behave as though no other threads have interleaved computation. Under "weak" atomicity, the language implementation is allowed to interleave computation that is not itself in transactions. As Section 7 discusses, most prior work provides only weak atomicity, and work providing strong atomicity has assumed new hardware or a uniprocessor.

Our prototype supports both semantics, and our experiments mostly confirm the conventional wisdom that the performance overhead of strong atomicity implemented in software on a multiprocessor is large. However, we develop a novel compile-time optimization to recover much of the overhead. Moreover, with our implementation approach, strong atomicity is not as expensive as treating every memory access as "its own little transaction."

1.2 Our Approach

Prior atomicity implementations have used some combination of optimistic concurrency protocols [29, 18], hardware support for aborting transactions [16, 27, 5], or a uniprocessor execution model [28, 22]. In this work, we evaluate an extreme, novel, and complementary implementation approach in the context of Java: We use automatically-managed locks to manage contention and have transactions rollback 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 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 fields 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 all threads to "poll" for locks they must release. If in an atomic block, "rollback" (abort) the transaction before releasing a lock.

Essentially, parts (1), (2), and (5) ensure no uncompleted 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. Conventional back-off techniques can avoid livelock. Note that 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 provided a thread is not in an atomic block and it reacquires the lock as necessary.

This basic description is the core of our approach, but we will describe many complications arising from modern language features. We will also consider implementation issues including compile-time optimization and tunable performance parameters.

1.3 Source-To-Source Translation To Java

Our prototype is also novel in that we have not modified a (Java) compiler nor a (Java) virtual machine. Rather, we perform a source-to-source transformation: We take a program written with `atomic` and translate it to a Java source program. Together with a few classes we wrote in Java (our "runtime system" for atomicity), we could 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.

One can view the source language for this translation as "Java with `atomic`" or as a dialect of Java. Our source language provides support (with some limitations as discussed later) for many difficult language features such as static initializers, exceptions, arrays subsumed to `Object`, and adherence to the Java Memory Model. Our focus has been on supporting atomicity in a full-featured language, but not necessarily integrating it with Java's semantics as carefully as an official language extension would require.

Conversely, although our translation occasionally uses particular Java features, our approach should apply to any object-oriented language. That is, we do not "pull any Java tricks" that restrict the applicability of lock-based atomicity or source-to-source translation.

1.4 Contributions

The focus of this work is a lock-based source-level implementation of atomicity. As such, we make little contribution to language-design issues beyond investigating the cost of strong versus weak atomicity. Our most novel contributions are:

- A proof-of-concept that implementing atomicity does not require virtual-machine or hardware support. This indicates that even with such support (which could improve performance and simplify the implementation) transactions can be largely decoupled from other components of the run-time system.

- A correct design for transactions that uses a lock-based approach and requires inter-thread communication only when there is data contention (Sections 2–4). This design enables future work (Section 8), such as adjusting locking granularity dynamically.

- A simple whole-program optimization that can recover much of the overhead of supporting strong atomicity by identifying data that is never accessed within transactions (Section 5).

We also conduct a preliminary performance evaluation for small interesting benchmarks (Section 6) and discuss related work (Section 7).

2. BASIC APPROACH

This section describes our implementation assuming the entire program consists of user-defined classes containing only constructors, instance methods, and field accesses. We defer language features such as arrays, native code, the standard library, static fields, methods, and initializers, exceptions, etc. In terms of Figure 2, we assume for now all types are (after translation) in the left two branches of the figure, i.e., they are not arrays, exceptions, or certain classes defined in the Java library.

We call a Java program that contains `atomic` an AtomJava program. Our implementation, built with the Polyglot extensible compiler [26], takes an AtomJava program and produces a Java program, which can then be compiled and run with any Java implementation. The basic approach in this section obeys separate compilation: we can compile each class as independently as a Java compiler can.

Figure 1 contains an example demonstrating the features in this section and how they fit together.

2.1 Acquiring and Releasing Locks

We use an extra field in every object and several new fields for each thread. As a source-to-source transformation,

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

```
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 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);
 }
}
```

```
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);
   ++i; __aj_f();
   A.set_atomic_x(get_a(this),A.get_x(get_a(this))+i);
 }
}
```

Figure 1: Class definitions before (above) and after (below) source-to-source translation (eliding access modifiers and package names). Later we slightly modify the translation of atomic{s} **and introduce optimizations.**

we cannot change the definition of Object or Thread, but we can create subclasses AObject and AThread respectively. Any AtomJava class that extends 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 x is roughly this method of AObject:

```
0. void acq_mylk() {
1.   AThread me = (AThread)Thread.currentThread();
2.   if(x.currentHolder != me)
3.     while(true) {
4.        if(x.currentHolder == null) {
5.          x.currentHolder = me;
6.          break;
7.        }
8.        x.currentHolder.lks_to_release.add(x);
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 (Section 2.3) before releasing any locks.

2.2 Synchronization of Locks

The above scheme ensures all field accesses require locks and locks always become available eventually. Rolling back a transaction before releasing a lock ensures strong atomicity. However, the actual implementation is more complicated: The currentHolder and lks_to_release fields are thread-shared so we must synchronize access to them.

For lks_to_release, each AThread has its own monitor (i.e., an object) that every thread acquires before accessing lks_to_release. Deadlock is impossible because no thread acquires another lock 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 6 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 x in the code above), but if user code also synchronizes on the object, we could introduce deadlock. So currently we conservatively use a separate array of monitors for controlling access to currentHolder fields. We use System.identityHashcode on an object to index into this array.

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, 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 [23], 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.

2.3 Logging and Rollback

Correctness demands an atomic block not release locks for any objects it has accessed, but this could lead to deadlock. 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 does everything the latter does plus 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 simply 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.) In particular, we must address these issues:

- 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.
- We do not want a field-assignment to cause memory allocation (for the log data structure).

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

[1]Throughout this paper, we ignore name-mangling issues. For example, we actually call the method __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.

```
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:

```
   int i = obj_log_index;
   for(;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`, we want an `UndoObject` with this method:

```
public void undo(Object container, Object old) {
  ((C)container).f = (D)old;
}
```

Note 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 space consumption. 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 simply 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 with bodies 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 6 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 [22] and our experiments indicate that whole-program analysis can remove most of this code duplication because most methods are never used within a transaction.)

One obvious exception to the above description 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 the common "flattened semantics" of 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; }` would become:

```
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(); }
}
```

Note that methods like `__aj_m` do not log their local variables; they are simply popped off the stack when a rollback occurs. Similarly, writes to fields of `this` in the atomic version of constructors need not be logged since the constructed 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 present some less interesting details relevant to the translation described earlier in this section.

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. Generating new local variables is an alternate solution.

Log-duplicates. If the same field of the same object is set multiple times in the same atomic block, we need to log only the first one. As in previous work [28], we exploit that most atomic blocks have few writes, so it is faster not to detect duplicates. However, to avoid pathological situations (such as `atomic{for(i=0;i<10000;++i) o.x=f(o.x);}`), we detect and remove duplicates once the log arrays fill. We currently use a simple $O(n^2)$ approach: Two entries are duplicates if the containers and undoers are the same object. If after duplicate removal the log arrays are still over half full, we create new arrays twice as long. Therefore, for atomic blocks that do many writes to distinct fields, we do allocate memory for the larger logs.

Using `AThread`. We need threads to be a subclass of `Thread`, so Java's single inheritance means threads cannot be a subclass of `AObject` (see Figure 2). Therefore, we also have a `currentHolder` field and `acq_mylk` method in `AThread`.

We also seem to assume every `Thread` is an `AThread` lest an expression like `(AThread)Thread.currentThread()` fail. We can almost ensure this by rewriting `new Thread(...)` to `new AThread(...)`, but the first thread and any other threads the virtual machine creates remain problems. For the former, we rewrite every `main` method to start an `AThread` and run in that thread. For the latter, we can catch the cast exception and use a look-aside table that has an `AThread` for any `Thread` that is not an `AThread`. In practice, the static method `AThread.currentThread` does this and all uses of `(AThread)Thread.currentThread()` in our earlier examples are actually `AThread.currentThread()`.

Final fields. We can read `final` fields directly; there is no need to generate getters, setters, and undoers. Similarly, for `final` local variables accessed within an atomic-block, we need not and must not try to do roll-back. What remains is initialization of 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. (Indeed, after translation the assignment appears within a loop and therefore will not compile.) In practice, this corner case has never been a problem.

Field and instance initializers. Given a field initialization like `T x=f();` we cannot just generate atomic and non-atomic versions like we do for methods and constructors. So here and only here we suffer a run-time test to determine if the running thread is in a transaction. We can use an instance initializer to share this test across many field initializations provided we do not change the order of initialization code.

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.[3]

[3]Inner classes cannot have static members, so we put the methods (and undoer) in a containing non-inner class.

2.6 Summary

To review our source-to-source transformation so far:

- 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).
- For every object, there is a `currentHolder` field. A global array of monitors synchronizes access to such fields.
- For every loop and method, there is 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. OTHER LANGUAGE FEATURES

In this section we "scale up" our basic approach to support interaction with other Java language features (variants of which exist in most object-oriented languages). Readers can safely skip obscure features they find less interesting.

Some features we can support fully (arrays, static fields, other concurrency primitives, and exceptions). For other features we have had to limit 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: We need all code to obey the invariants of our translation, but our translator is unable to change native code or classes whose definition is assumed by the virtual machine (i.e., certain standard-library classes). The latter issue is the subject of Section 4.

We have chosen to make these limitations lead to 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 matter of policy.

3.1 Arrays

Because we do not modify the virtual machine and cannot make array types subtypes of `AObject` (see the right side of Figure 2), we cannot add a `currentHolder` field to arrays. Therefore, we use a look-aside table: A separate fixed-size array of `AThread` references is indexed into with the array's hashcode to find the `currentHolder`. This basic approach has several shortcomings:

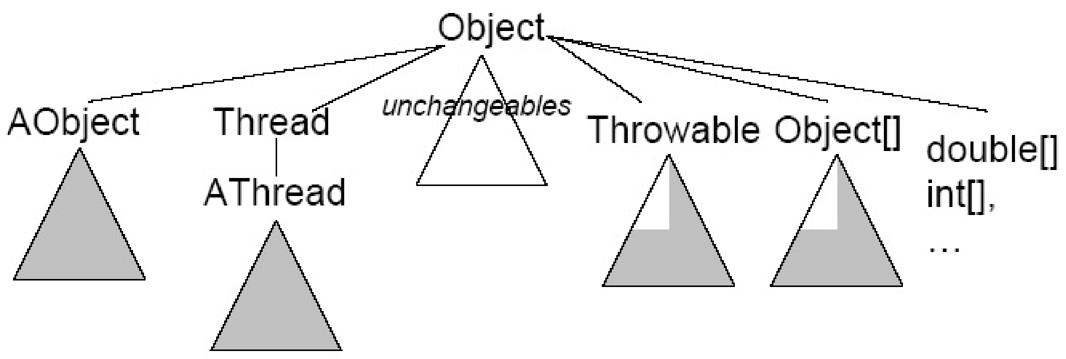

Figure 2: The type hierarchy after translation. Shaded areas contain types that the translation changes (and arrays of such types). Moving left-to-right: Subtypes of `AObject` and `AThread` are described in Section 2. Nonshaded classes and subtypes of `Throwable` are described in Section 4. Arrays are discussed in Section 3.

- It disallows parallel transactional access to disjoint portions of the array. This restriction is easily relaxed by indexing into the look-aside table with some combination of the hashcode and the index, i.e., this is just a question of locking granularity and any answer is easy to implement.

- Every array access (modulo optimization) requires calling the array's `hashCode` method.

- There is false sharing among arrays that hash to the same index in the table. Resizing the table dynamically requires additional synchronization.

We define getters and setters for arrays in a library and rewrite all subscripts to use the getters and setters. This library uses a generic method for all arrays with elements that are subtypes of `Object`, but we could also rely on covariant array subtyping and have callers downcast the result from `Object`.

A tempting alternative, creating a wrapper class for every array type and having the wrapper provide a `currentHolder` and getter/setter methods, does not interact well with separate compilation: Throughout the program, we would need every use of type `T[]` to use the same wrapper class. But we cannot create all wrappers when compiling a class `C` because we may need wrappers for `C[]`, `C[][]`, `C[][][]`, etc.

3.2 Static Fields and Methods

Static fields are easy to support in a manner analogous to instance fields and methods. For fields, we create getters, setters, and undoers and use a new static field for the `currentHolder` (since the fields are not part of an `AObject` instance). For a static field `f` in class `C`, the getters, and setters take a `C` as their first argument but do not use it. That way, `C.f` can be translated to `C.get_f(null)` and `c.f` can be translated to `C.get_f(c)` where `c` has type `C`. Similarly, the atomic-setter always uses `null` for the container-object in the log, and the `undo` method of the undoer ignores its first argument (which will be `null`).

We duplicate static methods just like instance methods.

3.3 Dead Threads

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 will never 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 pointers or no object would become garbage unless the owning thread did.)

Instead, the thread requesting an object checks if the holding thread is no longer alive. If it is not, then it is safe to set the object's `currentHolder` to `null` "on behalf" of the dead thread, using appropriate synchronization. This "lock stealing" is a simpler case of the stealing discussed next for interaction with legacy synchronization. It is simpler because terminated threads never run again.

3.4 Other Synchronization

In addition to termination, a thread cannot release objects if the thread is sleeping or blocked indefinitely since polling occurs only if the thread is running. Using `synchronized`, `wait`, or `join` could cause blocking. By allowing these features, we support "legacy code" and libraries without requiring changes, but we need to revoke access to objects held by blocked threads.

To do so, our source-to-source translation "surrounds" any potentially blocking operation with calls that indicate the thread may be blocked. For example, the statement `synchronized(`e`){`s`}` becomes something like:

```
T tmp = e;
((AThread)currentThread()).maybeBlocked();
synchronized(tmp) {
  ((AThread)currentThread()).unblocked();
  s;
}
```

(To enable this translation, we desugar synchronized methods to method bodies that synchronize on `this` and perform a similar transformation for static synchronized methods.)

The calls `maybeBlocked` and `unblocked` maintain a thread-shared `boolean`. If set, another thread "steals" access to an object immediately. A monitor guards this `boolean` to prevent race conditions. When "stealing" a lock, the stealer also sets thread-shared (and properly synchronized) state that `unblocked` checks. In the case a thread "was robbed" while in an atomic block, it aborts the transaction.

Note that using `synchronized` in a transaction causes no problems; if the transaction aborts, we throw an exception which will transfer control such that the monitor is released.

3.5 Exceptions

Throwing and catching exceptions is largely unchanged. Atomic code that throws an exception caught outside the transaction causes the transaction to commit; a `finally` clause in the translation of `atomic` ensures the transaction commits properly. Within atomic code, our translation uses a `Rollback` exception to abort a transaction. Therefore, we must ensure user-provided `catch` and `finally` clauses do not "see" this exception. For `catch` clauses, we add a first clause that catches and rethrows a `Rollback` exception (if there is a clause that catches a supertype of `Rollback`). For `finally` clauses, we add a statement at the beginning that consults thread-local data to see if a rollback is "in progress" and if so rethrows the exception.

User-defined exceptions are subclasses of `Throwable`, so they cannot be subclasses of `AObject`. This complication is the subject of Section 4 because it is a general problem with classes in the standard library that we cannot change without virtual-machine support.

3.6 Native Code (Including I/O)

We currently provide limited support for native code:

1. Calling a native method from within a transaction throws an exception. Implementing this semantics is elegant: The atomic version of a native method is a non-native method that contains the throw statement. So native methods can override non-native methods and vice-versa.

2. We do not let native code directly access fields of Java objects. (We do not actually check this.) Native code can call (non-atomic) methods or constructors.

3. We assume native code terminates and does not block indefinitely.

We could relax the first restriction as in our previous work on AtomCaml [28] by requiring native-code libraries to provide atomic (and non-atomic) method versions and supporting library-registered commit- and abort-actions.

With virtual-machine support or a separate tool, we could remove the second restriction by interpreting field accesses to be the appropriate method calls.

We could remove the third restriction if we allowed lock-stealing (see above) while native-code executes. Doing so is difficult because we do not always know at compile-time that a call-target is native-code.

3.7 Reflection

We currently disallow reflection. Clearly our translation can change the behavior of reflective programs (since we add fields and methods) and reflective programs can violate atomicity (since we cannot control the fields and methods they access). It is possible that in practice many uses of reflection are unproblematic; the applications and benchmarks we have considered do not use reflection.

Note our translation does not itself use reflection.

3.8 Finalizers

Finalizers are difficult without virtual-machine support because an object created by an aborted transaction should not have its finalizer run. Therefore, if an object has a user-defined finalizer, we disallow creating such an object within a transaction by changing the atomic versions of constructors to raise a run-time exception.

Note we need not worry about a transaction making an object unreachable (causing the object's finalizer to run) and then a rollback making the object reachable again: Logging ensures all objects reachable at the start of the transaction remain reachable until the transaction completes.

3.9 Class Loading and Static Initializers

We allow classes with static initializers, but our lack of virtual-machine support prevents us from extending Java's precise rules for when classes are loaded. To see the main problem, suppose a transaction causes a class to be loaded but the transaction later aborts. We cannot "unload" the class or cause the static initializers to run later. As a second problem, suppose we must abort a transaction while running a static initializer. We cannot throw the `Rollback` exception because Java forbids uncaught exceptions within static initializers. As a third problem, if one thread needs a class another thread is currently loading, the first thread implicitly blocks and our transformation cannot call `maybeBlocked` as described above.

To solve the first two problems, if a transaction causes a class with static initializers to load, we first rollback the transaction, then load the class, then restart the transaction. While unproblematic in practice (and with the advantage that static initializers do not see mid-transaction state), this semantics violates the spirit of Java's precise timing of class loading. First, we conceptually load the class early. Second, we may never actually use the class since other threads may change the program state before we rerun the transaction.

Even so, implementing this semantics is tricky since we do not want to check explicitly at every program point where a class might be loaded. Instead, if a class has static initialization, we add an initial static initializer that checks if the running thread is in a transaction. But this static initializer cannot throw an exception. Instead, we rollback the data in the thread-local logs and set a thread-local "doomed" flag before executing the other static initializers (which will now see the pre-transaction state). Later, the code that sees if locks must be released (and the code that completes a transaction) checks the "doomed" flag and if set aborts the transaction (doing a second rollback).[4] Most oddly, the first rollback may cause the transaction to follow otherwise impossible control-flow paths, but we still know the "doomed" flag will eventually be checked and the transaction correctly aborted.

The third problem (threads blocked on class loading by other threads) could in theory lead to deadlock. Though we have not yet implemented a fix to this, we believe we can create a thread that detects a thread needing an object owned by a thread blocked in this way and have this "third-party" thread release the object (and set the blocked thread's "doomed" flag if it is in a transaction).

[4]Thus the translation of `atomic` in Section 2 requires the modification that the `finally` clause check the "doomed" flag.

4. BUILT-IN CLASSES

Our translation so far assumes we can add to almost every object a `currentHolder` field, two versions of each method, etc. We can do this for instances of almost any class where the source code is available (the left two branches of Figure 2). Section 3 extended the translation for array types (the right two branches of Figure 2). This section considers the remaining objects.

The essence of the problem is our lack of virtual-machine support: Any class for which the virtual-machine (1) assumes the class definition (e.g., `java.lang.System`) or (2) may create instances (e.g., `java.lang.IOException`) must be handled differently. We call such a class *unchangeable*, and we assume our translation knows what the unchangeable classes are. (We currently treat classes in `java.lang` and `java.io` as unchangeable, but this categorization may not be exactly accurate.)

We address the following issues:

- Is it sound to call a method (or access a field) of an unchangeable class (either outside or inside a transaction)?
- Can we subclass an unchangeable class?
- How can we call a method if the static type is `Object` since the run-time type could be an unchangeable, an array, or neither.

In providing (partial) answers, we err toward disabling use rather than potentially violating atomicity, though this is only a policy choice. Nonetheless, we have architected our translator to make it easy to extend support for unchangeable classes.

Calls and Accesses.

For simplicity, we disallow accessing non-final fields of unchangeable classes directly. There are very few such fields because the standard library is well-designed. The translation of any such field access throws an exception.

For calls, we also by default translate the call to throw an exception, just as if the target were a native method. However, we have implemented a library of wrappers and if the proper wrapper is present, it is called instead. For example, our wrapper-library contains this class definition in package `wrapper.java.io`:

```
public class PrintStream{
  public static void
  print(java.io.PrintStream p, java.lang.String s){
    p.print(s);
  }
  public static void
  flush(java.io.PrintStream p){
    p.flush();
  }
  public static void
  __aj_flush(java.io.PrintStream p){
    p.flush();
  }
  // ... additional methods omitted ...
}
```

These wrappers, written by hand (and in practice on a demand-driven basis as our benchmarks have required it), support calling `print` from outside transactions and `flush` from outside and inside transactions. Given a call, e.g., `e.print("hi")`, the translator simply looks for a wrapper of the correct type. If present, it calls the wrapper, e.g. `wrapper.java.io.PrintStream.print(e,"hi")` else it produces an exception throw.

This design lets us add support for methods incrementally simply by writing new wrapper classes and methods. It does not even require recompiling our translator, but it does require recompiling code that uses a newly supported method.

While the wrappers above are straightforward, we have also implemented more sophisticated wrappers, such as for the `StringBuffer` class where we avoid deadlock by carefully acquiring the right locks before forwarding the call. We also allow the atomic version of wrappers to register actions that should be performed if the transaction aborts. (The implementation is just another log with wrapper-provided `Undo` objects.) Our `StringBuffer` wrappers use this feature to allow `append` operations inside transactions.

Subclassing.

We have not yet needed to let programs provide subclasses of unchangeable classes, nor have we fully worked out the details of how to do so. We believe this is an acceptable limitation except for the unchangeable class `Throwable` and unchangeable subclasses of it. Fortunately, these unchangeable exception classes are easy to support because they have no mutable state. That is, we can allow method calls on them even inside transactions, though the translator must be aware that atomic versions of the methods do not exist.

It remains that a subclass of an unchangeable exception is not an instance of `AObject`, so the translator may need to add a `currentHolder` (unless a superclass already has one). In practice, we have not yet experimented with a program that extends a subclass of `Throwable` with fields.

`Object` calls.

Consider this method prior to translation:

```
boolean m(Object o) {
  return o.toString().equals("hi");
}
```

Our wrappers can inform our translator that `equals` is allowable inside and outside transactions for the (final) class `String`. But the `toString` call is much more problematic: The run-time type of `o` could be anywhere in Figure 2. Therefore, the atomic and non-atomic wrappers for calls whose receiver has static type `Object` use downcasts to approximate the correct behavior. For example, a `toString` call inside a transaction calls this wrapper:

```
 static public java.lang.String
 __aj_toString(java.lang.Object o) {
   if(o instanceof AObject)
     return ((AObject)o).__aj_toString();
   if(o instanceof AThread)
     return ((AThread)o).__aj_toString();
   if(o.getClass().isArray())
     return o.toString();
   throw new NoSuchMethodError(".__aj_toString");
 }
```

That is, user-defined classes have `__aj_toString` methods (either defined in `AObject` or overridden in a subclass). Array types do not, but for arrays the `toString` method is safe to call. In other cases, we conservatively fail, though in the future we intend to use interfaces to indicate when other classes (unchangeable classes and subclasses of them) support certain methods defined in `Object`.

As for the other methods of `Object`:

- `hashCode` and `equals` are similar to `toString`.
- `getClass` and `wait` are easy to support because they are `final`. (The wrapper for the latter allows lock-stealing.)
- `notify` and `notifyAll` are allowed only outside transactions.
- `clone` requires first acquiring ownership of the receiver since it reads all the receiver's fields.
- Explicit calls to `finalize` are strange but not a problem.

5. OPTIMIZATIONS

The atomicity implementation described in the preceding three sections allows separate compilation: We can translate each class to Java-without-atomicity assuming only all other classes are translated similarly. Furthermore, what the type-checker must know about other classes is no more than what the regular type-checker must know. However, separate compilation and source-to-source translation comes at a cost: As the next section shows, the unoptimized translation can run many times slower than a corresponding Java program. This section describes simple optimizations that reclaim some of this slowdown on our benchmarks. The most interesting and novel is a whole-program type-based analysis that determines most fields are never accessed from within a transaction.

We can identify several obvious reasons our translation makes programs run slower, the last of which turns out to be the most significant:

1. Space: Every object has a `currentHolder` field, which could increase cache pressure or lead to more frequent garbage collection. We have not investigated the actual effects; at this point we can only acknowledge the issue.
2. Current-thread downcasts: Our translation puts its per-thread data in instances of `AThread`, a subclass of `Thread`, so we perform many dynamic downcasts from `Thread` to `AThread`.
3. Polling: Every loop and (non-leaf) method call includes an extra method call to decrement a thread-local counter.
4. Logging and rollback: Transactions must record their effects and if they rollback, all the work they did is wasted.
5. Read and write barriers: Every field and array access includes at least a check that the current thread owns the object. If not, synchronization with another thread is necessary.

The next section also considers some other minor aspects of our translation but establishes that their effect on performance is small.

	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.

5.1 Source-Level Artifacts

Issues (2) and (3) are largely artifacts of not changing the virtual-machine: For (2), we could access per-thread data more efficiently if we did not limit ourselves to Java source code and calls to `Thread.currentThread()`. For (3), we expect a virtual-machine can already stop a thread in a timely manner at a safe control point (to pre-empt the thread, perform a garbage collection, deliver an asynchronous exception, etc.) so we could add the "locks-to-release check" with little additional cost. (There are tricks for this, such as making it appear to the thread that it is being pre-empted and then letting it continue after releasing locks.)

To reduce current-thread downcasts, we can perform common subexpression elimination to retrieve the current-thread once and pass it around. How much to do so introduces the usual space versus recomputation-time trade-offs. The next section shows empirically that simple notions of sharing (roughly retrieving the current-thread once on method-entry and passing it to extra versions of getter and setter methods that take it as an argument) are easy and helpful.

For polling, we have not yet investigated more sophisticated interprocedural algorithms or loop-termination analyses to remove calls. (As previously mentioned, we do omit calls on entry to methods that do not make calls.) As the next section shows, we measured the effect of varying the "polling frequency," i.e., is, how often the polling call actually performs the (synchronized) operation of seeing if locks should be released. As expected, performance suffers from checking too frequently (due to too much synchronization) or too infrequently (due to other threads waiting too long).

5.2 Logging and Rollback

Code within transactions runs slower than code outside transactions because we log effects. As described in Section 2, we ensure logging requires only a method call and few array assignments (unless we must resize the array). Because rollback is rare, its cost is less important. Moreover, if most code does not execute within a transaction, the logging cost is also not crucial.

5.3 Read and Write Barriers

5.3.1 The Problem

Adding overhead to every field and array access is the primary reason translated programs may run so much slower. Therefore, eliminating these read barriers and write barriers (i.e., the extra work for reads and writes to memory) where possible is the goal of our optimizations.

Figure 3 summarizes the barrier overheads without optimization: Under strong atomicity, every access requires "owning" (i.e., having the `currentHolder` be the running

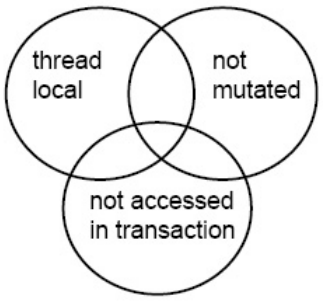

Figure 4: Relation of three sufficient conditions for removing barriers when accessing data in non-transactional code.

thread) the object. Under weak atomicity (transactions appear not to be interleaved with other transactions), only accesses within transactions require "owning" the object. 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.

Our ultimate goal is to approach the performance of weak atomicity with the semantics of strong atomicity by using compile-time analysis to show that most memory accesses outside of transactions do not need barriers.

5.3.2 *Our Approach*

Figure 4 summarizes three reasons a memory access would not need a barrier:

1. The memory is accessed by only one thread. (Transactional writes still need logging in this case, but no ownership barriers are necessary.)
2. The memory is read-only (so races are impossible).
3. The memory is not accessed (or perhaps just read but not written) from within a transaction.

Any one of the three is sufficient for barrier removal. They overlap in two senses: First, a given field may actually satisfy any combination of the conditions. Second, conservative compile-time analyses might determine any subset of the conditions a given field actually has. Also note that as usual with static analysis, establishing any of the properties at compile-time could require alias information.

To our knowledge, recognizing the importance of condition (3) is novel. (There is considerable prior work on showing data is thread-local [3, 12, 6, 8] or read-only [**?**].) It attacks directly the performance difference between strong and weak atomicity (or conversely, it is irrelevant if one allows a weak-atomicity semantics). Therefore, our prototype optimization focuses on this aspect of barrier-removal: we conservatively assume all data is thread-shared and all (non-`final`) data is mutable.

Figure 5 describes when the lack of access within a transaction lets us remove a barrier in non-transactional code: If memory is neither read nor written within a transaction, then no barriers are necessary. If memory is never written within a transaction, then no read barriers are necessary.

	remove barrier outside atomic	
atomic-access	read	write
none	yes	yes
read-only	yes	no
write-only	no	no
read-write	no	no

Figure 5: The barrier removal allowed by an analysis that determines how data is used in transactions.

If memory is written within a transaction, then in general both read and write barriers must remain.

Qualitatively, an analysis establishing that memory is accessed only outside transactions has two advantages over complementary analyses for thread-local or immutable data:

- It is effective even with very little alias information. As we describe below, we do not compute any non-trivial alias information yet we can eliminate most barriers. Essentially, we exploit the common case that a class is either used for data requiring synchronization (therefore accessed transactionally) or data not requiring synchronization, but not both. That is, assuming that all objects of compatible types may be aliased has in practice not destroyed our accuracy.

- It optimizes code that is written using the common producer/consumer pattern. Suppose one thread puts objects in a thread-shared queue and another thread removes them. While the queue data structure is presumably accessed within transactions, the fields of the objects put in the queue presumably are not. These fields are not thread-local and may not be immutable, so only an analysis with our goal would remove barriers on them. (Thread-local data analyses enriched with unique-pointer information [9] can capture this case, but they are typically very sophisticated or require programmer annotations.)

5.3.3 *Basic Analysis and Transformation*

Having outlined the goal of our analysis, computing the results is "textbook" whole-program optimization. We first describe the basic algorithm and then enrich it with some easy-to-compute unique-pointer information that can improve its results.

The basic optimization proceeds in three steps:

1. Approximate what code may be called from within a transaction, which is basically which atomic blocks, atomic methods, and atomic constructors may execute. We simply assume each `main` method (and static initializer) is reachable and every statement of every reachable method is reachable (i.e., we are flow- and context-insensitive). A reachable call makes every possible target reachable. We do not refine the possible targets beyond the static type information: Given `e.m(...)` where `e` has class type `C` (or interface type `I`) we assume any method overriding `m` in a subclass of `C` (or any method implementing `m` in a class implementing `I`) may be the target.[5]

[5]Sound reachability requires all code, so standard-library classes again complicate matters. We conservatively as-

With care, this code-reachability analysis can take time and space proportional to the program size.

2. Determine transactional access. In a single pass over the reachable transactional code, process each field access (`e.f`) and array access `e[i]`. Use the static type of the receiver (`e`) and whether the access is a read or write to compute what is accessed transactionally. Here is where we use only type-based alias information: If `e` has class type `C`, then a read (or write) of `e.f` means every instance of `C` may have its `f` field read (or written) in a transaction.[6] Similarly, if `e` has type `T[]`, then a read (or write) of `e[i]` means every index of every instance of `T[]` may be read (or written) with within a transaction. Note Java's covariant subtyping for arrays can hurt accuracy.

3. Remove barriers. Using the information from the previous step and the table in Figure 5, a single pass over the non-transactional removes read and write barriers.

5.3.4 A Unique-Pointer Extension

This optimization does very well in practice, but we can remove some additional barriers by exploiting some trivial alias information. To motivate the extension with an example from our benchmarks, suppose a program has several integer arrays (type `int[]`). Some are used within transactions, so our optimization so far would have barriers for all `e[i]` where `e` has type `int[]`. But suppose all the following hold:

- Field `f` of class `C` has type `int[]`.
- For every write of the form `e1.f=e2`, `e2` has the form `new int[e3]`, i.e., all assignments use fresh (so unaliased) references.
- There are no reads of the form `e.f` except to index into the array (i.e., `e1.f[e2]`). So we never create an alias of any `e.f` by assigning `e.f` elsewhere, passing it to a method, etc.[7]

Then there can be no aliases of an `f` field of `C`, so we can treat this field's contents "as its own type," distinct from other `int[]` instances. Therefore, if `e1.f[e2]` is not written (or read) within a transaction, we can remove read barriers (or all barriers) on array accesses of this form. Note the barrier we are discussing here is the one for the array access (accessing an index of the array); whether we need barriers for the field access is *not* affected by this extension.

To generalize, this extension is not particular to fields holding arrays: The point is that if we can determine that a field f's contents refer to an unaliased object o, then we know accesses to objects with the same type as o do not access o unless through a field f. The rules above are just a cheap syntactic approximation of this unaliased property.

sume all methods overriding a method in `Object` and similar classes are reachable. We assume calls into the standard library cannot cause (non-overriding) methods in user-defined packages to be called.

[6]We assume an unavailable class cannot access fields of an available class unless the latter is clonable, in which case we conservatively leave in all barriers.

[7]As a minor point, `C` should not be clonable since a `clone` call would also make an alias of an `e.f`.

Note we do not require that containers of the f field are unaliased.

We could also generalize this extension to support unique *paths* starting at some field f (such as `e.f[i].g.h`), but we have not done so and suspect it would not prove very profitable.

6. EXPERIMENTS

We view our current prototype as a proof-of-concept that one can implement atomicity for a modern object-oriented programming language without hardware or virtual-machine support. 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 the overall performance and the effectiveness of our optimizations. We conclude that the overall performance of our approach is sometimes but not always competitive with lock-based Java code, but it is a good starting point for ongoing research. Moreover, even when our translation makes code run much slower, we have a level of portability and ease-of-implementation that is desirable when prototyping, teaching concurrency, or perhaps comparing virtual machines.

Section 6.1 describes our benchmarks and evaluation platform. Section 6.2 describes the overall performance for our benchmarks and the overall effectiveness of optimization. Section 6.3 presents additional results from modifying parameters such as threads' polling frequency. Section 6.4 briefly summarizes our results.

6.1 Benchmarks and Platforms

We have investigated four small programs. For each, we changed uses of `synchronized` to uses of `atomic` after manually verifying that doing so would preserve meaning.

- `tsp` solves a traveling salesperson problem using a user-specified number of threads. It has been used in previous concurrency studies [14, 31]. The 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 program, the locking is coarse: just two separate locks protect all thread-shared data (some of which is immutable) and nontrivial work is done while holding them. 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 seeing stale values leads only to useless work.
- `crypt` is an embarrassingly parallel encryption program in the JavaGrande suite[8] that does not need synchronization (if multiple threads are used, they operate over disjoint data). Therefore it is a useful benchmark 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

[8]`http://www.epcc.ed.ac.uk/javagrande/`

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 [18] in which parallel threads access a shared hashtable with a mix of insert and lookup operations.[9] 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 hashtable supporting parallelism, but we expect to soon.

We ran all experiments using 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 the virtual machine" by first running each program until we saw consistent timing data and then taking the average of twenty runs. This methodology has two caveats:

- Without "warm up" all data had much larger variance, the lock-based code had larger variance than the atomic code, and the slowdown from our translation was on average much lower. That is, the "warmed up" results are worse for our translation. We have not had time to investigate "cold start" times without the `-server` option.

- While most runs have times near the average, occasionally runs take twice as long or longer. We attribute this to unfortunate thread pre-emptions. These outliers exist for the lock-based code and the atomic code but are more common in the atomic code.

We ran experiments on three machines, all running Linux 2.6.12.[10] 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.

6.2 Overall Performance

We can measure the slowdown of the atomic versions of our benchmarks relative to the performance of 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. (We manually verified that the benchmarks are correct with weak atomicity.) With optimization, our translator is given the whole program and produces specialized Java files that we then pass to `javac`. Though we believe weak-atomicity is an inferior semantics, it is an appropriate "limit study" for how well our optimization could do. (In fact, for tsp it is beyond the limit because the application has benign data races.) Figure 6 shows the results for our benchmark programs for each semantics, various numbers of threads, and various machines.

[9] 16% of the operations are inserts.

[10] The 8-processor machine has Red Hat 4.1.1-5. The others have Red Had 4.0.0-8.

The `tsp` program shows a significant slowdown compared to lock-based code, even with weak atomicity. This application has larger atomic blocks than the other benchmarks, and we did instrument the run-time system to observe 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 some value: We recover about half the significant 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.

The `crypt` program has no synchronization (locks in the Java version or `atomic` in our version). Hence this is the ideal case for our optimization: It can and does remove 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 discussion below for `hashtable` for how to avoid this.) Moreover, the remaining overhead after removing barriers (polling for locks to release, space increases, etc.) is minimal; we run only 10% slower than the Java version. The Java, weak, and optimized versions show super-linear speedup on 2 processors and almost 5x speedup on 8 processors.

The original `synchBench` is designed to measure the cost of acquiring and releasing locks. Because all threads contend for the same data, the Java program and our program have no parallelism. In fact, adding parallel threads slows down the program proportionally. 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 number of threads does not exceed the number of processors is reliable, with 16 threads and 8 processors, run times for any semantics vary by several factors. We conclude that short, highly contended atomic blocks have unpredictable performance when there are more threads than processors; the 16-thread data for `synchBench` in Figure 6 may not be useful.

`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. Once we add support for doing so, perhaps along the lines of the work in [2], we can revisit this benchmark and attempt to estab-

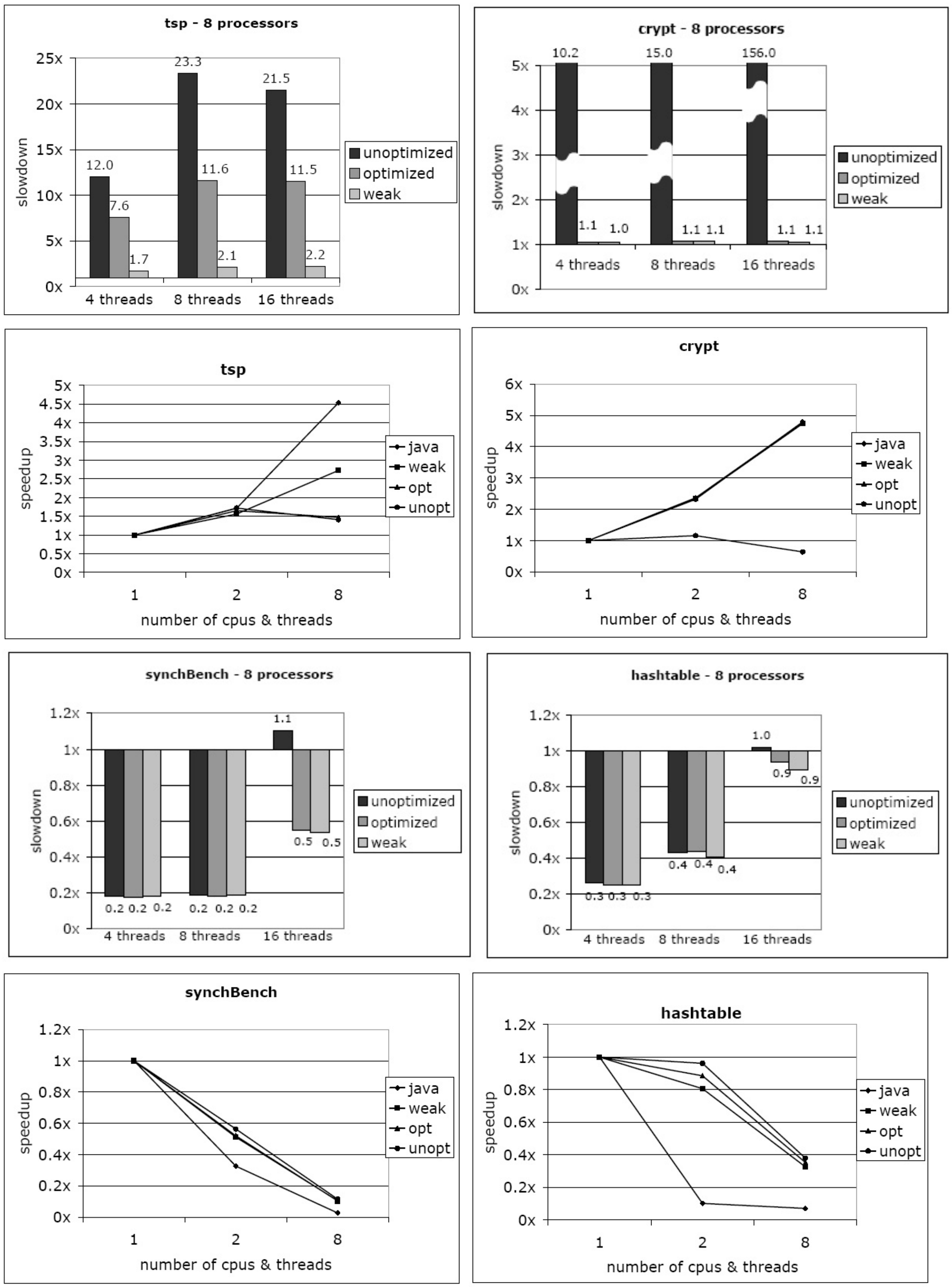

Figure 6: Benchmark performance: The bar graphs show results for our 8-processor machine as we vary semantics, optimization, and thread count. We present slowdown relative to the lock-based Java version. The line graphs show the results for 2 threads on a 2-processor machine and 8 threads on an 8-processor machine, relative to 1 thread on a 1-processor machine (so all results for 1-processor are normalized to 1).

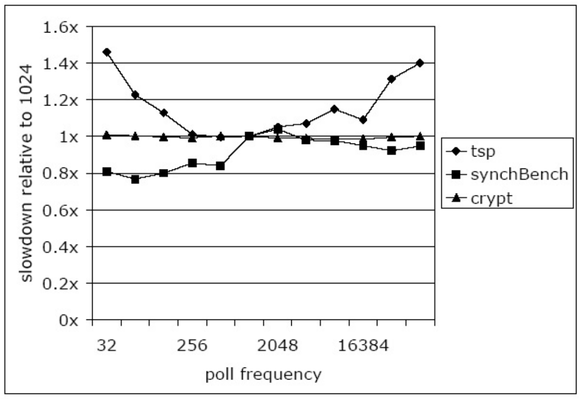

Figure 7: Effect of polling frequency with optimized strong atomicity, 8 processors, and 8 threads. The x-axis has powers of 2 from 32 to 64K.

lish parallelism.

6.3 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." We consider a subset of benchmarks and thread-counts that have more reliable and interesting performance characteristics.

Polling Frequency. Recall that on every loop and non-leaf method call, a thread calls `check_release` but that 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 will wait too long for other threads. If too high, we will spend too much time polling (which involves 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 Java synchronization required when checking for locks to release is extremely fast because the virtual machine special-cases synchronizing on a monitor that is never held by another thread.

Back-Off Policy. We can also consider different back-off policies when a thread rolls back a transaction. To reduce the likelihood of livelock we use exponential backoff, i.e., wait time $b * c^n$ where n is the number of times a transaction has failed to complete. We can choose b and c, though unfortunately the virtual machine we used does not support a result less than one millisecond.[11]

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).

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

We also noticed that while a thread in `tsp` is sleeping after having aborted a transaction, it is common for other threads to "steal" locks it owns (as described in Section 3.4). The reason is this common sequence of events, when two threads are executing code that uses many of the same objects:

- Thread 1 needs object 1, currently owned by thread 2 so it waits for it.
- Thread 2 rolls back a transaction, releases object 1, and sleeps.
- Thread 1 acquires object 1 and then acquires other objects that thread 2 holds by stealing them.

If objects with this sort of locality shared a `currentHolder` (i.e., the ownership had coarser granularity), we expect performance would improve. This idea is our most important future work.

Current-Holder Synchronization. We investigate the cost of using an array of monitors for synchronizing access to `currentHolder` fields. This array is necessary only to avoid deadlock if the program still has `synchronized` statements. Our benchmarks do not, so we can safely synchronize directly on the object whose `currentHolder` field we are accessing, which saves indirection and computing hashcodes. The third column in Figure 8 shows the slowdown (i.e., speedup when numbers are less than 1) with this change. The improvement is less than we expected, suggesting that this level of indirection and hashcode computation is well-optimized by the underlying virtual-machine.

Current-Thread Sharing. Results presented so far include a simple optimization in our translation: Methods that use the current-thread object more than once make one call to `AThread.currentThread` and store the result in a local variable. To make this even more useful, we also use versions of the getter methods that take the current-thread as an argument. As the fourth column in Figure 8 shows, disabling this common-subexpression elimination slows down `tsp` and `synchBench` by 11–36%. This suggests we should add even more sharing to our implementation.

Log Destruction. When an atomic block successfully completes, we just set the log indices back to 0 in preparation for the next `atomic` block. Hence the logs can leak space if they contain references to otherwise unreachable objects. For all our benchmarks, the size of atomic blocks and/or the number of objects is too small for these potential leaks to occur, 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 since the number of log entries is less than or equal to the number of writes that occurred.) As the fifth column in Figure 8 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).

6.4 Summary

With weak atomicity, our performance results for small benchmarks are surprisingly good considering the extra work

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)
tsp (16)	0.74s	0.75s (1.01x)	0.96s (1.29x)	0.74s (1.00x)
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)
crypt (16)	2.94s	2.90s (0.99x)	2.94s (1.00x)	3.00s (1.02x)
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 8: 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.

we add and the lack of virtual-machine support. With strong atomicity, the results are less impressive. Our optimization does improve overall performance, but too much unnecessary synchronization remains to take advantage of a multiprocessor. Tuning parameters such as polling frequency and the cost of finding thread-local data can affect results, but not dramatically. Hopefully simple dynamic adjustment of such parameters will suffice for most applications.

7. RELATED WORK

Language design and implementation for software transactions is a very active research area in the programming-language and architecture communities, but we believe our source-to-source translation approach, lock-based implementation, and barrier-removal optimization are all 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 [18], better support for external actions [17, 28], alternative composition [19, 2], open transactions [10], and nested transactions [2]. Most systems let a transaction abort explicitly; this addition is trivial for us to support. Some systems let an uncaught exception abort a transaction [19, 30] though we believe this semantics is ill-advised [28]. Next-generation language designs including Fortress [4], Chapel [13], and X10 [11] have transactions, but implementations are not yet available.

Implementation Approach. To our knowledge, all prior systems guaranteeing atomicity and fair-scheduling employ one or more of: special-purpose hardware [16, 27, 5, 10], optimistic concurrency protocols for software transactional memory [29, 18, 19, 21, 2], limiting execution to one processor [28, 22], or requiring the programmer to use special library calls for transactional access to shared data. Software approaches that require exclusive ownership for writes to shared memory are perhaps closest to our approach, but they still use version-number techniques for reads [21, 2].

Pessimistic Atomicity. The "pessimistic atomic sections" provided by the Autolocker system [24] 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:

- Autolocker provides weak atomicity.
- Autolocker does not provide even weak atomicity if the programmer wrongly indicates that data accessed within a transaction does not need a lock.
- Autolocker does not provide rollback or fairness: A transaction that does not terminate will hold locks forever, which can starve other threads.
- Autolocker requires whole-program analysis to avoid deadlock, whereas we use whole-program analysis only for optimization.
- Autolocker has the programmer choose locking granularity and may reject a choice (if it cannot avoid deadlock), whereas we pick a granularity behind-the-scenes and any granularity is correct.

Strong vs. Weak Atomicity. Although it is well-known that weak and strong atomicity are semantically incomparable [7], it is also 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 [5, 10], a uniprocessor [28, 22], or a purely functional source language [20]. One could provide strong atomicity in a weak atomicity system in other ways, such as using a sound data-race detector [1, 9] or treating every memory operation as "its own little transaction." Note our implementation is *not* equivalent to the latter because outside transactions we require ownership but not logging.

Optimization. While the performance of the key primitives for atomicity has been considered carefully by the work described above, there has been considerably less work on integrating traditional compiler optimizations, i.e., considering program-specific optimizations. The work we are aware of considers only weak atomicity [2] or transactional monitors [32] (which have an even weaker semantics). The former considers a tight integration of transactions into the compile-time optimizer and the run-time system, leading to performance often within 20% of non-transactional code. The latter inserts read and write barriers into a fairly high-level intermediate representation so that the compiler (in this case, the Jikes RVM) can hopefully perform local optimizations.

8. CONCLUSIONS AND FUTURE WORK

We view our work as the first prototype of atomicity implemented in terms of locks. We have shown the approach scales reasonably well 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 transformation written with an extensible compiler, our implementation will serve as an easy-to-use starting point for us and others in ongoing research.

However, there remain performance limitations to overcome and design parameters that remain completely uninvestigated. Here we sketch the areas of future work we find particularly interesting.

- Ownership granularity: Our current implementation groups ownership of all an object's fields or array's indices but uses separate `currentHolder` fields for separate objects. Other granularities, both finer (e.g., each array index) and coarser (e.g., entire data structures), would improve performance in some situations. We are most interested in dynamically adjusting granularity by changing how objects' ownership is determined by exploiting locality, namely the heuristic that objects accessed in the same transaction are likely to be accessed in the same transaction again. The current owner of an object can change how future ownership is acquired based on dynamic information.

- Advantage of virtual-machine support: While our current prototype gives us an unprecedented degree of portability and let us build our prototype rapidly, we are not opposed to virtual-machine support. More specifically, we would like to determine which aspects of the translation benefit most performance-wise when we implement them beneath the Java layer and the extent to which we can keep atomicity decoupled from other run-time support.

- More barrier removal: We have focused on our novel optimization. It remains to complement it with alias analysis, escape analysis, and immutability analysis.

- Early lock releasing: Our current implementation does not release ownership of an object until another thread requests it, but any release-point outside a transaction is sound. For contended objects that the owning thread is unlikely to use again soon, an "early release" would improve performance. We believe dynamic information measuring contention will help. Moreover, a thread that knows it holds no locks does not need to poll to see if it must release any.

- Less conservative rollback: Our current implementation rolls back a transaction if a thread releases ownership of an object. Doing so is correct, but unnecessary if the transaction did not actually access the object (i.e., the thread owned the object only due to an earlier access).

- Reader-writer locks: Our current implementation disallows concurrent reads of thread-shared data. Extending the notion of `currentHolder` to allow multiple readers is straightforward in principle, but it remains to investigate if the extra complexity helps or hurts performance. For rarely mutated, highly-contended data it probably helps.

- Other atomicity features: We would like to incorporate "fancy" atomicity features such as the orelse combinator [19] and parallelism within transactions [25, 2].

Availability: Our translator is publicly available. See `http://wasp.cs.washington.edu/wasp_atomjava.html` or contact the authors.

9. ACKNOWLEDGMENTS

Eric Choi and Tim Prouty provided implementation help on log-duplicate detection and standard-library wrappers, respectively. The Polyglot developers provided an excellent tool, and in particular Nathaniel Nystrom provided prompt and thorough assistance. Shaz Qadeer initially suggested implementing atomicity with locks. This work benefited from conversations with Manuel Fähndrich, Jim Larus, Vijay Menon, Leaf Peterson, members of the WASP group at the University of Washington, and many others.

10. 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):207–255, 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*, 2006. To appear.

[3] J. Aldrich, E. G. Sirer, C. Chambers, and S. Eggers. Comprehensive synchronization elimination for Java. *Science of Computer Programming*, 47(2–3):91–120, May–June 2003.

[4] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., and S. Tobin-Hochstadt. The Fortress language specification, version 0.707. http://research.sun.com/projects/plrg/fortress0707.pdf.

[5] 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*, pages 316–327, Feb. 2005.

[6] B. Blanchet. Escape analysis for object-oriented languages: Application to Java. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 20–34, Denver, CO, Nov. 1999.

[7] 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.

[8] J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 35–46, Denver, CO, Nov. 1999.

[9] 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*, pages 211–230, Seattle, WA, Nov. 2002.

[10] 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*, 2006. To appear.

[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 *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, 2005.

[12] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–19, Denver, CO, Nov. 1999.

[13] Cray Inc. Chapel specification 0.4. http://chapel.cs.washington.edu/specification.pdf.

[14] C. Flanagan, S. N. Freund, and M. Lifshin. Type inference for atomicity. In *2005 ACM International Workshop on Types in Language Design and Implementation*, pages 47–58, 2005.

[15] D. Grossman. Software transactions are to concurrency as garbage collection is to memory management, Feb. 2006. Submitted to an informal workshop. Available at http://www.cs.washington.edu/homes/djg/papers.

[16] 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*, pages 1–13, 2004.

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

[18] T. Harris and K. Fraser. Language support for lightweight transactions. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, Oct. 2003.

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

[20] T. Harris, S. Marlow, and S. Peyton Jones. Haskell on a shared-memory multiprocessor. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, pages 49–61, Sept. 2005.

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

[22] 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.

[23] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *32nd ACM Symposium on Principles of Programming Languages*, pages 378–391, Long Beach, CA, Jan. 2005.

[24] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. In *33rd ACM Symposium on Principles of Programming Languages*, pages 346–358, 2006.

[25] J. E. B. Moss and A. L. Hosking. Nested transactional memory: Model and preliminary architecture sketches. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages*, Oct. 2005. https://urresearch.rochester.edu/handle/1802/2099.

[26] 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*, pages 138–152, Warsaw, Poland, Apr. 2003.

[27] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *32nd International Symposium on Computer Architecture*, pages 494–505, June 2005.

[28] M. F. Ringenburg and D. Grossman. AtomCaml: First-class atomicity via rollback. In *10th ACM International Conference on Functional Programming*, pages 92–104, Tallinn, Estonia, Sept. 2005.

[29] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, Special Issue(10):99–116, 1997.

[30] A. Shinnar, D. Tarditi, M. Plesko, and B. Steensgaard. Integrating support for undo with exception handling. Technical Report MSR-TR-2004-140, Microsoft Research, Dec. 2004.

[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*, pages 115–128, 2003.

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