Papers Presented at the 2004 Workshop on Concurrency and Synchronization in Java Programs

St. Johns, Newfoundland, Canada

July 25–26, 2004

In conjunction with:
The ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing

FOREWORD

This volume contains the 11 papers presented at the 2004 Workshop on Concurrency and Synchronization in Java Programs, held in conjunction with ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC) from July 25 to 26, 2004, in St. Johns, Newfoundland, Canada. The goal of this workshop is to promote interaction between the distributed computing and Java™ communities.

The contributed papers were selected from 16 submissions. The submissions were read and evaluated by the program committee, but were not formally refereed; it is expected that many of them will appear in more polished form in fully refereed scientific publication venues. A selection of papers will appear in a special journal issue dedicated to the workshop.

The Program Committee would like to thank all the authors who submitted papers for consideration. We would also like to thank Krishnamurthy Vidyasankar for his generous help with local arrangements for the workshop and in preparing this collection of papers presented at the workshop. We also thank Panagiota Fatourou and Victor Luchangco for their help with publicity, and Marcos Aguilera and Jaap-Henk Hoepman for putting together the submission management software.

Financial support for the workshop was provided by Sun Microsystems Laboratories, and we are grateful to Steve Heller and Rita Tavilla for their help with this.

Program Committee

David Bacon, *IBM*
Hans Boehm, *HP*
Josh Bloch, *Sun Microsystems*
David Detlefs, *Sun Microsystems Laboratories*
Tim Harris, *Cambridge University*
Maurice Herlihy, *Brown University*
David Holmes, *DLTeCH Pty Ltd*
Doug Lea, *SUNY Oswego*
Mark Moir, *Sun Microsystems Laboratories* (Co-Chair)
Vivek Sarkar, *IBM*
Nir Shavit, *Sun Microsystems Laboratories* (Co-Chair)
Martin Rinard, *MIT*
Jan Vitek, *Purdue University*

TABLE OF CONTENTS

Session 1: Tools for Synchronization

The java.util.concurrent Synchronizer Framework
D. Lea 1

Exclusion Control for Java and C#: Experimenting with Granularity of Locks
J. Potter, A. Shanneb, and E. Yu 10

Dynamic Inference of Polymorphic Lock Types
J. Rose, N. Swamy, and M. Hicks 18

Session 2: Memory Models and Formal Methods

Rigorous Concurrency Analysis of Multithreaded Programs
Y. Yang, G. Gopalakrishnan, and G. Lindstrom 26

Requirements for Programming Language Memory Models
J. Manson and W. Pugh 36

Session 3: Software Transactional Memory

Exceptions and side-effects in atomic blocks
T. Harris 46

Transactional Lock-Free Objects for Real-Time Java
F. Pizlo, M. Prochazka, S. Jagannathan, and J. Vitek 54

Snapshots and Software Transactional Memory
C. Cole and M. Herlihy 63

Contention Management in Dynamic Software Transactional Memory
W. Scherer and M. Scott 70

Session 4: Software Engineering

Finding Concurrency Bugs In Java
D. Hovemeyer and W. Pugh 80

Observations on the Assured Evolution of Concurrent Java Programs
A. Greenhouse, T. Halloran, and W. Scherlis 90

Presented Papers

The java.util.concurrent Synchronizer Framework

Doug Lea
SUNY Oswego
Oswego NY 13126
dl@cs.oswego.edu

ABSTRACT

Most synchronizers (locks, barriers, etc.) in the J2SE1.5 java.util.concurrent package are constructed using a small framework based on class `AbstractQueuedSynchronizer`. This framework provides common mechanics for atomically managing synchronization state, blocking and unblocking threads, and queuing. The paper describes the rationale, design, implementation, usage, and performance of this framework.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming – Parallel Programming

General Terms

Algorithms, Measurement, Performance, Design.

Keywords

Synchronization, Java

1. INTRODUCTION

Java[tm] release J2SE-1.5 introduces package java.util.concurrent, a collection of medium-level concurrency support classes created via Java Community Process (JCP) Java Specification Request (JSR) 166. Among these components are a set of *synchronizers* – abstract data type (ADT) classes that maintain an internal *synchronization state* (for example, representing whether a lock is locked or unlocked), operations to update and inspect that state, and at least one method that will cause a calling thread to block if the state requires it, resuming when some other thread changes the synchronization state to permit it. Examples include various forms of mutual exclusion locks, read-write locks, semaphores, barriers, futures, event indicators, and handoff queues.

As is well-known (see e.g., [2]) nearly any synchronizer can be used to implement nearly any other. For example, it is possible to build semaphores from reentrant locks, and vice versa. However, doing so often entails enough complexity, overhead, and inflexibility to be at best a second-rate engineering option. Further, it is conceptually unattractive. If none of these constructs are intrinsically more primitive than the others, developers should not be compelled to arbitrarily choose one of them as a basis for building others. Instead, JSR166 establishes a small framework centered on class `AbstractQueuedSynchronizer`, that provides common mechanics that are used by most of the provided synchronizers in the package, as well as other classes that users may define themselves.

The remainder of this paper discusses the requirements for this framework, the main ideas behind its design and implementation, sample usages, and some measurements showing its performance characteristics.

2. REQUIREMENTS

2.1 Functionality

Synchronizers possess two kinds of methods [7]: at least one *acquire* operation that blocks the calling thread unless/until the synchronization state allows it to proceed, and at least one *release* operation that changes synchronization state in a way that may allow one or more blocked threads to unblock.

The java.util.concurrent package does not define a single unified API for synchronizers. Some are defined via common interfaces (e.g., `Lock`), but others contain only specialized versions. So, *acquire* and *release* operations take a range of names and forms across different classes. For example, methods `Lock.lock`, `Semaphore.acquire`, `CountDownLatch.await`, and `FutureTask.get` all map to *acquire* operations in the framework. However, the package does maintain consistent conventions across classes to support a range of common usage options. When meaningful, each synchronizer supports:

- Nonblocking synchronization attempts (for example, `tryLock`) as well as blocking versions.
- Optional timeouts, so applications can give up waiting.
- Cancellability via interruption, usually separated into one version of acquire that is cancellable, and one that isn't.

Synchronizers may vary according to whether they manage only *exclusive* states – those in which only one thread at a time may continue past a possible blocking point – versus possible *shared* states in which multiple threads can at least sometimes proceed. Regular lock classes of course maintain only exclusive state, but counting semaphores, for example, may be acquired by as many threads as the count permits. To be widely useful, the framework must support both modes of operation.

The java.util.concurrent package also defines interface `Condition`, supporting monitor-style await/signal operations that may be associated with exclusive `Lock` classes, and whose implementations are intrinsically intertwined with their associated `Lock` classes.

2.2 Performance Goals

Java built-in locks (accessed using `synchronized` methods and blocks) have long been a performance concern, and there is a sizable literature on their construction (e.g., [1], [3]). However, the main focus of such work has been on minimizing space overhead (because any Java object can serve as a lock) and on minimizing time overhead when used in mostly-single-threaded

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CSJP'04, July 26, 2004, St John's, Newfoundland, CA.

contexts on uniprocessors. Neither of these are especially important concerns for synchronizers: Programmers construct synchronizers only when needed, so there is no need to compact space that would otherwise be wasted, and synchronizers are used almost exclusively in multithreaded designs (increasingly often on multiprocessors) under which at least occasional contention is to be expected. So the usual JVM strategy of optimizing locks primarily for the zero-contention case, leaving other cases to less predictable "slow paths" [12] is not the right tactic for typical multithreaded server applications that rely heavily on java.util.concurrent.

Instead, the primary performance goal here is *scalability*: to predictably maintain efficiency even, or especially, when synchronizers are contended. Ideally, the overhead required to pass a synchronization point should be constant no matter how many threads are trying to do so. Among the main goals is to minimize the total amount of time during which some thread is permitted to pass a synchronization point but has not done so. However, this must be balanced against resource considerations, including total CPU time requirements, memory traffic, and thread scheduling overhead. For example, spinlocks usually provide shorter acquisition times than blocking locks, but usually waste cycles and generate memory contention, so are not often applicable.

These goals carry across two general styles of use. Most applications should maximize aggregate throughput, tolerating, at best, probabilistic guarantees about lack of starvation. However in applications such as resource control, it is far more important to maintain fairness of access across threads, tolerating poor aggregate throughput. No framework can decide between these conflicting goals on behalf of users; instead different fairness policies must be accommodated.

No matter how well-crafted they are internally, synchronizers will create performance bottlenecks in some applications. Thus, the framework must make it possible to monitor and inspect basic operations to allow users to discover and alleviate bottlenecks. This minimally (and most usefully) entails providing a way to determine how many threads are blocked.

3. DESIGN AND IMPLEMENTATION

The basic ideas behind a synchronizer are quite straightforward. An acquire operation proceeds as:

while (synchronization state does not allow acquire) {
enqueue current thread if not already queued;
possibly block current thread;
}
dequeue current thread if it was queued;

And a release operation is:

update synchronization state;
if (state may permit a blocked thread to acquire)
unblock one or more queued threads;

Support for these operations requires the coordination of three basic components:

- Atomically managing synchronization state
- Blocking and unblocking threads
- Maintaining queues

It might be possible to create a framework that allows each of these three pieces to vary independently. However, this would neither be very efficient nor usable. For example, the information kept in queue nodes must mesh with that needed for unblocking, and the signatures of exported methods depend on the nature of synchronization state.

The central design decision in the synchronizer framework was to choose a concrete implementation of each of these three components, while still permitting a wide range of options in how they are used. This intentionally limits the range of applicability, but provides efficient enough support that there is practically never a reason not to use the framework (and instead build synchronizers from scratch) in those cases where it does apply.

3.1 Synchronization State

Class `AbstractQueuedSynchronizer` maintains synchronization state using only a single (32bit) `int`, and exports `getState`, `setState`, and `compareAndSetState` operations to access and update this state. These methods in turn rely on java.util.concurrent.atomic support providing JSR133 (Java Memory Model) compliant `volatile` semantics on reads and writes, and access to native compare-and-swap or load-linked/store-conditional instructions to implement `compareAndSetState`, that atomically sets state to a given new value only if it holds a given expected value.

Restricting synchronization state to a 32bit `int` was a pragmatic decision. While JSR166 also provides atomic operations on 64bit `long` fields, these must still be emulated using internal locks on enough platforms that the resulting synchronizers would not perform well. In the future, it seems likely that a second base class, specialized for use with 64bit state (i.e., with `long` control arguments), will be added. However, there is not now a compelling reason to include it in the package. Currently, 32 bits suffice for most applications. Only one java.util.concurrent synchronizer class, `CyclicBarrier`, would require more bits to maintain state, so instead uses locks (as do most higher-level utilities in the package).

Concrete classes based on `AbstractQueuedSynchronizer` must define methods `tryAcquire` and `tryRelease` in terms of these exported state methods in order to control the acquire and release operations. The `tryAcquire` method must return `true` if synchronization was acquired, and the `tryRelease` method must return `true` if the new synchronization state may allow future acquires. These methods accept a single `int` argument that can be used to communicate desired state; for example in a reentrant lock, to re-establish the recursion count when re-acquiring the lock after returning from a condition wait. Many synchronizers do not need such an argument, and so just ignore it.

3.2 Blocking

Until JSR166, there was no Java API available to block and unblock threads for purposes of creating synchronizers that are not based on built-in monitors. The only candidates were `Thread.suspend` and `Thread.resume`, which are unusable because they encounter an unsolvable race problem: If an unblocking thread invokes `resume` before the blocking thread has executed `suspend`, the `resume` operation will have no effect.

The java.util.concurrent.locks package includes a `LockSupport` class with methods that address this problem. Method `LockSupport.park` blocks the current thread unless or until a `LockSupport.unpark` has been issued. (Spurious wakeups are also permitted.) Calls to `unpark` are not "counted", so multiple `unparks` before a `park` only unblock a single `park`. Additionally, this applies per-thread, not per-synchronizer. A thread invoking `park` on a new synchronizer might return immediately because of a "leftover" `unpark` from a previous usage. However, in the absence of an `unpark`, its next invocation will block. While it would be possible to explicitly clear this state, it is not worth doing so. It is more efficient to invoke `park` multiple times when it happens to be necessary.

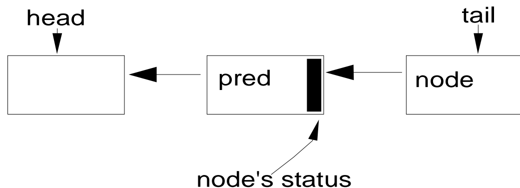
This simple mechanism is similar to those used, at some level, in the Solaris-9 thread library [11], in WIN32 "consumable events", and in the Linux NPTL thread library, and so maps efficiently to each of these on the most common platforms Java runs on. (However, the current Sun Hotspot JVM reference implementation on Solaris and Linux actually uses a pthread condvar in order to fit into the existing runtime design.) The `park` method also supports optional relative and absolute timeouts, and is integrated with JVM `Thread.interrupt` support — interrupting a thread unparks it.

3.3 Queues

The heart of the framework is maintenance of queues of blocked threads, which are restricted here to FIFO queues. Thus, the framework does not support priority-based synchronization.

These days, there is little controversy that the most appropriate choices for synchronization queues are non-blocking data structures that do not themselves need to be constructed using lower-level locks. And of these, there are two main candidates: variants of Mellor-Crummey and Scott (MCS) locks [9], and variants of Craig, Landin, and Hagersten (CLH) locks [5][8][10]. Historically, CLH locks have been used only in spinlocks. However, they appeared more amenable than MCS for use in the synchronizer framework because they are more easily adapted to handle cancellation and timeouts, so were chosen as a basis. The resulting design is far enough removed from the original CLH structure to require explanation.

A CLH queue is not very queue-like, because its enqueuing and dequeuing operations are intimately tied to its uses as a lock. It is a linked queue accessed via two atomically updatable fields, `head` and `tail`, both initially pointing to a dummy node.



A new node, `node`, is *enqueued* using an atomic operation:

```
do {  pred = tail;
 } while(!tail.compareAndSet(pred, node));
```

The release status for each node is kept in its predecessor node. So, the "spin" of a spinlock looks like:

```
while (pred.status != RELEASED) ; // spin
```

A *dequeue* operation after this spin simply entails setting the `head` field to the node that just got the lock:

```
head = node;
```

Among the advantages of CLH locks are that enqueuing and dequeuing are fast, lock-free, and obstruction free (even under contention, one thread will always win an insertion race so will make progress); that detecting whether any threads are waiting is also fast (just check if `head` is the same as `tail`); and that release status is decentralized, avoiding some memory contention.

In the original versions of CLH locks, there were not even links connecting nodes. In a spinlock, the `pred` variable can be held as a local. However, Scott and Scherer[10] showed that by explicitly maintaining predecessor fields within nodes, CLH locks can deal with timeouts and other forms of cancellation: If a node's predecessor cancels, the node can slide up to use the previous node's status field.

The main additional modification needed to use CLH queues for blocking synchronizers is to provide an efficient way for one node to locate its successor. In spinlocks, a node need only change its status, which will be noticed on next spin by its successor, so links are unnecessary. But in a blocking synchronizer, a node needs to explicitly wake up (`unpark`) its successor.

An `AbstractQueuedSynchronizer` queue node contains a `next` link to its successor. But because there are no applicable techniques for lock-free atomic insertion of double-linked list nodes using compareAndSet, this link is not atomically set as part of insertion; it is simply assigned:

```
pred.next = node;
```

after the insertion. This is reflected in all usages. The `next` link is treated only as an optimized path. If a node's successor does not appear to exist (or appears to be cancelled) via its `next` field, it is always possible to start at the tail of the list and traverse backwards using the `pred` field to accurately check if there really is one.

A second set of modifications is to use the status field kept in each node for purposes of controlling blocking, not spinning. In the synchronizer framework, a queued thread can only return from an acquire operation if it passes the `tryAcquire` method defined in a concrete subclass; a single "released" bit does not suffice. But control is still needed to ensure that an active thread is only allowed to invoke `tryAcquire` when it is at the head of the queue; in which case it may fail to acquire, and (re)block. This does not require a per-node status flag because permission can be determined by checking that the current node's predecessor is the `head`. And unlike the case of spinlocks, there is not enough memory contention reading `head` to warrant replication. However, cancellation status must still be present in the status field.

The queue node status field is also used to avoid needless calls to `park` and `unpark`. While these methods are relatively fast as blocking primitives go, they encounter avoidable overhead in the boundary crossing between Java and the JVM runtime and/or OS. Before invoking `park`, a thread sets a "*signal* me" bit, and then rechecks synchronization and node status once more before invoking `park`. A releasing thread clears status. This saves threads from needlessly attempting to block often enough to be worthwhile, especially for lock classes in which lost time waiting for the next eligible thread to acquire a lock accentuates other contention effects. This also avoids requiring a releasing thread to determine its successor unless the successor has set the signal bit, which in turn eliminates those cases where it must traverse

multiple nodes to cope with an apparently null `next` field unless signalling occurs in conjunction with cancellation.

Perhaps the main difference between the variant of CLH locks used in the synchronizer framework and those employed in other languages is that garbage collection is relied on for managing storage reclamation of nodes, which avoids complexity and overhead. However, reliance on GC does still entail nulling of link fields when they are sure to never to be needed. This can normally be done when dequeuing. Otherwise, unused nodes would still be reachable, causing them to be uncollectable.

Some further minor tunings, including lazy initialization of the initial dummy node required by CLH queues upon first contention, are described in the source code documentation in the J2SE1.5 release.

Omitting such details, the general form of the resulting implementation of the basic acquire operation (exclusive, noninterruptible, untimed case only) is:

```
if (!tryAcquire(arg)) {
  node = create and enqueue new node;
  pred = node's effective predecessor;
  while  (pred is not head node ||  !tryAcquire(arg)) {
    if (pred's signal bit is set)
      park();
    else
      compareAndSet pred's signal bit to true;
    pred = node's effective predecessor;
  }
  head = node;
}
```

And the release operation is:

```
if (tryRelease(arg) && head node's signal bit is set) {
   compareAndSet head's signal bit to false;
   unpark head's successor, if one exists
}
```

The number of iterations of the main acquire loop depends, of course, on the nature of `tryAcquire`. Otherwise, in the absence of cancellation, each component of acquire and release is a constant-time $O(1)$ operation, amortized across threads, disregarding any OS thread scheduling occuring within `park`.

Cancellation support mainly entails checking for interrupt or timeout upon each return from `park` inside the acquire loop. A cancelled thread due to timeout or interrupt sets its node status and unparks its successor so it may reset links. With cancellation, determining predecessors and successors and resetting status may include $O(n)$ traversals (where n is the length of the queue). Because a thread never again blocks for a cancelled operation, links and status fields tend to restabilize quickly.

3.4 Condition Queues

The synchronizer framework provides a `ConditionObject` class for use by synchronizers that maintain exclusive synchronization and conform to the `Lock` interface. Any number of condition objects may be attached to a lock object, providing classic monitor-style `await`, `signal`, and `signalAll` operations, including those with timeouts, along with some inspection and monitoring methods.

The `ConditionObject` class enables conditions to be efficiently integrated with other synchronization operations, again by fixing some design decisions. This class supports only Java-style monitor access rules in which condition operations are legal only when the lock owning the condition is held by the current thread (See [4] for discussion of alternatives). Thus, a `ConditionObject` attached to a `ReentrantLock` acts in the same way as do built-in monitors (via `Object.wait` etc), differing only in method names, extra functionality, and the fact that users can declare multiple conditions per lock.

A `ConditionObject` uses the same internal queue nodes as synchronizers, but maintains them on a separate condition queue. The signal operation is implemented as a queue transfer from the condition queue to the lock queue, without necessarily waking up the signalled thread before it has re-acquired its lock.

The basic await operation is:

```
create and add new node to condition queue;
release  lock;
block until node is on lock queue;
re-acquire lock;
```

And the signal operation is:

```
transfer the first node from condition queue to lock queue;
```

Because these operations are performed only when the lock is held, they can use sequential linked queue operations (using a `nextWaiter` field in nodes) to maintain the condition queue. The transfer operation simply unlinks the first node from the condition queue, and then uses CLH insertion to attach it to the lock queue.

The main complication in implementing these operations is dealing with cancellation of condition waits due to timeouts or `Thread.interrupt`. A cancellation and signal occuring at approximately the same time encounter a race whose outcome conforms to the specifications for built-in monitors. As revised in JSR133, these require that if an interrupt occurs before a signal, then the `await` method must, after re-acquiring the lock, throw `InterruptedException`. But if it is interrupted after a signal, then the method must return without throwing an exception, but with its thread interrupt status set.

To maintain proper ordering, a bit in the queue node status records whether the node has been (or is in the process of being) transferred. Both the signalling code and the cancelling code try to compareAndSet this status. If a signal operation loses this race, it instead transfers the next node on the queue, if one exists. If a cancellation loses, it must abort the transfer, and then await lock re-acquisition. This latter case introduces a potentially unbounded spin. A cancelled wait cannot commence lock re-acquisition until the node has been successfully inserted on the lock queue, so must spin waiting for the CLH queue insertion compareAndSet being performed by the signalling thread to succeed. The need to spin here is rare, and employs a `Thread.yield` to provide a scheduling hint that some other thread, ideally the one doing the signal, should instead run. While it would be possible to implement here a helping strategy for the cancellation to insert the node, the case is much too rare to justify the added overhead that this would entail. In all other cases, the basic mechanics here and elsewhere use no spins or yields, which maintains reasonable performance on uniprocessors.

4. USAGE

Class `AbstractQueuedSynchronizer` ties together the above functionality and serves as a "template method pattern" [6] base class for synchronizers. Subclasses define only the methods that implement the state inspections and updates that control acquire and release. However, subclasses of `AbstractQueuedSynchronizer` are not themselves usable as synchronizer ADTs, because the class necessarily exports the methods needed to internally control acquire and release policies, which should not be made visible to users of these classes. All java.util.concurrent synchronizer classes declare a private inner `AbstractQueuedSynchronizer` subclass and delegate all synchronization methods to it. This also allows public methods to be given names appropriate to the synchronizer.

For example, here is a minimal `Mutex` class, that uses synchronization state zero to mean unlocked, and one to mean locked. This class does not need the value arguments supported for synchronization methods, so uses zero, and otherwise ignores them.

```
class Mutex {
  class Sync
    extends AbstractQueuedSynchronizer {
    public boolean tryAcquire(int ignore) {
      return compareAndSetState(0, 1);
    }
    public boolean tryRelease(int ignore) {
       setState(0); return true;
    }
  }
  private final Sync sync = new Sync();
  public void lock()   { sync.acquire(0); }
  public void unlock() { sync.release(0); }
}
```

A fuller version of this example, along with other usage guidance may be found in the J2SE documentation. Many variants are of course possible. For example, `tryAcquire` could employ "test-and-test-and-set" by checking the state value before trying to change it.

It may be surprising that a construct as performance-sensitive as a mutual exclusion lock is intended to be defined using a combination of delegation and virtual methods. However, these are the sorts of OO design constructions that modern dynamic compilers have long focussed on. They tend to be good at optimizing away this overhead, at least in code in which synchronizers are invoked frequently.

Class `AbstractQueuedSynchronizer` also supplies a number of methods that assist synchronizer classes in policy control. For example, it includes timeout and interruptible versions of the basic acquire method. And while discussion so far has focussed on exclusive-mode synchronizers such as locks, the `AbstractQueuedSynchronizer` class also contains a parallel set of methods (such as `acquireShared`) that differ in that the `tryAcquireShared` and `tryReleaseShared` methods can inform the framework (via their return values) that further acquires may be possible, ultimately causing it to wake up multiple threads by cascading signals.

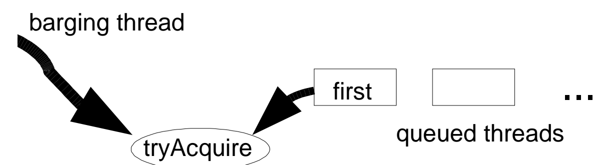
Although it is not usually sensible to serialize (persistently store or transmit) a synchronizer, these classes are often used in turn to construct other classes, such as thread-safe collections, that are commonly serialized. The `AbstractQueuedSynchronizer` and `ConditionObject` classes provide methods to serialize synchronization state, but not the underlying blocked threads or other intrinsically transient bookkeeping. Even so, most synchronizer classes merely reset synchronization state to initial values on deserialization, in keeping with the implicit policy of built-in locks of always deserializing to an unlocked state. This amounts to a no-op, but must still be explicitly supported to enable deserialization of `final` fields.

4.1 Controlling Fairness

Even though they are based on FIFO queues, synchronizers are not necessarily fair. Notice that in the basic acquire algorithm (Section 3.3), `tryAcquire` checks are performed *before* queuing. Thus a newly acquiring thread can “steal” access that is "intended" for the first thread at the head of the queue.

This *barging FIFO* strategy generally provides higher aggregate throughput than other techniques. It reduces the time during which a contended lock is available but no thread has it because the intended next thread is in the process of unblocking. At the same time, it avoids excessive, unproductive contention by only allowing one (the first) queued thread to wake up and try to acquire upon any release. Developers creating synchronizers may further accentuate barging effects in cases where synchronizers are expected to be held only briefly by defining `tryAcquire` to itself retry a few times before passing back control.

Barging FIFO synchronizers have only probablistic fairness properties. An unparked thread at the head of the lock queue has



an unbiased chance of winning a race with any incoming barging thread, reblocking and retrying if it loses. However, if incoming threads arrive faster than it takes an unparked thread to unblock, the first thread in the queue will only rarely win the race, so will almost always reblock, and its successors will remain blocked. With briefly-held synchronizers, it is common for multiple bargings and releases to occur on multiprocessors during the time the first thread takes to unblock. As seen below, the net effect is to maintain high rates of progress of one or more threads while still at least probabilistically avoiding starvation.

When greater fairness is required, it is a relatively simple matter to arrange it. Programmers requiring strict fairness can define `tryAcquire` to fail (return false) if the current thread is not at the head of the queue, checking for this using method `getFirstQueuedThread`, one of a handful of supplied inspection methods.

A faster, less strict variant is to also allow `tryAcquire` to succeed if the the queue is (momentarily) empty. In this case, multiple threads encountering an empty queue may race to be the first to acquire, normally without enqueuing at least one of them. This strategy is adopted in all java.util.concurrent synchronizers supporting a "fair" mode.

While they tend to be useful in practice, fairness settings have no guarantees, because the Java Language Specification does not provide scheduling guarantees. For example, even with a strictly fair synchronizer, a JVM could decide to run a set of threads purely sequentially if they never otherwise need to block waiting for each other. In practice, on a uniprocessor, such threads are

likely to each run for a time quantum before being pre-emptively context-switched. If such a thread is holding an exclusive lock, it will soon be momentarily switched back, only to release the lock and block now that it is known that another thread needs the lock, thus further increasing the periods during which a synchronizer is available but not acquired. Synchronizer fairness settings tend to have even greater impact on multiprocessors, which generate more interleavings, and hence more opportunities for one thread to discover that a lock is needed by another thread.

Even though they may perform poorly under high contention when protecting briefly-held code bodies, fair locks work well, for example, when they protect relatively long code bodies and/or with relatively long inter-lock intervals, in which case barging provides little performance advantage and but greater risk of indefinite postponement. The synchronizer framework leaves such engineering decisions to its users.

4.2 Synchronizers

Here are sketches of how java.util.concurrent synchronizer classes are defined using this framework:

The `ReentrantLock` class uses synchronization state to hold the (recursive) lock count. When a lock is acquired, it also records the identity of the current thread to check recursions and detect illegal state exceptions when the wrong thread tries to unlock. The class also uses the provided `ConditionObject`, and exports other monitoring and inspection methods. The class supports an optional "fair" mode by internally declaring two different `AbstractQueuedSynchronizer` subclasses (the fair one disabling barging) and setting each `ReentrantLock` instance to use the appropriate one upon construction.

The `ReentrantReadWriteLock` class uses 16 bits of the synchronization state to hold the write lock count, and the remaining 16 bits to hold the read lock count. The `WriteLock` is otherwise structured in the same way as `ReentrantLock`. The `ReadLock` uses the `acquireShared` methods to enable multiple readers.

The `Semaphore` class (a counting semaphore) uses the synchronization state to hold the current count. It defines `acquireShared` to decrement the count or block if nonpositive, and `tryRelease` to increment the count, possibly unblocking threads if it is now positive.

The `CountDownLatch` class uses the synchronization state to represent the count. All acquires pass when it reaches zero.

The `FutureTask` class uses the synchronization state to represent the run-state of a future (initial, running, cancelled, done). Setting or cancelling a future invokes `release`, unblocking threads waiting for its computed value via `acquire`.

The `SynchronousQueue` class (a CSP-style handoff) uses internal wait-nodes that match up producers and consumers. It uses the synchronization state to allow a producer to proceed when a consumer takes the item, and vice-versa.

Users of the java.util.concurrent package may of course define their own synchronizers for custom applications. For example, among those that were considered but not adopted in the package are classes providing the semantics of various flavors of WIN32 events, binary latches, centrally managed locks, and tree-based barriers.

5. PERFORMANCE

While the synchronizer framework supports many other styles of synchronization in addition to mutual exclusion locks, lock performance is simplest to measure and compare. Even so, there are many different approaches to measurement. The experiments here are designed to reveal overhead and throughput.

In each test, each thread repeatedly updates a pseudo-random number computed using function `nextRandom(int seed)`:

```
int t = (seed % 127773) * 16807 –
        (seed / 127773) * 2836;
return (t > 0)? t : t + 0x7fffffff;
```

On each iteration a thread updates, with probability *S*, a shared generator under a mutual exclusion lock, else it updates its own local generator, without a lock. This results in short-duration locked regions, minimizing extraneous effects when threads are preempted while holding locks. The randomness of the function serves two purposes: it is used in deciding whether to lock or not (it is a good enough generator for current purposes), and also makes code within loops impossible to trivially optimize away.

Four kinds of locks were compared: *Builtin*, using synchronized blocks; *Mutex*, using a simple `Mutex` class like that illustrated in section 4; *Reentrant*, using `ReentrantLock;` and *Fair*, using `ReentrantLock` set in its "fair" mode. All tests used build 46 (approximately the same as beta2) of the Sun J2SE1.5 JDK in "server" mode. Test programs performed 20 uncontended runs before collecting measurements, to eliminate warm-up effects. Tests ran for ten million iterations per thread, except Fair mode tests were run only one million iterations.

Tests were performed on four x86-based machines and four UltraSparc-based machines. All x86 machines were running Linux using a RedHat NPTL-based 2.4 kernel and libraries. All UltraSparc machines were running Solaris-9. All systems were at most lightly loaded while testing. The nature of the tests did not demand that they be otherwise completely idle. The "4P" name reflects the fact a dual hyperthreaded (HT) Xeon acts more like a 4-way than a 2-way machine. No attempt was made to normalize across the differences here. As seen below, the relative costs of synchronization do not bear a simple relationship to numbers of processors, their types, or speeds.

Table 1 Test Platforms

Name	*Processors*	*Type*	*Speed (Mhz)*
1P	1	Pentium3	900
2P	2	Pentium3	1400
2A	2	Athlon	2000
4P	2 HT	Pentium4/Xeon	2400
1U	1	UltraSparc2	650
4U	4	UltraSparc2	450
8U	8	UltraSparc3	750
24U	24	UltraSparc3	750

5.1 Overhead

Uncontended overhead was measured by running only one thread, subtracting the time per iteration taken with a version setting S=0 (zero probability of accessing shared random) from a run with S=1. Table 2 displays these estimates of the per-lock overhead of synchronized code over unsynchronized code, in

nanoseconds. The Mutex class comes closest to testing the basic cost of the framework. The additional overhead for Reentrant locks indicates the cost of recording the current owner thread and of error-checking, and for Fair locks the additional cost of first checking whether the queue is empty.

Table 2 also shows the cost of `tryAcquire` versus the "fast path" of a built-in lock. Differences here mostly reflect the costs of using different atomic instructions and memory barriers across locks and machines. On multiprocessors, these instructions tend to completely overwhelm all others. The main differences between Builtin and synchronizer classes are apparently due to Hotspot locks using a compareAndSet for both locking and unlocking, while these synchronizers use a compareAndSet for acquire and a `volatile` write (i.e., with a memory barrier on multiprocessors, and reordering constraints on all processors) on release. The absolute and relative costs of each vary across machines.

At the other extreme, Table 3 shows per-lock overheads with S=1 and running 256 concurrent threads, creating massive lock contention. Under complete saturation, barging-FIFO locks have about an order of magnitude less overhead (and equivalently greater throughput) than Builtin locks, and often two orders of magnitude less than Fair locks. This demonstrates the effectiveness of the barging-FIFO policy in maintaining thread progress even under extreme contention.

Table 2 Uncontended Per-Lock Overhead in Nanoseconds

Machine	*Builtin*	*Mutex*	*Reentrant*	*Fair*
1P	18	9	31	37
2P	58	71	77	81
2A	13	21	31	30
4P	116	95	109	117
1U	90	40	58	67
4U	122	82	100	115
8U	160	83	103	123
24U	161	84	108	119

Table 3 Saturated Per-Lock Overhead in Nanoseconds

Machine	*Builtin*	*Mutex*	*Reentrant*	*Fair*
1P	521	46	67	8327
2P	930	108	132	14967
2A	748	79	84	33910
4P	1146	188	247	15328
1U	879	153	177	41394
4U	2590	347	368	30004
8U	1274	157	174	31084
24U	1983	160	182	32291

Table 3 also illustrates that even with low internal overhead, context switching time completely determines performance for Fair locks. The listed times are roughly proportional to those for blocking and unblocking threads on the various platforms.

Additionally, a follow-up experiment (using machine 4P only) shows that with the very briefly held locks used here, fairness settings had only a small impact on overall variance. Differences in termination times of threads were recorded as a coarse-grained measure of variability. Times on machine 4P had standard deviation of 0.7% of mean for Fair, and 6.0% for Reentrant. As a contrast, to simulate long-held locks, a version of the test was run in which each thread computed 16K random numbers while holding each lock. Here, total run times were nearly identical (9.79s for Fair, 9.72s for Reentrant). Fair mode variability remained small, with standard deviation of 0.1% of mean, while Reentrant rose to 29.5% of mean.

5.2 Throughput

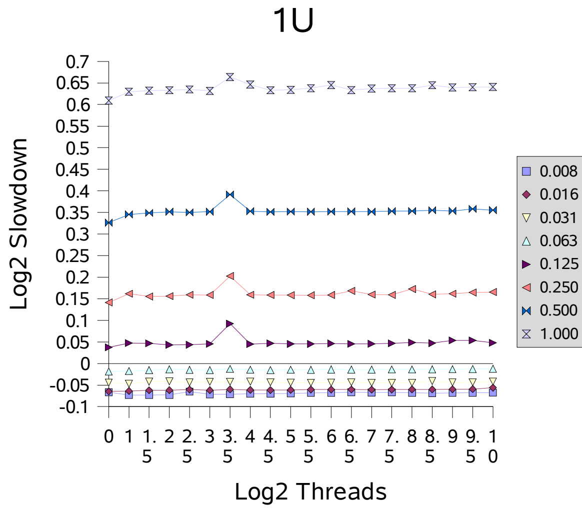
Usage of most synchronizers will range between the extremes of no contention and saturation. This can be experimentally examined along two dimensions, by altering the contention probability of a fixed set of threads, and/or by adding more threads to a set with a fixed contention probability. To illustrate these effects, tests were run with different contention probablilities and numbers of threads, all using Reentrant locks. The accompanying figures use a *slowdown* metric:

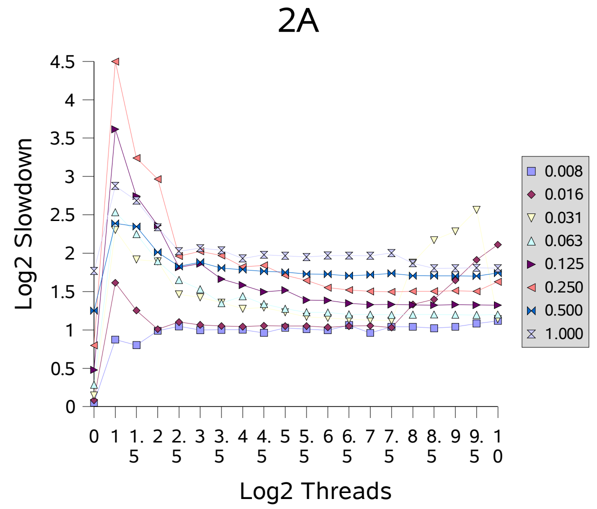
$$slowdown = \frac{t}{S \cdot b \cdot n + (1-S) \cdot b \cdot max(1, \frac{n}{p})}$$

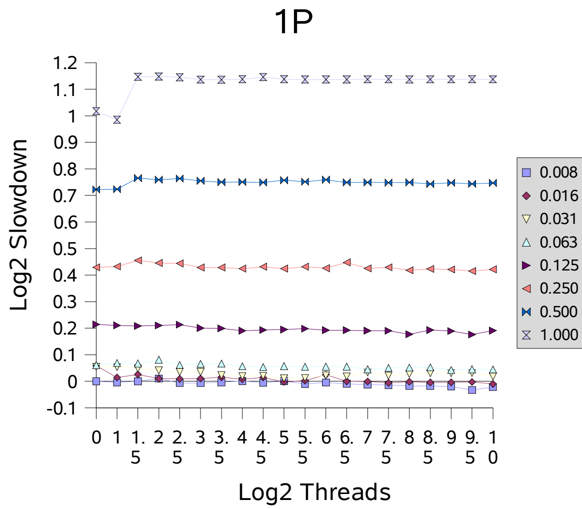
Here, *t* is the total observed execution time, *b* is the baseline time for one thread with no contention or synchronization, *n* is the number of threads, *p* is the number of processors, and S remains the proportion of shared accesses. This value is the ratio of observed time to the (generally unattainable) ideal execution time as computed using Amdahl's law for a mix of sequential and parallel tasks. The ideal time models an execution in which, without any synchronization overhead, no thread blocks due to conflicts with any other. Even so, under very low contention, a few test results displayed very small speedups compared to this ideal, presumably due to slight differences in optimization, pipelining, etc., across baseline versus test runs.

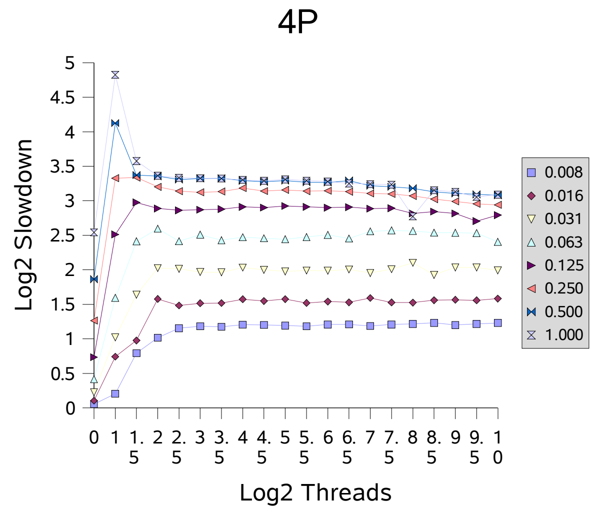
The figures use a base 2 log scale. For example, a value of 1.0 means that a measured time was twice as long as ideally possible, and a value of 4.0 means 16 times slower. Use of logs ameliorates reliance on an arbitrary base time (here, the time to compute random numbers), so results with different base computations should show similar trends. The tests used contention probabilities from 1/128 (labelled as "0.008") to 1, stepping in powers of 2, and numbers of threads from 1 to 1024, stepping in half-powers of 2.

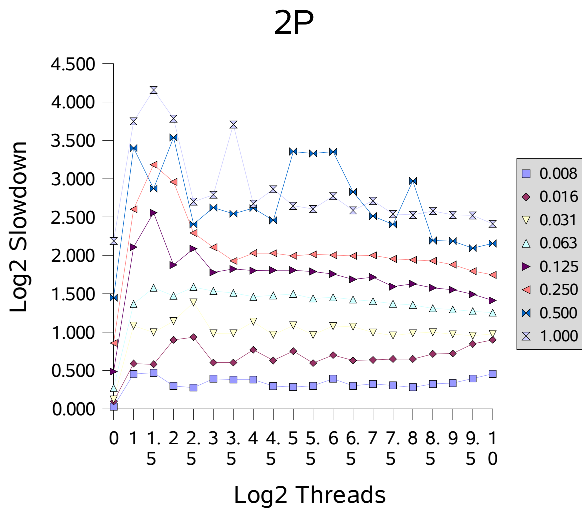
On uniprocessors (1P and 1U) performance degrades with increasing contention, but generally not with increasing numbers of threads. Multiprocessors generally encounter much worse slowdowns under contention. The graphs for multiprocessors show an early peak in which contention involving only a few threads usually produces the worst relative performance. This reflects a transitional region of performance, in which barging and signalled threads are about equally likely to obtain locks, thus frequently forcing each other to block. In most cases, this is followed by a smoother region, as the locks are almost never available, causing access to resemble the near-sequential pattern of uniprocessors; approaching this sooner on machines with more processors. Notice for example that the values for full contention (labelled "1.000") exhibit relatively worse slowdowns on machines with fewer processors.

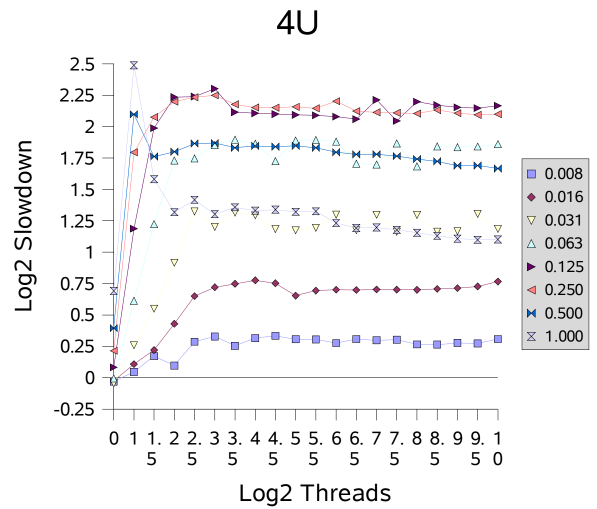

1U
Log2 Slowdown
Log2 Threads
0.008
0.016
0.031
0.063
0.125
0.250
0.500
1.000

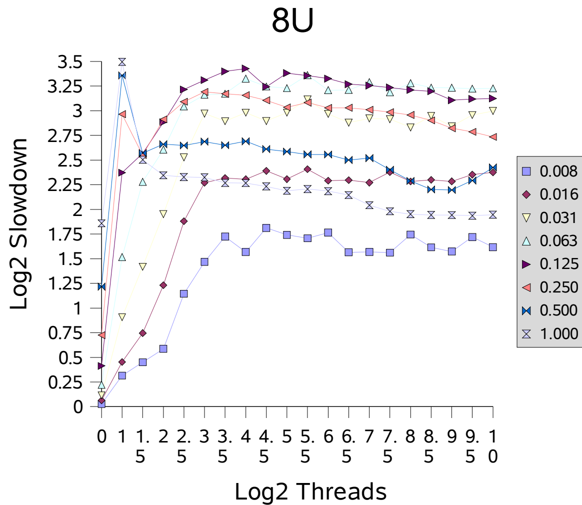

2A
Log2 Slowdown
Log2 Threads
0.008
0.016
0.031
0.063
0.125
0.250
0.500
1.000

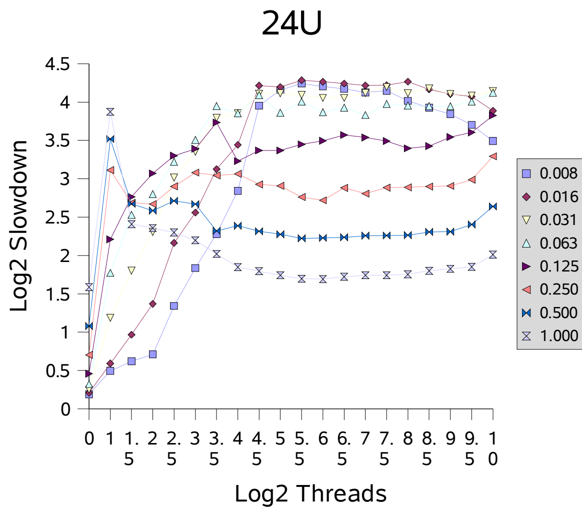

1P
Log2 Slowdown
Log2 Threads
0.008
0.016
0.031
0.063
0.125
0.250
0.500
1.000


4P
Log2 Slowdown
Log2 Threads
0.008
0.016
0.031
0.063
0.125
0.250
0.500
1.000


2P
Log2 Slowdown
Log2 Threads
0.008
0.016
0.031
0.063
0.125
0.250
0.500
1.000


4U
Log2 Slowdown
Log2 Threads
0.008
0.016
0.031
0.063
0.125
0.250
0.500
1.000





On the basis of these results, it appears likely that further tuning of blocking (`park/unpark`) support to reduce context switching and related overhead could provide small but noticeable improvements in this framework. Additionally, it may pay off for synchronizer classes to employ some form of adaptive spinning for briefly-held highly-contended locks on multiprocessors, to avoid some of the flailing seen here. While adaptive spins tend to be very difficult to make work well across different contexts, it is possible to build custom forms of locks using this framework, targetted for specific applications that encounter these kinds of usage profiles.

6. CONCLUSIONS

As of this writing, the java.util.concurrent synchronizer framework is too new to evaluate in practice. It is unlikely to see widespread usage until well after final release of J2SE1.5, and there will surely be unexpected consequences of its design, API, implementation, and performance. However, at this point, the framework appears successful in meeting the goals of providing an efficient basis for creating new synchronizers.

7. ACKNOWLEDGMENTS

Thanks to Dave Dice for countless ideas and advice during the development of this framework, to Mark Moir and Michael Scott for urging consideration of CLH queues, to David Holmes for critiquing early versions of the code and API, to Victor Luchangco and Bill Scherer for reviewing previous incarnations of the source code, and to the other members of the JSR166 Expert Group (Joe Bowbeer, Josh Bloch, Brian Goetz, David Holmes, and Tim Peierls) as well as Bill Pugh, for helping with design and specifications and commenting on drafts of this paper. Portions of this work were made possible by a DARPA PCES grant, NSF grant EIA-0080206 (for access to the 24way Sparc) and a Sun Collaborative Research Grant.

8. REFERENCES

[1] Agesen, O., D. Detlefs, A. Garthwaite, R. Knippel, Y. S. Ramakrishna, and D. White. An Efficient Meta-lock for Implementing Ubiquitous Synchronization. *ACM OOPSLA Proceedings,* 1999.

[2] Andrews, G. *Concurrent Programming*. Wiley, 1991.

[3] Bacon, D. Thin Locks: Featherweight Synchronization for Java. *ACM PLDI Proceedings*, 1998.

[4] Buhr, P. M. Fortier, and M. Coffin. Monitor Classification, *ACM Computing Surveys*, March 1995.

[5] Craig, T. S. Building FIFO and priority-queueing spin locks from atomic swap. Technical Report TR 93-02-02, Department of Computer Science, University of Washington, Feb. 1993.

[6] Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*, Addison Wesley, 1996.

[7] Holmes, D. *Synchronisation Rings*, PhD Thesis, Macquarie University, 1999.

[8] Magnussen, P., A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. *8th Intl. Parallel Processing Symposium*, Cancun, Mexico, Apr. 1994.

[9] Mellor-Crummey, J.M., and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. on Computer Systems*, February 1991

[10] M. L. Scott and W N. Scherer III. Scalable Queue-Based Spin Locks with Timeout. *8th ACM Symp. on Principles and Practice of Parallel Programming*, Snowbird, UT, June 2001.

[11] Sun Microsystems. *Multithreading in the Solaris Operating Environment*. White paper available at http://wwws.sun.com/software/solaris/whitepapers.html 2002.

[12] Zhang, H., S. Liang, and L. Bak. *Monitor Conversion in a Multithreaded Computer System*. United States Patent 6,691,304. 2004.

Exclusion Control for Java and C#: Experimenting with Granularity of Locks

John Potter
potter@cse.unsw.edu.au

Abdelsalam Shanneb
shanneba@cse.unsw.edu.au

Eric Yu
ericyu02@alumni.cse.unsw.edu.au

School of Computer Science and Engineering
The University of New South Wales
Sydney – Australia

ABSTRACT

We present experimental results on how the granularity of locking provided by synchronization wrappers affects performance. The wrappers implement method-level synchronization whose exclusion behavior is specified using a declarative approach. Our experimental results with both Java and C# suggest that fine-grained exclusion control is worthwhile implementing, at least for heavily loaded objects, and that there appears to be no performance penalty in doing so.

To overcome resistance to programming more complex and fragile synchronization code, we propose generating the code automatically, based on the exclusion specification. Essentially this only entails describing the read-write conflicts between method pairs in an object's interface. The automatic code generation can either be done dynamically using existing reflection mechanisms, or statically by a compiler. In the latter case we envisage minor language extensions to allow programmers to specify both the exclusion requirements for a given object, and the exclusion control to be provided by a given object when it is constructed.

Keywords

Concurrency control, Java, C#, mutual exclusion, read-write locks, synchronization, decorators, wrappers, algebra of exclusion, locking granularity.

1. INTRODUCTION

With increasing use of concurrency in applications software, there is a need to simplify the task of programmers in writing correct and efficient code for multi-threaded applications. We believe that a sensible approach to this problem is to provide the programmer with a variety of concurrency models, together with supporting specification mechanisms that encourage a separation between the functional code of the system, and the concurrency management. Where there is a dependency between the two, that is, between function and control, we would like to see automated tool support for generating code that is guaranteed to implement the specified concurrency model.

This paper explores this idea in the context of synchronization wrappers based on method-level exclusion control. Exclusion control is probably the easiest synchronization control to deal with. Nevertheless, it still remains a pain for programmers to implement the code for more sophisticated exclusion control. Typically such code will be fragile with respect to both interface extensions, and internal data representation and sharing; this fragility manifests itself, for example, in the inheritance anomaly, but is also apparent in wrapper-based synchronization: when new methods interfere with existing methods, synchronization wrappers will need to be modified, and not just simply extended (except for the simplest form of synchronization wrapper based on complete mutual exclusion of method calls).

For our work we rely on the algebra of exclusion [8] to express the synchronization specification for a given class. This allows programmers to escape from the tedium of managing the synchronization code implementing the requirements. Given an exclusion expression, and the interface for the class, it is easy to automatically generate a corresponding synchronization wrapper. Whenever classes are modified, the corresponding exclusion expression will need to be re-specified, and the wrapper re-generated. A wrapper may provide more exclusion than that required, but should not provide less. In practice, most programmers simply choose to implement a mutex to achieve thread-safety by locking out all but one thread. Our goal is to make it easy for programmers to specify finer-grained exclusion, without the tedium of managing the locking code.

In line with this goal, we have designed a single general-purpose lock that can be customized to provide precisely the behavior specified by the exclusion expression; the general-purpose lock employs a simple bitset implementation of the current lock-state, and admits methods according to the admissibility of the consequent lock-state; admissibility is specified by the exclusion expression. For such a lock to be useful in providing programmer-specified synchronization wrappers there needs to be some language level mechanism for mapping between a method index, as used in the bitset, and method calls: method reification is one heavy-handed possibility; a second is code generation. We defer further discussion of language or tools support to the end of the paper.

In this paper we conduct experiments exploring the effect on performance of varying the granularity of exclusion-based synchronization control. The purpose of this study is to gain some insight into the effect of different controls, and whether it is worth the trouble of providing programmers with extra support for implementing exclusion controls with different granularity. So we have focused on a typical application environment, with a uniprocessor; furthermore, because we are attempting to explore the effect of control granularity, we only compare like with like. Our experiments are designed to explore the *relative* differences between different exclusion policies, using the same general-purpose lock described above, and *not absolute* performance measures.

The experimental framework is akin to an open queuing model as discussed in Kleinrock[6]. Queuing theory predicts a linear increase of throughput with load until saturation is reached, after which throughput

remains constant with increasing load. Roughly speaking, our results are in accord with theory, with some caveats to be discussed later. We conducted our experiments both with Java and C#. The Java-based experiments were conducted with Sun's 1.3 JVM. The C#-based experiments showed some irregularities initially, so we conducted more extensive experiments that allowed us to partition the data into two more regular sets of observations. In the end we have produced experimental results for Java and C# that provide us not only with enough similarities to draw some general conclusions, but also with some differences that may bear further investigation.

The paper is organized as follows. The next section discusses related work by others and our own broader research objectives. Section 3 introduces the concept of exclusion expressions, and the algebra of exclusion. Section 4 discusses how to generate synchronization wrappers automatically given an exclusion expression. Section 5 details our experimental set-up. The results of our experiments are summarized next: Section 6 deals with the Java-based experiment, and Section 7 with C#: Section 7 goes into more detail discussing how we had to partition our observations to explain irregularities in the C# experiments. Section 8 summarizes the similarities and differences between the Java and C# experiments. Section 9 discusses the overall outcomes of the experiments, together with some further comments on other effects we observed in varying the granularity of exclusion control. A brief summary concludes the paper.

2. RELATED WORK

Software wrappers, or decorators, are a general-purpose mechanism for adding new functionality or for restricting access to existing functionality in the embedded software. The technique has been applied in areas as diverse as security and privacy [2] and database access. Various techniques have been used, including pre-processors and class libraries.

Using wrappers to separate concurrency control from the behavior of underlying object is standard practice now. With JDK 1.2, the Java Collections Framework addressed thread-safety by providing unsynchronized base classes and factory methods for generating synchronized wrappers. These enforce single threading within the wrapped object. Lea's book [7] reveals the variety of approaches for designing and implementing concurrent programs in Java; it offers many techniques and patterns for letting threads work together safely. JSR166 [5], following on from the Java Concurrency Package of JDK 1.5, builds on the work of Doug Lea and others to offer a new concurrency primitives and a set of standardized concurrency utilities, intended to simplify the development of concurrent applications. Our work could provide a small addition to this new package.

Philippsen [9] gives a comprehensive comparison survey of concurrent object-oriented languages in terms of identifying the key areas of integration as well as differences between the object-oriented and concurrent programming paradigms. In terms of performance comparisons between concurrent object-oriented systems based on locking granularity, we have found little evidence of published work, whereas much work investigating locking granularity is evident in the area of database systems [10] [11].

The emergence of the .NET programming suite with the accompanying new languages, opened the door for more comparative studies. Nebro et al [1] present their experience when porting a Java-based runtime system (JACO) to the .NET platform under the programming languages C# and J#. Their comparison highlights some strengths and weaknesses in both platforms. Our work presented here walks along the same line and provides a customizable exclusion mechanism which sits comfortably on either a JVM or the .NET CLR.

The work on exclusion controls and the experimental study reported here forms part of an ongoing research program dealing with concurrency and objects. A recurring theme of much of this work is the separation of function and control. David Holmes et al [3] proposed the notion of dividing the synchronization control aspect of object systems into three further aspects, the exclusion aspect, the state-based aspect and the coordination aspect. Noble et al [8] devised the Algebra of Exclusion as a means for simply describing the requirements of the exclusion aspect of synchronization. In his doctoral thesis, Holmes [4] introduced the synchronization rings model for reusing synchronization controls in wrapping layers. One of his conclusions was that the execution cost of layering and extra complexity associated with queue management for attempted optimizations, such as implementing the specific notification mechanism, was prohibitive when compared with direct implementation of controls. Our hope remains that layering controls is still the right way to view the specification of such systems, irrespective of their implementation. However if there is a mismatch between the layers of specification and layers of implementation, correctness of the implementation becomes more of an issue.

Zhang [12] is considering this problem in his doctoral work, and has produced a variant of the π-calculus with locking semantics, together with a new equivalence for reasoning about compositional method-based control. The implementation of exclusion control in a single lock can be seen as a simple example of the potential for optimization: for example, four read-write sets could be implemented as four separate read-write locks, with appropriate coordination, or as four read-write synchronization rings, or, as in our implementation, via a single table-based exclusion lock.

3. ALGEBRA OF EXCLUSION

Noble et al [8] introduce the Algebra of Exclusion for specifying method-level exclusion. The algebra is intended for reasoning about the synchronization requirements of objects, including composites, where there may be some choice about whether controls should be implemented internally or externally. We focus on external control for a single object here.

The algebra provides a way of expressing different levels of exclusion control for any object; there is a finest grain of control that guarantees the object to be thread-safe. The algebra can model exclusion requirements, such as those based on a methods' read-write access to its fields. The following figure depicts an example of an object with three methods m_1, m_2, and m_3 and two fields v_1 and v_2.

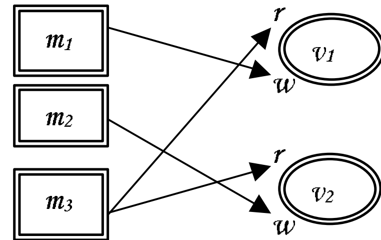


Figure 1: Read-Write Dependency

In this example method m_1 writes to the field v_1, m_2 writes to v_2, and m_3 reads from both v_1 and v_2, as shown in the following table.

	v1		v2	
	r	w	r	w
m1		✓		
m2				✓
m3	✓		✓	

Figure 2: Read-Write Dependency Table

For this example, m_1 and m_2 are independent writers, with m_3 a reader interfering with both; this can be represented by the exclusion expression $\overline{m_1}\, m_3 \mid \overline{m_2}\, m_3$ as explained below. This abstracts away from the explicit dependence on the fields, reflecting the synchronization requirements at the interface of the object.

The algebra can define different expressions that correspond to different levels of exclusion control. These expressions are formulated using two main operators; the disjunction operator $a|b$ which means that method a and b can execute concurrently, and the mutual exclusion operator $a\times b$ which means that a and b conflict (that is, form an exclusion pair). We use $\overline{a}$ to indicate the self-exclusion $a\times a$. Not only do exclusion expressions denote a set of exclusion pairs, but they also define all names in the interface; for example, both $\overline{a}$ and $\overline{a} \mid b$ denote the same self-exclusion on a but the latter is for an extended interface that includes b, that is independent of a.

Assuming an object has three methods a, b and c, the algebra can describe different levels of exclusion control. For convenience we omit the operators and infer them as follows: a list of names is implicitly a disjunct. Such a list followed by a self-exclusion expression has a mutual exclusion operator inferred. This is clarified in the following simple examples. The expression $\overline{abc}$ is a mutex on all the methods; it is equivalent to $\overline{a|b|c}$ and also to $\overline{a\times b\times c}$. The expression $a\ \overline{bc}$ (equivalently $a \times\ \overline{b|c}$) represents a read-write lock with a as the reader and b and c as writers. A finer grain control has c as an independent writer, which may execute concurrently with either of the others: $a\ \overline{b} \mid \overline{c}$ (equivalently $a \times \overline{b} \mid \overline{c}$); this exclusion represents the combination of a read-write lock on a and b, with a mutex on c. The finest grain control is vacuous, allowing all methods to execute concurrently; we denote this simply as abc (equivalently $a|b|c$).

Exclusion expressions can be presented in a conflict matrix format. For example, the read-write expression $ac\ \overline{b}$ is represented by the following bit matrix:

	a	***b***	***c***
a	0	1	0
b	1	1	1
c	0	1	0

4. EXCLUSION WRAPPERS

Having introduced the algebra of exclusion, we show how our model incorporates these control expressions into wrappers to enforce the desired exclusion control. The main goal of the model is to keep the original code separate from control code. This is achieved by wrapping each method with a lock defined the exclusion control expression, as suggested in Figure 3.

The exclusion lock code is customized according to the exclusion expression. These customized locks are used in defining the synchronization wrapper for the object.

We now consider some code excerpts to illustrate the synchronization wrapper. There are no surprises here. The SafeData class wraps an object of class Data with entry and exit components to enforce the exclusion control on the core object. For example, given:

```
class Data {
  public void method1( ) { … }
  public void method2( ) { … }
}
```

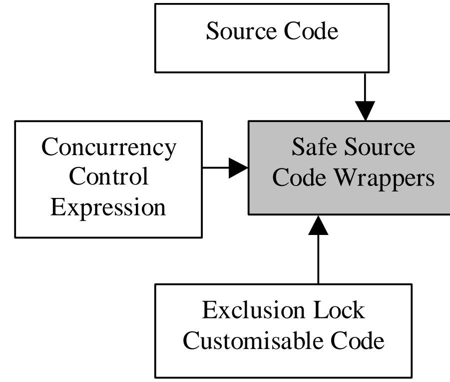


Figure 3: Exclusion Wrappers

The corresponding SafeData wrapper would look like this:

```
class SafeData{
  Data data;
  ExclusionLock lock;
  public void method1( ) {
    lock.enter(method1ID);
    data.method1( );
    lock.exit(method1ID);
  }
  public void method2( ) {
    lock.enter(method2ID);
    data.method2( );
    lock.exit(method2ID);
  }
}
```

The `enter` method tests if the base method specified by the method identifier is ready for execution, that is, this method is allowed now for "real" execution after possible blocking by other methods. If the base method is ready, the `grab` method sets the corresponding bit array in the conflict matrix.

```
synchronized public void enter (int methodID) {
                              // method invocation time
  try {
    while (!ready(methodID) wait();
    grab(methodID)          // method run time
  } catch (InterruptedException e) { }
}
```

The `exit` method simply releases the lock by updating the conflict matrix and then notifying all waiting threads.

```
synchronized public void exit(int methodID) {
  release(methodID);       // method end time
  notifyAll();
}
```

To summarize, given an exclusion expression for an object, the system defines a control over that object. This gives the programmer the choice of choosing different levels of exclusion controls for an object, just by varying the exclusion expression. The separation of object and concurrency control code is clear: synchronization code is added to an object via a standardized wrapper with a customized exclusion lock, and the code of the original class remains unmodified.

5. EXPERIMENTAL DESIGN

The goal of our experiments is to test the effect of varying the granularity of exclusion control. We performed a number of simulation experiments whose results we present in the next sections.

Our experiments are based on a simple object with eight identical

methods, each implementing a busy loop to provide a fixed duration for method execution. We investigate the response (in particular, throughput) for different granularities of exclusion control: full mutual exclusion; two independent read-write sets; four independent read-write sets; and full concurrency (no exclusion). In all cases, we use the same exclusion locking code: only the entries of the conflict matrix embedded within the exclusion lock change between experiments.

To vary the load on the controlled object in a predictable manner, we issue periodic method calls, each serviced by one thread drawn from a large thread-pool. The role of the thread pool is to isolate the method call generation from any blocking by the controlled object. We are thus able to set the interarrival rate for method calls experimentally. This experimental design is geared towards measuring performance in high contention situations, and is appropriate for measuring the relative performance of locks of varying granularity under high load. We have not attempted to measure the absolute performance of our locks, so detailed specification of the processor, OS version, and VM or run-time version are not particularly relevant. Further we do not present any performance figures comparing our general exclusion lock mechanism with a native mutex. Such extensions to our experiments may be worthy of further consideration but have not yet been conducted.

The eight methods ($i \in 1..8$) of the object are designated as readers or writers. Eight executing threads (ET_i) periodically generate calls on each of these methods. Finally, there is a pool of threads (T_j, $j \in 1..1000$) (see Figure 4). Each pool thread can be grabbed by an executing thread and assigned a method index to be run.

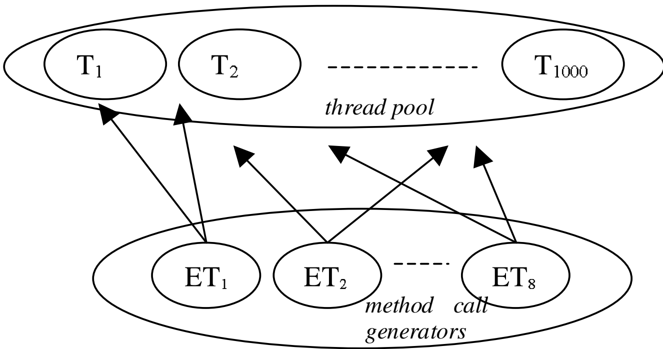


Figure 4. Simulation Setup

The main goal of this setup is to view the behavior of the core object under experimentally controlled rates of method call. The thread pool provides isolation between the blocking behavior of the core object, and the load generation performed by the executing threads; by this means we can accurately control the load generated, and not have any blocking interference (provided of course that the thread pool is large enough). In summary, each recycled pool thread will call only a single method, and will itself serve just one executing thread at a time. The duration of the simulation is fixed at 20 seconds during which methods are called at fixed intervals. The experiment is repeated many times for each interarrival time. The call rate for method invocations is calculated by:

(*Simulation Duration / Interarrival Time*) x *Number of Methods*

So, if the interarrival time is set at 200 ms, the rate at which methods are called is 800 invocations per simulation experiment. All experiments were carried out on a Windows 2000 uniprocessor machine. Each experiment is repeated three times under different exclusion control levels.

Rx8 $\quad r_1 r_2 r_3 r_4 r_5 r_6 r_7 r_8$

RWx4 $\quad r_1 \overline{w_1} \mid r_2 \overline{w_2} \mid r_3 \overline{w_3} \mid r_4 \overline{w_4}$

RWx2 $\quad r_1 r_2 \overline{w_1 w_2} \mid r_3 r_4 \overline{w_3 w_4}$

MUTEX $\quad \overline{w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8}$

The first control level (Rx8) represents a vacuous control where all methods are readers and may proceed concurrently. The control (RWx4) represents a fine-grain exclusion control where all writers are allowed to run concurrently. The third level of control (RWx2) is more restrictive where at most two writers are running at a time. The last control (MUTEX) represents a full mutual exclusion.

The main goal behind this setup is to observe the system behavior under each exclusion control level. In particular, we are interested in observing the throughput of the system, that is, the number of completed methods with respect to the load and under the three main different levels of exclusion control (RWx4,RWx2,MUTEX). Two sets of experiments were conducted; the first set was implemented with Java and the other with C#.

The following are some definitions that are related to our experiments that may mean something different in other contexts:

Throughput: the number of completed methods at the end of experiment duration.

Blocking Time: the difference between when the base method is allowed to execute and when the method is initially invocated.

Running Time: the actual execution time of a method ignoring blocking time.

Figure 5 depicts the relationship between blocking time and running time for a single method invocation; essentially these correspond to lock request, lock acquisition and lock release times.

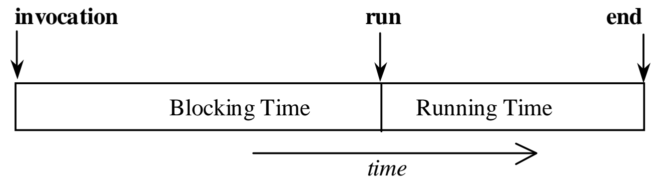


Figure 5. Method Times

6. RESULTS FOR JAVA

The first set of experiments was implemented with Java using JDK 1.3 with Sun's JVM. Figures 6, 7 and 8 depict the throughput, the blocking time, and the running time respectively. We neglected to test the vacuous control (no blocking) case, but we would expect the results to be similar to those for the C# experiments. The x-axis represents the varying load, that is, the rate at which methods are called during the experiment. The y-axis represents the number of completed method calls, representing the throughput. Each mark on the graph represents the mean value of three observations of the experiment execution.

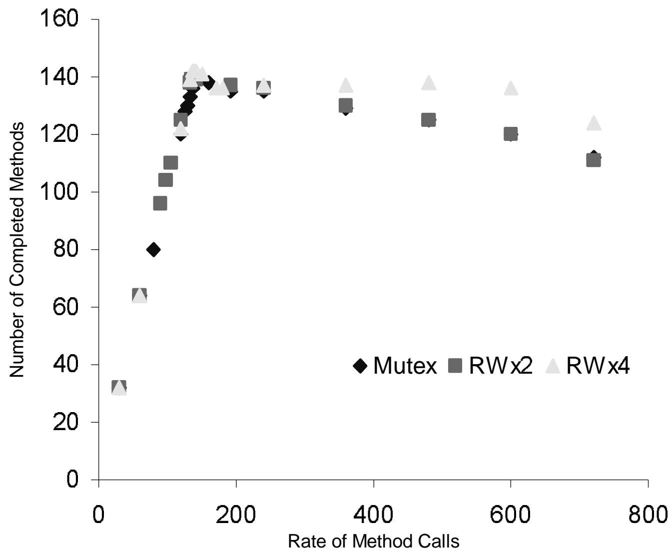


Figure 6. Throughput under Java

Up to the saturation point, all three levels of exclusion control behave similarly in terms of the throughput vs. load relation. As the load increases the throughput levels climb proportional to the load. The blocking time is close to zero in all three, and the running time is also constant with MUTEX < RWx2 < RWx4.

After the saturation point, throughput starts to decrease for all three levels. This is expected since there will be an increasing number of threads lining up waiting to be processed. This will in turn increase the demand for the processor's resources due to the management of more threads and more resources are used up for threads in context switching. Thus fewer resources would be available for method execution and consequently the total throughput would decrease. Our general observation is that there is little difference between MUTEX and RWx2 results. The fine-grain exclusion RWx4 gives just around 5-10% of improved throughput under heavy load. Results for blocking time and running time will be discussed in section 8 when compared to the C# results.

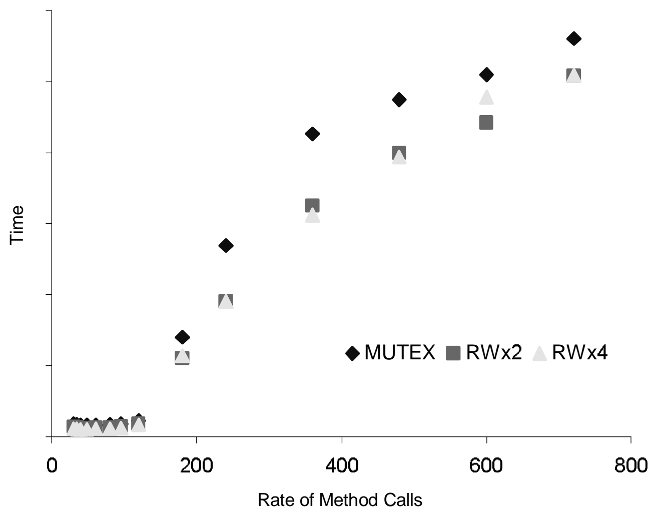


Figure 7. Blocking Time under Java

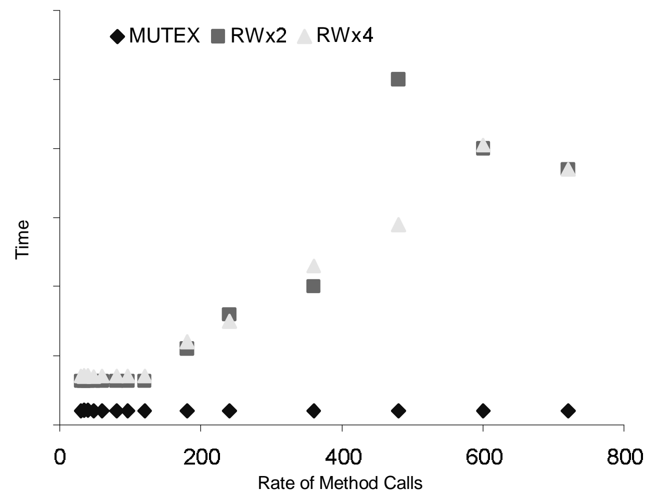


Figure 8. Running Time under Java

7. RESULTS FOR C#

7.1 Preliminary Results for C#

Initial results under the C# (.NET) implementation are shown in figure 9 which represents the throughput under all four levels of control. Each mark on the graph represents the mean value of three observations of the experiment execution. Again, here we can see that there is almost a one-to-one relation between the number of threads serving method calls and the number of completed threads, up to the saturation point for each of the controls where things start to change. After saturation, the MUTEX control starts to decrease as the load of the system increases, whereas the RW controls maintained their pace with a slight increase at higher loads.

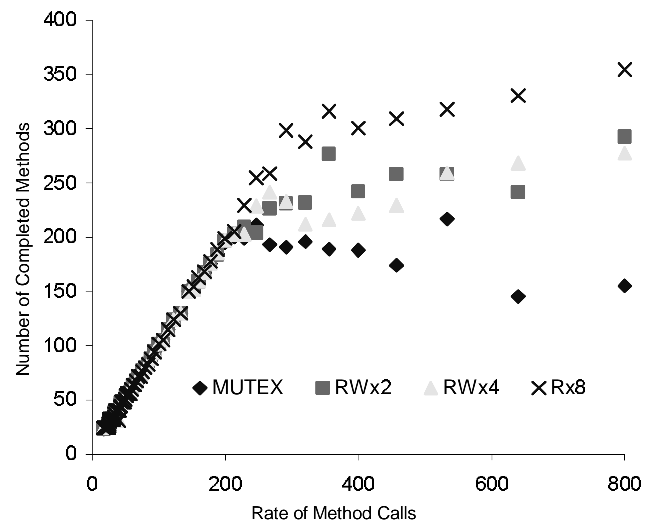


Figure 9. C# Throughput Preliminary Results

However, it is also clear that some marks on the graph show a highly variable behavior and this suggests differences in operating conditions. One possible explanation for this irregularity is that the scheduler may be indulging in priority boosting, which is a trick used by Windows NT to adapt priority levels, which can help to circumvent priority inversion problems, as well as allow the OS to modify its fixed-priority schedule. We also noticed a similar irregularity in the related blocking time graph and the running time graph.

We conducted more extensive experiments to analyze this behavior. Figure 10 depicts the throughput behavior when run 20 times

for a fixed method call rate. Particularly for the RWx2 and RWx4 data one can clearly observe a split into two categories. Hence we have decided to (visually) split all of these measurements into two classes that we call unboosted (class I) and boosted (class II). This admittedly ad hoc approach has resulted in much more regular graphs.

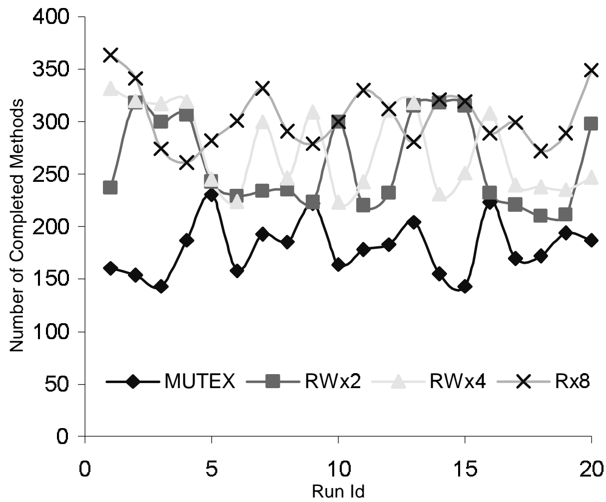


Figure 10. High Variations

7.2 Reclassified C# Results

At each point of the delay time we made 20 measurements for each control. We then classified the resulting data, ending up with a more regular presentation of the observations. Figure 11 represents the throughput vs. load for the unboosted data and Figure 12 represents the same relationship for the boosted data.

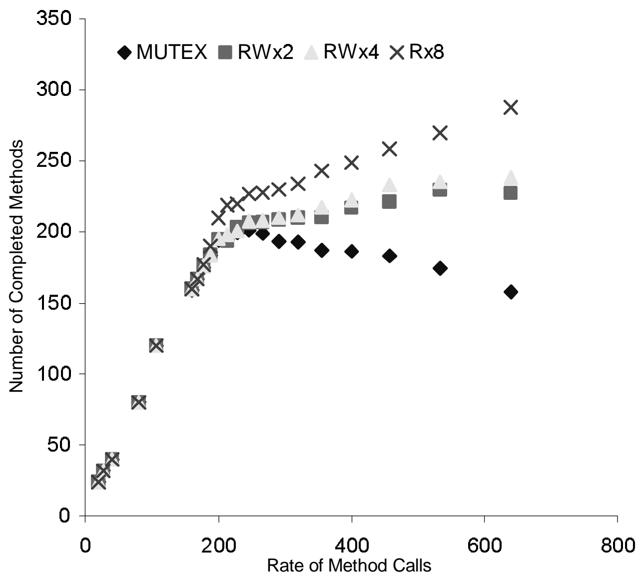


Figure 11. Throughput (classification I)

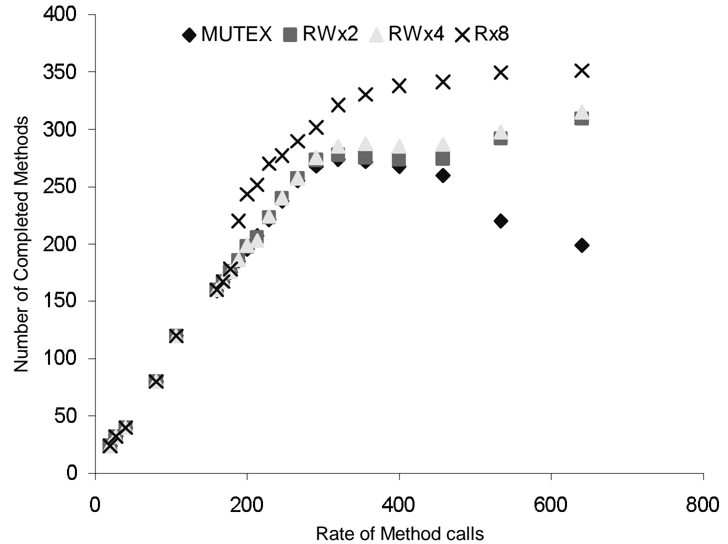


Figure 12. Throughput Boosted (classification II)

The two resulting graphs clearly show a smoother representation of the differences between all three levels of exclusion control. All three controls respond to the system load in a similar way, as all three controls increase proportionally to the load before reaching their saturation points. After the saturation point, a clear separation is observed between the MUTEX control and the RW controls.

Figure 13 presents the blocking time per method under all exclusion controls. Initially, all exclusion controls show low blocking time before saturation point. As the number of method calls increases past the saturation points, MUTEX blocking time is considerably higher than for the other three controls. This result is expected because RW controls allow more than one thread to access the object at a time.

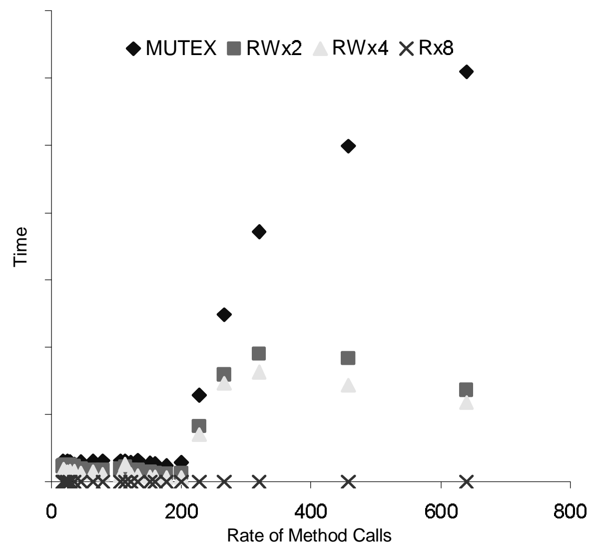


Figure 13. Blocking time under C#

The average running time per method for each level of control is depicted in figure 14. Here, also as expected, the running time under the MUTEX control is minimal and constant; this is because the MUTEX control always guarantee one active thread at a time, thus maximum processor's resources are utilized. The other three RW controls show less utilization of resources for each method, and that's due to the increasing number of threads that are being released, and thus the sharing of processor's resources among the threads.

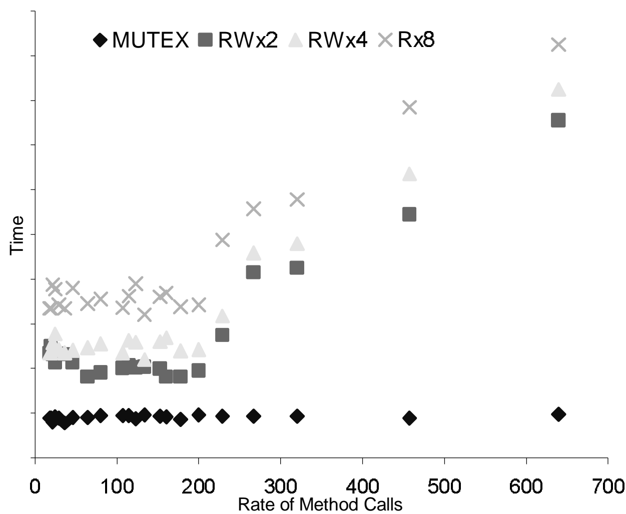


Figure 14. Method running time under C#

8. JAVA VS C#

As mentioned earlier, we have been able to produce two sets of experiments that provide us with results showing some similarities and some differences too. Throughput graphs under both implementations show identical behavior before the saturation points. In both systems, an obvious pattern of one-to-one relation between the number of started threads and the number of completed threads is observed. After the saturation point, both systems behave differently. The Java graph shows the throughput slightly decreasing under all three controls, whereas its C# counterpart shows a decrease only under the MUTEX control while the other three RW controls show a surprising slight increase after the saturation point: we find this increase difficult to explain. Also, C# graph shows a bigger gap between the MUTEX control and the RW controls.

Blocking time per method is the time spent by each thread waiting to enter a method once that thread is launched. Before saturation point, blocking time for both Java and C# (Figures 7 and 13) shows a similar trend where all threads under all controls experience almost no blocking time. Java blocking times start to increase sharply after the saturation point with little difference between the controls. This expected behavior is not totally manifested in the C# blocking time graph. The C# blocking time for the MUTEX control is similar to the Java counterpart. After saturation points, the other three RW controls surprisingly start to drop after gaining some increase.

Running time graphs (Figures 8 and 14) represent the time a thread actually takes to execute within the core object. In this case we expect to see a minimum and constant time for the MUTEX control throughout the simulation run, and that's clearly observed under the two implementations. The RW behavior on the other hand shows a similar and expected pattern in both implementations. As mentioned earlier, RW control allows multiple threads to be active concurrently, consequently, the processor's resources are shared among threads for method executions, thus methods take longer to complete.

Although the two implementations of the same simulation model are almost identical in terms of program logic and flow, results under C# show clear differences between the full MUTEX control and the less restrictive RW controls. At the time the Java throughput graph shows a slight decrease as the system load increases, C# throughput graph shows a slight increase for the RW controls as the load increases. An obvious difference appears when comparing the blocking graphs of both implementations. It's evident that thread management algorithms work differently in the two environments.

The Java implementation was developed first and then the C# one. The similarities between Java and C# allowed us to rewrite Java programs in C# with little effort. However, there remained some design issues for the threading model. Java presents a hierarchical model where its threads are created by subclassing the *java.lang.Thread* class and overriding its run( ) method or by implementing the *java.lang.Runnable* interface and implementing the run( ) method. On the other hand in C#, one creates a thread by creating a new *System.Threading.ThreadStart* object and passing it a delegate which is initialized with the method that is to be run as a thread.

9. DISCUSSION

Our concern with these experiments has been to make relative comparisons, and we have made no attempt to look at absolute performance issues. Our motivation for looking at both Java and C# was to see if the effects of variable locking granularity were consistent across both environments. In general, we can say that there is indeed much in common in our observations. First, under low load conditions, varying granularity makes little difference to overall performance. For isolated applications, the differences appear when we overload the controlled object. In all cases, finer-grained controls resulted in better system throughput than did a mutex strategy. This reassures us in two complementary ways: employing finer-grained control imposes no extra unforeseen runtime overheads; and, it is actually beneficial under heavy-load conditions.

Overall the observations that surprised us most were the high variability of our initial observations in the C# environment. Our further detailed observations are consistent with this variability being caused by some kind of adaptive thread management such as priority boosting. We have compensated for this by partitioning our data into two cases; this provides more regularity in our observations. The other surprise bears further investigation: why does the throughput under heavy load for C#, RW, continue to increase, albeit at a slower rate, after saturation?

9.1 Further Observations

One further observation is quite obvious on afterthought, but is still interesting to note. With our Java experiments we produced a small demonstration displaying the concurrent execution of the various method bodies under the three different exclusion controls. We ran the three displays in parallel, one for each level of locking granularity. Lo and behold, the fine-grained display raced away with its methods executing "concurrently"; after it terminated, the intermediate level control display raced ahead; and finally, when left on it's own, the mutex display sped up. This arises simply because the finer-grained control allows a greater number of active threads, it is able to grab a proportionately bigger chunk of the CPU resource. This suggests that we could choose to use finer-grained locking to increase our share of the CPU resource. In a truly competitive situation though, this argument does wear thin, for what we have is essentially a zero-sum game.

Another observation took us by surprise. The MUTEX measurements reported in this paper have all used the same exclusion lock as the other locking strategies, but with the conflict table configured for mutual exclusion. However, with Java, we also ran the experiment with a conventional synchronization wrapper. Given that our exclusion lock itself uses synchronization, we were surprised that in its MUTEX configuration, it performed marginally better than the conventional synchronization wrapper. We did not investigate this further, and without being privy to the details of synchronization in the JVM, we can only surmise that this is either some inefficiency of the synchronization mechanism, or perhaps that we have cheated by not making our lock reentrant. It would be interesting to check this with the new synchronization primitives in Java 1.5.

9.2 Reentrancy

In implementing our general-purpose exclusion lock, we chose not to make the lock reentrant. Clearly this simplifies the implementation because otherwise we would need to track thread identifiers in the state of the lock for all admitted threads. The exclusion model is implicitly more complex to reason about: any reentrant thread should only be permitted to reenter with the same "permission" it was granted in its original access. This implies that a programmer must be able to track these dependencies, in order to specify the exclusion requirement in the first place. In fact, in contexts where wrappers are applicable, there is an assumption that self-calls by the core object are not controlled, so there is no need to handle reentrant self-calls. If other external call cycles are permitted, we suspect that alternative, more complex locking strategies should be employed. All being said though, there is no particular difficulty in providing a reentrant implementation of the general-purpose exclusion lock.

9.3 Language vs Tools Support

We believe that providing programmers with simple means to achieve better control over the synchronization is achievable based on the implementation experiments we have conducted. The exclusion controls are easy to specify, provided the programmer understands the read-write effects on shared components by its methods.

The simplest way to provide this support would be to write factory methods for exclusion controls, parameterized by the object's type and an exclusion expression. Using reflection it is straightforward to set up the appropriate tables for managing the exclusion lock, and to dynamically generate the wrapper code. This code needs to be created dynamically, or else there needs to be a standard (preferably efficient) way to bind method calls to entries in a table. Without method call reification, dynamic code generation, compilation and loading, seems to be the way to go.

Although the dynamic approach is probably the easiest way to proceed in the short term, we believe that there is scope for language extensions that admit customization of object construction by the compiler. For example, if we could couple an exclusion expression with an object creation statement, then a compiler would be able to provide specific versions of a class with direct implementation of the exclusion control, without using wrappers. This is similar to the way in which class templates are instantiated in C++: each different instantiation generates new code. For us, code bloat could be limited by restricting to two versions of the class: synchronized and unsynchronized. The task of the compiler would then be to instantiate the conflict table for the exclusion control for each different exclusion expression used, as part of the object data.

Finally we would like to see language support for class implementers to express the exclusion contract. This amounts to an ability to write an exclusion expression that specifies the minimum exclusion requirements of the class; such requirements will be implementation dependent. Client code must meet these requirements, either by directly providing an appropriate synchronization control, or by operating within a controlled environment (for example, being single-threaded). Such requirements are quite similar to specifying read-write effects, except that the requirements do not need to specify what is being affected, but rather what the read-write conflicts are between different methods.

10. CONCLUSION

We have conducted experiments to compare the effect of locking granularity for exclusion controls in both Java and C# environments. These experiments have conclusively demonstrated that there need be no performance penalty in implementing a fine-grained locking strategy. For isolated applications under low load there is negligible performance difference with varying locking granularity. However, at hotspots where an object is not coping with its load, finer-grained control yields improved system throughput. The improvement appears to be relatively greater with C#, but we have not investigated possible explanations for this.

For a programmer, such fine-grain controls are trivial to specify, either using a textual form such as that offered by the algebra of exclusion, or by some other conflict matrix associated with an object's interface. It is then relatively straightforward to automatically generate synchronization code or wrappers that implement the specified exclusion control. A side-benefit of encouraging this approach for incorporating thread-safety is that the minimal required exclusion can be documented as part of the object's contract, thereby providing some further information about the read-write effects and underlying sharing between different methods, without any need to expose implementation code and data.

10.1 Acknowledgments

The authors thank the anonymous reviewers for the effort and attention they gave to their comments and criticisms, which have helped us to express the goals of our experiments more clearly.

11. REFERENCES

[1] Antonio J. Nebro, Enrique Alba, Francisco Luna, José M. Troya: *.NET as a Platform for Implementing Concurrent Objects* (Research Note). Euro-Par 2002: 125-130.

[2] Fraser, T., Badger, L., Feldman, M. (1999): *Hardening COTS components with generic software wrappers.* Proc. 1999 IEEE Symposium on Security and Privacy. IEEE Computer Society Press.

[3] Holmes D., Noble J. and Potter J., *Aspects of Synchronization*, Proceedings of the Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific '97), IEEE Computer Society, ISBN 0-8186-8485-2, pp 2-14, 1998

[4] Holmes., D., *Synchronization Rings - Composable Synchronization for Object-Oriented Systems.* PhD thesis, Macquarie University, Australia, 1999.

[5] JSR-166 Concurrency Utilities, *Java Concurrency Process.* www.jcp.org

[6] Kleinrock, L., *Queueing Systems*, Volume II: *Computer Applications* , Wiley Interscience, New York, 1976.

[7] Lea D. *Concurrent Programming in Java: Design Principles and Patterns*, (2nd Edition) Addison-Wesley, 1999

[8] Noble J., Holmes D., and Potter J., *Exclusion for Composite Objects*, Proceedings of ACM Conference on Object-Oriented Programming, Systems, and Languages OOPSLA 2000, Minneapolis, USA, 2000.

[9] Philippsen, M., *A Survey of Concurrent Object-Oriented Languages, Concurrency: Practice and Experience.* 12 (2000) 917- 980.

[10] Rez, Theo Harder, *Concurrency Control Issues in Nested Transactions with Enhanced Lock Modes for KBMSs*, DEXA'95, 6th International Conference and Workshop on Database and Expert Systems Applications, 1995.

[11] Suh-Yin Lee, Ruey-Long Liou: *A Multi-Granularity Locking Model for Concurrency Control in Object-Oriented Database Systems.* IEEE Trans. Knowl. Data Eng. 8(1): 144-156 (1996).

[12] Zhang X., Potter J., *Responsive Bisimulation* IFIP TCS 2002: 601-612.

Dynamic Inference of Polymorphic Lock Types

James Rose
rosejr@cs.umd.edu

Nikhil Swamy
nswamy@cs.umd.edu

Michael Hicks
mwh@cs.umd.edu

Computer Science Department
University of Maryland, College Park

ABSTRACT

We present an approach for automatically proving the absence of race conditions in multi-threaded Java programs, using a combination of dynamic and static analysis. The program in question is instrumented so that when executed it will gather information about locking relationships. This information is then fed to our tool, FindLocks, that generates annotations needed to type check the program using the Race-Free Java [12] type system. Our approach extends existing inference algorithms by being fully context-sensitive. We describe the design and implementation of our approach, and our experience applying the tool to a variety of Java programs. We have found the approach works well, but has trouble scaling to large programs, which require extensive testing for full coverage.

1. INTRODUCTION

Writing correct multi-threaded programs is much more difficult than writing correct sequential programs. As Java's language and library support has brought multi-threaded programming into the mainstream, there has been widespread interest in developing tools for detecting and/or preventing concurrency errors in multi-threaded programs, including race conditions, timing dependencies, and deadlocks. There are two main approaches. *Static* approaches, such as those based on type systems, take a program annotated with locking information and prove that the program is free from certain multi-threaded programming errors. A canonical example is the Race-Free Java type system [12]. Dynamic approaches monitor the execution of the program to discover violations of locking protocols based on observed execution history. A canonical example is Eraser [23].

On the face of it, these two techniques that address the same problems seem very far apart.

- The static approach is appealing because static analysis can conservatively model all paths through a program. When a *sound* static analysis can show a fact, that fact must hold in all executions. Thus static analysis can prove the absence of errors such as race conditions and deadlocks without ever running the program, and without requiring the overhead of run-time code monitoring. The downside is that because static analysis must be conservative, it will incorrectly signal errors in correct programs. Such false alarms can be reduced, but not eliminated, by employing sophisticated techniques—e.g., context-, flow-, or path-sensitivity—but at the cost of scalability and implementation complexity.
- The dynamic approach is appealing because run-time code monitors are relatively easy to implement and are less conservative than static analyses [17], in part because they have complete, precise information about the current execution state. The downside of the dynamic approach is that dynamic systems see only certain executions of the program, and so in general they can only conclude facts based on those cases. This means that either the code monitor must be packaged with the code permanently, or else run the risk of post-deployment failures.

Because static analysis can reach sound conclusions[1] and imposes no runtime overhead, we believe it to be the preferred approach whenever possible. However, as we have just discussed, the limitations of static analysis sometimes make it "too hard." Indeed, many static analyses require users to provide additional program annotations to guide the static analyzer. Experience has shown that programmers are reluctant to provide any but the most minimal annotations. For example, the designers of ESC/Java [15] state that such reluctance "has been a major obstacle to the adoption of ESC/Java. This annotation burden appears particularly pronounced when faced with the daunting task of applying ESC/Java to an existing (unannotated) code base." [14].

1.1 Dynamic Annotation Inference

An *annotation inference* tool can reduce or eliminate the need for annotations. A typical approach is to use a whole-program, constraint-based analysis [2]. For detecting race conditions, such an analysis must consider the aliasing relationships between objects in the program. Unfortunately, the state-of-the-art in scalable points-to analysis can be imprecise when modeling common linked data structures, such as lists and trees. These analyses often model elements of a linked structure as a single abstract location, and thus will fail to distinguish between a lock that protects one list element versus another.

In contrast, a dynamic analysis has ready access to program objects and their aliasing relationships. Therefore, we could use a dynamic analysis to generate *candidate annotations* [14] based on its observations, and these can be checked by the static system. The intuitive idea here is that,

[1] Not all static analyses are sound. Indeed, unsound "pattern detectors" have proven to be quite useful for finding bugs [10, 18].

just like for problems in NP, it may be difficult to generate correct statements about a program, but it is easy to check them. We call this combination of dynamic and static analysis *dynamic annotation inference*.

1.2 Contributions

We are exploring the possible benefits of dynamic annotation inference. In this paper, we describe the progress we have made on a prototype system that employs an Eraser-like dynamic analysis to generate candidate annotations for the Race-Free Java (RFJ) type system [12]. This paper makes the following contributions:

- We present a new algorithm (Section 2) for dynamic annotation inference. Our algorithm improves on prior static [14] and dynamic [1] algorithms in being fully context-sensitive (polymorphic), and thus is able to infer types for more programs.
- We describe our experience applying our tool FindLocks to prove the absence of race conditions in a number of variously-sized Java programs (Section 3). This experience speaks to both the power of the static checking system (RFJ) and the efficacy of dynamic inference. We extend prior studies in both areas [13, 1]. In our experience, dynamic inference imposes reasonably little runtime overhead, and infers most needed annotations.
- After comparing to related work (Section 4) we present a number of lessons learned, and lay out a path for continuing work (Section 5). Two key lessons learned are as follows. First, dynamic analysis can frequently discover properties that typical type-based static analyses cannot check. We must consider new, path-sensitive static analyses that can take advantage of dynamically-discovered information. Second, the larger the program being checked, the more difficult it is to write test cases that cover all its code. In the end, we believe the most effective approach will be to combine static, whole-program analysis with dynamic traces to improve the quality of the inferred types.

2. DYNAMIC LOCK TYPE INFERENCE

To check for race conditions, most static and dynamic tools seek to infer or check the *guarded-by* relation. This relation describes which memory locations (and possibly what sequence of operations) are guarded by which locks. Assuming that this relation is consistent, we can be sure that a particular data object or operation will only be accessed when a lock is held, thus ensuring mutual exclusion. In a dynamic system like Eraser [23], the guarded-by relationship is inferred at run-time. In static type checking systems, types expressing what locks guard which locations must be specified by the programmer as annotations, though well-chosen defaults can make the task easier.

2.1 Race-Free Java

The RFJ type system requires that each field `f` of class `C` be guarded by either an internal or an external lock for `f`. To prevent race conditions, this lock must be held whenever `f` is accessed. An internal lock is any "field expression" in scope within class `C`, i.e., an expression of the form `this.f1.f2...fn` or `D.f1.f2...fn`, where `f1` is a static field of `D`. An external lock is one that is acquired outside the scope of `C` by users of `C`'s methods or fields. In RFJ, an external lock is expressed as a class parameter called a *ghost variable*. The guarded-by annotation can refer to this variable as if it were a local field; however, it cannot be acquired within the class, because it is a type-level variable that exists only at compile-time. All locks must be `final`.

Here is a small example of a class that contains both an internal and external lock:

```
class C<ghost Object o> {
  int count guarded_by this;
  int value guarded_by o;
  synchronized void set(int x) requires o {
    count++;
    value = x;
  }
}
```

`C`'s field `count` is guarded by `this`, an internal lock, while `value` is guarded `o`, an external lock. The method `set` ensures these locks are always held when the fields are accessed. In particular, the fact that `set` is synchronized means that the lock `this` is held when it executes, and thus accessing `count` is legal. In addition, the `requires` clause ensures that `o` is held whenever `set` is called.

A ghost variable is instantiated when the object is created. For example:

```
C<this> x = new C<this>();
synchronized (this) {
  x.set(1);
}
```

This code fragment creates a `C` object, instantiating its external lock with the current object. Thus, the call to `x.set(1)` is legal, because the external lock is held before the call.

RFJ also supports the notion of a class whose objects are never shared between threads (they are *thread local*), and thus no lock need be held to access their fields. This mechanism for thread-local data is a weakness of the RFJ type system, as we elaborate further in Section 3.

2.2 Dynamic Annotation Inference

Our goal is to automatically infer `guarded_by` and `requires` annotations for unannotated Java programs. To do this, the target program is instrumented and executed, and we use an Eraser-like algorithm to infer the guarded-by relationship dynamically. The results are used to generate candidate annotations that can be checked statically.

During execution, for every object o and field `f` we maintain the set *lockset*(o, `f`). The lockset is the set of locks that are consistently held when accessing o.`f`. In Eraser, locks and objects o are merely machine addresses. We must additionally store statically expressible "names" for each lock. These names are of the form (`C`,L), such that within an instance of class `C`, L is a valid field expression for the object. Because some objects have an infinite number of potential names (e.g., `List.next.next.next` ... for a cyclic linked list) we set a constant maximum length for the names.

We also maintain a set containing all locks currently held, and their names. Every field access uses this set to refine the lockset for the involved object/field pair: the set of objects in the lockset is intersected with the set of locks currently held; additionally, the set of names for each object in the lockset is intersected with the set of names that is currently valid for that lock.

Once the execution has completed, we attempt to resolve each field's lockset into either an internal or external lock. Consider the *result set*:

$$R(\texttt{C.f}) = \{(o, ls) \mid ls = lockset(o, \texttt{f})\}$$

If we can find a single field expression L within the scope of `C` such that, for all $(o, ls) \in R(\texttt{C.f})$, L names an object in ls, then we have found an internal lock for `C.f`. In the common case, this means simply looking for a field of `C`, preferably `this`, that guards `f` in every instance of `C`. We also consider static fields of another class if no internal field exists.

If no internal lock exists, we must parameterize `C` and look for names at `C`'s allocation sites. Let $allocsite(o)$ be the allocation site where the object o was allocated. Let $allocator(o)$ be the address of `this` when o was allocated. We can partition the result set based on allocation site I, as follows:

$$R(\texttt{C.f})[I] = \left\{ (a, ls) \;\middle|\; \begin{array}{c} (o, ls) \in R(\texttt{C.f}) \\ I = allocsite(o) \\ a = allocator(o) \end{array} \right\}$$

$R(\texttt{C.f})[I]$ can be used to compute the value of the instantiation parameter for `f` for objects created at allocation site I, just as $R(\texttt{C.f})$ contained the information that was used in the initial attempt to resolve an internal lock for `f`.

In other words, we consider separately the n locations where `C` is allocated, and attempt to provide a lock name for `f` that is valid at that location. Because the set associated with each location is smaller than the original set (when $n > 1$), it is possible that each subset is resolvable (i.e., will have a consistent name for the protecting lock) even when the original set is not. If a subset is not resolvable, we repartition the subset: because the partitions have the same structure as the original set, this procedure can be repeated recursively until we reach a static allocation site. Each repartitioning adds a class parameter to the class to be instantiated.

Because each field's lock is resolved separately, a class will initially have as many parameters as it has externally-locked fields. Because classes rarely use more than a single external lock, these parameters will generally be redundant. Thus, FindLocks will merge parameters if, at every instantiation site, the parameters are equivalent.

Consider the example of external locking shown in Figure 1 adapted from Java-Server, one of our case studies. Here the class `CircList` is used as a buffer for log messages. The class provides no synchronization constructs of its own. Instead, the client `LocalLog` protects accesses to the buffer using itself as the lock.

When attempting to infer the `guarded_by` clause of the `alist` field of `CircList`, FindLocks is unable to discover a candidate among the names within the scope of `CircList`. By examining the set of names available at the allocation site of `CircList` within `LocalLog`, FindLocks is able to add the appropriate instantiation parameter for `clist` and make the type of `CircList` polymorphic in the lock.

The above description omits one important special case: classes which allocate themselves. An example is an externally guarded linked list, where each element creates its successor. If we were to use the above algorithm on such a list, we would end up with as many parameters as there are elements in the list. The initial partitioning of the set would result in a set of size one (containing the head of the list), and a set of size $len - 1$, containing the rest of the elements; $len - 1$ subsequent repartitionings would be required to each the ancestral allocator of the tail of the list, outside of the list class.

```
class LocalLog {
  CircList<this> clist guarded_by this;
  LocalLog(int size) {
    clist = new CircList<this>(size);
  }
  synchronized void add(LogRecord lr){
    clist.add(lr);
  }
}

class CircList<ghost Object _l>{
  ArrayList alist guarded_by _l;
  CircList(int size) requires _l {
    alist = new ArrayList(size);
  }
  boolean add(LogRecord lr) requires _l {
    if (isFull()) {
      int oldestIdx = getOldestIndex();
      alist.set(oldestIdx, lr);
    }
    else alist.add(lr);
    return true;
  }
}
```

Figure 1: Example of Polymorphic Locking

To solve this case, we require that every instantiation site of class `C` within class `C` must supply the allocated class with the same parameters that it received; i.e., class `C<a,b>` will only allocate `C` as `C<a,b>`. In effect, we forbid polymorphic recursion of class parameters.

Now, the resolution on result sets is thus changed so that, instead of partitioning objects based on their allocation sites, they are partitioned based on their most recent *external* ancestor in the tree formed by the allocation relation $\{(a, o) \mid a = allocator(o)\}$. For example, assume there are three classes `C`, `D`, `E` which each create a linked list of ten elements of class `L`. Even though 27 of the elements were allocated within class `L`, a single recursive step will result in all 30 objects being associated with the allocation sites of the head elements within `C`, `D`, and `E`, skipping the sites within `L`.

2.3 Implementation

FindLocks executes in two phases. First, the target program is instrumented using the BCEL [7] bytecode manipulation library and executed. During execution, we track field writes to maintain the mapping between the objects and names; we track lock acquire/release to maintain the set of locks held and their names; and we track field reads/writes in order to determine the locks that are consistently held for every object/field pair. We also record the allocation site of every object, to be used in resolving external locks. We do not need to instrument bytecodes that read and write local variables: their names are not relevant, because they can't be locks; and, because the analysis is dynamic, we always have the identity of objects when they are used. Once execution terminates, we resolve the locksets as described above to produce annotations. These annotations are added to the source code by an external tool, which is then checked by RccJava, a type-checker for RFJ.

It should be noted that most JVMs make strong assumptions with regard to the layout of the certain core classes, particularly those in the the `java.lang` package. Our technique involves instrumenting all application code and, only where permissible, instrumenting library classes. We have found that in certain situations it is not straightforward to instrument packages such as `java.util` since there exist circular dependencies from this package to other packages such as `java.lang` which may not be modified. To avoid this problem, core library classes could be annotated manually; this would only need to be done once.

3. EXPERIMENTAL RESULTS

In this section, we describe our experience using FindLocks on a number of Java programs. We present the runtime overhead of running FindLocks. Next we address the accuracy, expressiveness and completeness of the annotations emitted by FindLocks. Finally, we describe our experience using our tool on a large program.

3.1 Sample Programs

Table 1 lists our benchmark programs, with relevant statistics in the first two columns. *Classes* refers to the number of classes that were instrumented. The numbers in parentheses indicate the number of library classes that were instrumented. *LOC* is the number of non-comment non-blank lines of code.

The programs Elevators1 and Elevators2 were written by students at the University of Maryland as part of CMSC433, a course in object-oriented programming. They simulate the scheduling of elevators in a building. Each program came with its own test cases that ran various simulations. For both Elevators programs we were able to instrument the 152 classes that form the `java.util` library.

The program Proxy-Cache was adapted from a program developed at the Technion, Israel Institute of Technology. It consists of an HTTP proxy that runs on a local server. It also provides content caching. Our test cases consisted of stressing Proxy-Cache with concurrent requests using httperf [19].

Weblech is a small web-crawler that was adapted from a program developed at MIT. To test Weblech we had it perform a depth 1 crawl from the University of Maryland home page.

Java-Server is small HTTP server that was developed at the University of Maryland. The program came with its own test cases that consisted of placing about 50 requests to the server.

3.2 Runtime Overhead

In Table 1 the column *Orig* refers to the maximum memory and elapsed time consumed by the program prior to instrumentation. The columns labeled *Instr* refers to the resources consumed by the instrumented program. The *Annot* columns refer to the resources consumed while annotating the source-code with the results of the inference. In each case, the numbers represent the median value from ten trials. The variation is not appreciable. These measurements were performed on a 2 GHz Pentium 4 with 750MB of RAM.

In each of these cases the overhead incurred by the instrumented code is within acceptable limits. While the resources used by the annotation phase may appear extravagant, it should be noted that this phase can easily be integrated into the RccJava framework. The cost of annotation can thus be amortized against the cost of analysis performed by RccJava. We report the expense of the annotation phase only for completeness.

3.3 FindLocks and RccJava

Table 2 describes the results of running RccJava on the annotated programs. The column *Classes* shows the number of classes that were actually annotated. Classes contain no annotations if either the test cases did not cover the class, or sometimes if the class contains no fields. The column *Auto* refers to the number of annotations that were added automatically by FindLocks. In a few cases we were required to add annotations manually. These are recorded in the column *Manual*. The section of the table labeled *Rcc Warnings* classifies the type of warnings issued by RccJava when run on the annotated programs. *Thl* represents spurious race condition warnings about fields that are in fact thread local, or are read-only. The column *Final* records RccJava warnings about the use of locks that are not final expressions. These are spurious warnings too, since, in each case, the lock expressions, though not final expressions, are actually constants. The column *Race* records warnings about real race conditions.

It is clear from the table that the overwhelming majority of warning issued by RccJava refer to thread local fields. Our analysis is able to easily discover when a field is accessed only by a single thread, or if the field, after initialization, is a read-only field. Furthermore, FindLocks notes in particular the case where an object is constructed by one thread and is then handed off to another thread. In each of these cases FindLocks annotates the source with comments (invisible to RccJava) that assists the user in classifying RccJava warnings as spurious or genuine. These results reflect the weakness of the RFJ type system's handling of thread-local data. A more advanced type system would be able to check these usages [8, 16].

The dynamic analysis also assists the user in ignoring RccJava warnings about non-final expressions used as locks. The read-only annotations added by FindLocks help the user to confirm that lock expressions are constant. This was particularly useful in the case of Weblech. Again, a stronger static analysis would be able to check these cases.

Manual annotations were added in some cases to suppress some warnings. For example, RccJava (optionally) assumes that the `this` lock is held during object construction in order to allow for common initialization patterns; this is sound if the constructor does not allow `this` to escape. However, Elevators1 uses a dummy object as a mutex instead of synchronizing on `this`. Thus, in the constructor of the object, RccJava issues warnings about the mutex not being held. Despite adding the annotation to escape these warnings Elevators1 fails to typecheck under RccJava because it contains a real race condition. This race condition is also detected by FindLocks. In this case, FindLocks adds a comment to the field noting the problem.

Elevators2 uses an external synchronization mechanism to guard instances of `java.util.HashSet`. That is, there is a field `elevators` of type `HashSet` and each access to this field is protected by obtaining a lock external to the scope of the `HashSet`. FindLocks infers that `HashSet` has a type that is polymorphic in the type of the lock. FindLocks correctly annotates the `java.util.HashMap` field of

Program	Classes	LOC	Memory (MB)			Time (sec)		
			Orig	**Instr**	**Annot**	**Orig**	**Instr**	**Annot**
ELEVATORS1	4(+152)	567	8.8	48.1	110	8.9	10.0	23.0
ELEVATORS2	4(+152)	408	8.7	46.6	112	8.4	10.2	22.6
PROXY-CACHE	7	1218	12.0	49.7	112	9.8	21.4	14.9
WEBLECH	12	1306	33.1	48.7	127	17.5	18.8	20.3
JAVA-SERVER	36	1768	10.3	37.4	126	7.0	7.9	15.2

Table 1: Runtime Overhead for FINDLOCKS

Program	Annotations			Rcc Warnings			
	Classes	**Auto**	**Manual**	**Thl**	**Final**	**Race**	**Oth**
ELEVATORS1	3	26	1	5	0	1	0
ELEVATORS2	6	27	0	1	0	0	0
PROXY-CACHE	7	69	0	15	0	0	4
WEBLECH	11	52	4	30	10	0	1
JAVA-SERVER	18	59	2	5	0	0	0

Table 2: Checking Annotated Programs

the `HashSet` field as being guarded by the lock parameter. Furthermore, an inner class of `HashMap`, `HashMap.Entry` is also parameterized by the same lock parameter. (This is why ELEVATORS2 annotates six classes: three are from the program itself, and two are from the standard library.)

JAVA-SERVER also uses a similar external synchronization construct as described in Section 2.2. Two manual annotations were required to handle a class that was not executed by our test cases.

We were able get PROXY-CACHE to type check without any further annotations. RCCJAVA does, however, issue warnings regarding two fields of array type. The contents of the arrays are read-only and do not require any synchronization.

Attempting to type-check WEBLECH reveals another limitation of RCCJAVA. The code in Figure 2 is illegal in RCCJAVA: even though every access to the field `Spider.q` is guarded by `Spider.q`, it is not possible to instantiate the lock parameter of the `DownloadQueue` object with `Spider.q`. Using a separate mutex allows the lock parameter to be instantiated correctly.

WEBLECH also reveals a problem associated with subtyping of methods in the presence of `requires` annotations. The `DownloadQueue` object overrides the `Object.toString()` method in which it accesses all its fields. But annotating the overridden method with a requires clause that contains the lock parameter is illegal since required lock sets on function types are contravariant with regard to subtyping. Thus, an assertion that the lock was held was added to handle this case.

3.4 Scaling to Large Programs

We also ran FINDLOCKS on HSQL[2], an open-source, JDBC-compliant database. HSQL consists of 260 classes and about 55000 lines of code. Unfortunately, we did not have access to a comprehensive test-suite for the application. Instead, we devised a simple test program that spawned a large number of threads and repeatedly performed simple queries on the database.

We found both the runtime overhead as well as the annotation overhead to be acceptable. However, the accuracy of the annotations that were inserted were greatly undermined

```
class Spider {
  DownloadQueue<q> q guarded_by q;
  Spider() {
    q = new DownloadQueue<q>();
  }
  URL nextURL(){
    synchronized(q) {
      return q.nextURL();
    }
  }
}
class DownloadQueue<ghost Object _l>{
  ArrayList urls guarded_by _l;
  DownloadQueue() requires _l {
    alist = new ArrayList();
  }
  URL nextURL() requires _l {
    return urls.remove(0);
  }
  String toString() requires _l {
    return urls.toString();
  }
}
```

Figure 2: Illegal Code snippet from WEBLECH

[2] `http://hsqldb.sourceforge.net/`

by the extreme simplicity of the test case. Our test case only managed to cover some 90 out of the 260 classes. The annotations generated are thus skewed with respect to the particular execution trace that the instrumented program generated.

A large number of the 208 warnings issued by RccJava were with regard to fields that were marked thread local by FindLocks. While the inability to handle thread-local fields is an obvious limitation of RccJava, the accuracy of the annotations generated by FindLocks is also questionable. As with any dynamic analysis, FindLocks is limited to drawing unsound conclusions about the program based only on the executions that the analysis has witnessed. A stronger type system is needed in this case.

In light of this experience, it becomes clear that our approach is most likely to succeed when it complements a thorough testing regime. Achieving a good degree of code coverage is essential to inferring correct lock relationships from program traces.

4. RELATED WORK

Our approach is an example of what we call a *dynamic-static analysis*, in which dynamically-gathered information is used to support or improve a static analysis. Ernst's Daikon tool [11] infers simple invariants between variables in a program through run-time profiling. Nimmer and Ernst [20, 21] showed that many of the inferred invariants could be proven sound using a theorem prover. In their approach, dynamically-determined invariants are part of a *candidate set*, and the theorem prover removes those invariants that it cannot prove. Specification mining [4] is a technique for automatically discovering sequencing and data constraints on API calls. The information may be useful for a static verification tool. The most common example of dynamic-static analysis is profile-directed compilation [5, 6, 22, 25]. In this case, generated code is improved by considering run-time profiles. This is a matter of performance, not correctness, so poor profiling information will not cause the program to produce the wrong answers.

A wide variety of type-checking systems have been developed for preventing possible race conditions. However, we know of only two approaches that infer types for such systems, to relieve the annotation burden on the programmer. Houdini [14] is a self-described *annotation assistant* that statically generates sets of candidate annotations based on domain knowledge. Houdini was applied to a simplified version of RFJ [13]. Unfortunately, Houdini does not support external (polymorphic) locks, restricting the set of programs for which it can infer types.

Concurrently with us, Agarwal and Stoller [1] developed a dynamic inference algorithm for the Parameterized Race-Free Java (PRFJ) [8] type system. Their algorithm is similar to ours in many respects. One difference is that it is not *fully* context-sensitive in that it handles polymorphic instantiation, but not polymorphic generalization. In particular, it assumes that either a class has a single lock parameter, or if the class has multiple parameters then the user has annotated it as such. With the knowledge of these lock parameters, their algorithm can infer how to most appropriately instantiate them. Our algorithm not only instantiates lock parameters, but can infer them as well by maintaining an allocation map between an object and the object that allocated it. This allows us to generalize (or "resolve" using our terminology) to arbitrary depth in the allocation chain, creating as many parameters as needed. This is particularly useful for library-like functions, like the `CircList` class we used as an example, which may wish to admit a variety of locking patterns. Agarwal and Stoller's algorithm handles some advanced features of PRFJ not present in RFJ, such as *unique* and *read-only* objects. It would be interesting to carefully consider how the two approaches could be combined, as we discuss below.

5. CONCLUSIONS

Our experience thus far leads us to believe that dynamic analysis can usefully perform annotation inference. Since programmers typically write tests for their programs, dynamic annotation inference imposes only a small burden, and adds value by proving sound properties, in our case the absence of race conditions, based on collected traces. Indeed, our tool inferred the majority of annotations needed for idioms RFJ could check. Moreover, a number of applications made use of external locking, and our approach correctly parameterized classes to express this fact, an improvement over past work [1, 14].

However, our experience has exposed two limiting factors in the technique:

1. In general, a given static analysis may not be able to verify properties easily detected by dynamic analysis. For example, RFJ does not support treating classes as thread-local on a per-field basis. It also cannot check temporal shifts in protection, such as an object that is thread-local at first, but later becomes read-only or shared and locked. Our dynamic tool discovered these situations easily, but the static analysis could not check them.

   The solution is to develop a stronger complementary static checking system. Indeed, PRFJ fixes the first problem, and, to a limited degree, the second, by allowing uniquely referenced objects to be unguarded. This is sufficient to allow hand-offs between threads, which are supported in RFJ only by escapes from the type system. (Other idioms, such as barrier synchronization, remain uncheckable.) The type inference algorithm for PRFJ developed by Agarwal and Stoller [1] is able to indicate uniquely referenced objects (using a complementary static algorithm described in [3]). An immediate possibility for future work is to develop an inference algorithm for PRFJ that combines this ability with our algorithm's ability to infer multiple owner parameters, which are analogous to external locks in RFJ.

   A more ambitious approach would be to develop a more sophisticated type system which requires more annotations, since we have a tool to assist with annotation inference. For example, dynamic analysis can easily and efficiently capture the program execution paths for which a safety property holds. To make best use of this information, our static checking system should be path sensitive. Type systems with intersection-, union-, and dependent types can describe path-sensitive properties. Since (static) type inference in such a system is generally undecidable, dynamic path information will supply needed annotations.

2. A large program may only execute a portion of its code during common usage, and thus a dynamic tool may not generate annotations for the entire program. This was a problem for HSQL.

   We believe that the right approach to this problem, beyond having more comprehensive tests, is to have the dynamic analysis "add value" to a more traditional static inference system. This is similar to the idea of profile-directed compilation [5, 22]. In this case, generated code is improved by considering run-time profiles. In our case, candidate annotations could be generated both statically and dynamically, and checked for soundness in the style of Houdini [13, 14]. One challenge would be the effective handling of class parameterization.

An interesting avenue of future work is to evaluate, under a variety of metrics, when the technique of applying dynamic analysis to aid sound static analysis makes sense. In general, the fact that a property is satisfied by some set of executions does not imply that the property holds for the entire program. However, in our work the guarded-by relation discovered by the dynamic instrumentation can frequently be proved sound for the whole program. The interesting question is when and why this is the case. While work has been done to characterize the computability classes of runtime analysis as compared to static analysis [17, 24], little has been done to explore the two at the level of actual programs. For example, Ernst [11] has found that dynamically-inferred properties sometimes hold statically, but does little to explain why. We intend to consider program traces and programs that induce them, following abstract interpretation [9].

6. REFERENCES

[1] Rahul Agarwal and Scott D. Stoller. Type Inference for Parameterized Race-Free Java. In *Proceedings of the Fifth International Conference on Verification, Model Checking and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, Venice, Italy, January 2004. Springer-Verlag.

[2] Alexander Aiken, Manuel Fähndrich, Jeffrey S. Foster, and Zhendong Su. A Toolkit for Constructing Type- and Constraint-Based Program Analyses. In Xavier Leroy and Atsushi Ohori, editors, *Proceedings of the Second International Workshop on Types in Compilation*, volume 1473 of *Lecture Notes in Computer Science*, pages 78–96, Kyoto, Japan, March 1998. Springer-Verlag.

[3] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias Annotations for Program Understanding. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 311–330, October 2002.

[4] Glenn Ammons, Rastislav Bodik, and James R. Larus. Mining specifications. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, Portland, Oregon, January 2002.

[5] Glenn Ammons and James R. Larus. Improving data-flow analysis with path profiles. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 72–84, Montreal, Canada, June 1998.

[6] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. In *Proceedings of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 59–70, Albuquerque, New Mexico, January 1992.

[7] Bytecode engineering library. `http://jakarta.apache.org/bcel/`.

[8] Chandrasekhar Boyapati and Martin Rinard. A Parameterized Type System for Race-Free Java Programs. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 56–69, November 2001.

[9] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[10] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 237–252, Bolton Landing, New York, October 2003.

[11] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001.

[12] Cormac Flanagan and Stephen N. Freund. Type-Based Race Detection for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 219–232, Vancouver B.C., Canada, June 2000.

[13] Cormac Flanagan and Stephen N. Freund. Detecting race conditions in large programs. In *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 90–96, Snowbird, Utah, June 2001.

[14] Cormac Flanagan and K. Rustan M. Leino. Houdini, an Annotation Assitant for ESC/Java. In J. N. Oliverira and Pamela Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods*, number 2021 in Lecture Notes in Computer Science, pages 500–517, Berlin, Germany, March 2001. Springer-Verlag.

[15] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245, Berlin, Germany, June 2002.

[16] Dan Grossman. Type-Safe Multithreading in Cyclone. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, pages 13–25, New Orleans,

Louisiana, USA, January 2003.

[17] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. Technical Report 2003-1908, Cornell Unviversity Department of Computer Science, 2003.

[18] David Hovemeyer and William Pugh. Finding Bugs Is Easy. `http://www.cs.umd.edu/~pugh/java/bugs/docs/findbugsPaper.pdf`, 2003.

[19] David Mosberger and Tai Jin. httperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance*, pages 59—67. ACM, June 1998.

[20] Jeremy W. Nimmer and Michael D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *Proceedings of the First Workshop on Runtime Verification (RV '01)*, July 2001.

[21] Jeremy W. Nimmer and Michael D. Ernst. Invariant Inference for Static Checking: An Empirical Evaluation. In *Tenth Symposium on the Foundations of Software Engineering*, pages 11–20, Charleston, South Carolina, USA, November 2002.

[22] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the 1990 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 16–27, White Plains, New York, June 1990.

[23] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 27–37, St. Malo, France, October 1997.

[24] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and Systems Security*, 3(1):30–50, February 2000.

[25] Youfeng Wu and James R. Larus. Static branch frequency and program profile analysis. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 1–11, San Jose, CA, 1994.

Rigorous Concurrency Analysis of Multithreaded Programs

Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom*
School of Computing, University of Utah
{yyang | ganesh | gary}@cs.utah.edu

ABSTRACT

This paper explores the practicality of conducting program analysis for multithreaded software using constraint solving. By precisely defining the underlying memory consistency rules in addition to the intra-thread program semantics, our approach offers a unique advantage for program verification — it provides *accurate* and *exhaustive* coverage of all thread interleavings for any given memory model. We demonstrate how this can be achieved by formalizing sequential consistency for a source language that supports control branches and a monitor-style mutual exclusion mechanism. We then discuss how to formulate programmer expectations as constraints and propose three concrete applications of this approach: execution validation, race detection, and atomicity analysis. Finally, we describe the implementation of a formal analysis tool using constraint logic programming, with promising initial results for reasoning about small but non-trivial concurrent programs.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages; D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms

Verification, Testing, Reliability

Keywords

Multithreaded programming, constraint solving, data races, race conditions, atomicity, memory consistency models, static checking

*This work was supported in part by Research Grant No. CCR-0081406 (ITR Program) of NSF and SRC Task 1031.001.

1. INTRODUCTION

Unlike a sequential program, which simply requires that each read observes the latest write on the same variable according to program order, a multithreaded program has to rely on the *thread semantics* (also known as the *memory model*) to define its legal outcome in a shared memory environment. The most commonly known memory model is *sequential consistency* (SC) [1]. As a natural extension of the sequential model, sequential consistency requires that (i) operations of all threads can exhibit a total order, (ii) operations of each individual thread appear in this total order following program order, and (iii) a read observes the latest write on the same variable according to this total order. Many weaker shared memory systems (see [2] for a survey) have also been developed to enhance performance.

Java is the first widely deployed programming language that provides built-in threading support at the language level. Unfortunately, developing a rigorous and intuitive Java Memory Model (JMM) has turned out to be very difficult. The existing JMM is flawed [3] due to the lack of rigor. It is currently under an official revision process and a new JMM draft [4] is proposed for community review.

Although multithreading provides a powerful programming paradigm for developing well structured and high performance software, it is also notoriously hard to get right. Programmers are torn on the horns of a dilemma regarding the use of synchronization: too much may impact performance and risk deadlock, too little may lead to race conditions and application inconsistency. Therefore, a formal analysis about thread behaviors is often needed to make a program more reliable. However, this can become a daunting task with a traditional pencil-and-paper approach.

For example, one common analysis is *race detection*. Consider program 1 in Figure 1 (taken from [4]), where each thread issues a read and a conditional write. Does this program contain data races? At the first glance, it may appear that the answer is "yes" since it seems to fit the conventional intuition about a race condition — two operations from dif-

Thread 1	Thread 2
$r1 = x;$	$r2 = y;$
if$(r1 > 0)$	if$(r2 > 0)$
$y = 1;$	$x = 1;$

(a) Program 1

Thread 1	Thread 2
$r1 = x;$	$r2 = y;$
if$(r1 > 0)$	if$(r2 >= 0)$
$y = 1;$	$x = 1;$

(b) Program 2

Figure 1: Initially, x = y = 0. Are these programs race-free?

Thread 1 (Deposit)	Thread 2 (Withdraw)
Lock $l1$;	Lock $l1$;
$r1$ = balance;	$r3$ = balance;
Unlock $l1$;	Unlock $l1$;
$r2 = r1 + 1$;	$r4 = r3 - 1$;
Lock $l1$;	Lock $l1$;
balance = $r2$;	balance = $r4$;
Unlock $l1$;	Unlock $l1$;

Figure 2: The transactions are not atomic even though the program is race-free.

ferent threads (e.g., $r1 = x$ in thread 1 and $x = 1$ in thread 2) attempt to access the same variable without explicit synchronization, and at least one of them is a write. However, a more careful analysis reveals that the control flow of the program, which must be consistent with the read values allowed by the memory model, needs to be considered to determine whether certain operations will ever happen. Therefore, before answering the question, one must clarify what memory model is assumed. With sequentially consistent executions, for instance, the branch conditions in program 1 will never be satisfied. Consequently, the writes can never be executed and the code is race-free. Now consider program 2 in Figure 1, where the only difference is that the branch condition in Thread 2 is changed to $r2 >= 0$. Albeit subtle, this change would result in data races.

Another useful analysis is *execution validation*, which is for verifying whether a certain outcome is permitted. This can help a programmer understand the memory ordering rules and aid them in code selection. Similar to a race analysis, data/control flow must be tracked for execution validation. For multithreaded programs, data/control flow is interwoven with shared memory consistency requirements. This makes it extremely hard and error-prone to hand-prove thread behaviors, even for small programs.

The *atomicity* requirement is also frequently needed to ensure that certain operations appear to be executed atomically. As pointed out in previous publications (e.g., [5]), the absence of race conditions does not guarantee the absence of atomicity violations. For example, consider the program in Figure 2. Thread 1 and 2 respectively implement the deposit and withdraw transactions for a bank account. This program is race-free because all accesses to the global variable *balance* are protected by the same lock. If these two threads are issued concurrently when *balance* is initially 1, *balance* should remain the same after the program completes if implemented properly. But due to the use of local variables, one thread can interleave with another while in a "transient" state. Consequently, the final balance can be 0, 1, or 2 depending on the scheduling, which is clearly not what has been intended. Hence, it would be highly desirable to have a systematic approach to specifying and verifying such programmer expectations.

From these examples, several conclusions can be drawn.

- The precise thread semantics, in addition to the intra-thread program semantics, must be taken into account to enable a rigorous analysis of multithreaded software, because a property that is satisfied under one memory model can be easily broken under another.
- Program properties such as race conditions and atomicity requirements need to be formalized because informal intuitions often lead to inaccurate results.
- An automatic verification tool with exhaustive coverage is extremely valuable for general software development purposes because thread behaviors are often confusing.

Based on these observations, we develop a formal framework for reasoning about multithreaded software. Our key insight is that by capturing thread semantics and correctness properties as constraints, we can reduce a verification problem to a constraint satisfaction problem or an equivalent boolean satisfiability problem, thus allowing us to employ an efficient constraint/SAT solver to automate the analysis. Using a verification tool harnessed with these techniques, we can configure the underlying memory model, select a program property of interest, take a test program as input, and verify the result automatically under all executions. Existing program analyses rely on simplifying assumptions about the underlying execution platform and thereby introduce unsoundness. They tend to only concentrate on efficiency or scalability. While these are highly worthy goals, there is also a clear need for supporting exhaustive analysis. Our approach fills in this gap by providing a mechanism that makes the thread semantics explicit.

This paper offers the following contributions. (i) We develop a formal executable specification of sequential consistency for a non-trivial source language that supports the use of local variables, computations, control branches[1], and a monitor-like mechanism for mutual exclusion. One key result of this paper is to show that it is feasible and beneficial to precisely capture both program semantics (including local data dependence and local control dependence) and memory model semantics in the same setting. As far as we know, no one has previously provided such a formal executable specification. (ii) We propose a method to formulate program properties as constraints and automatically verify them using constraint solving. In particular, we formalize the conditions of three critical safety properties: execution legality, race conditions, and atomicity requirements. (iii) We build a tool using constraint logic programming (CLP) and report the experiences gained during the implementation.

The rest of the paper proceeds as follows. In Section 2, we provide an overview of our approach. Section 3 describes the source language used as the basis of our presentation. In Section 4, we apply our approach for execution validation. Race conditions and atomicity requirements are formalized in Section 5 and 6, respectively. We discuss the implementation of our prototype tool in Section 7. Related work is reviewed in Section 8. We conclude and explore future work in Section 9. The detailed formal specification is presented in the Appendix.

2. OVERVIEW

Our approach is based on a memory model specification framework called Nemos (Non-operational yet Executable

[1]Currently our formal executable specification does not directly support loops.

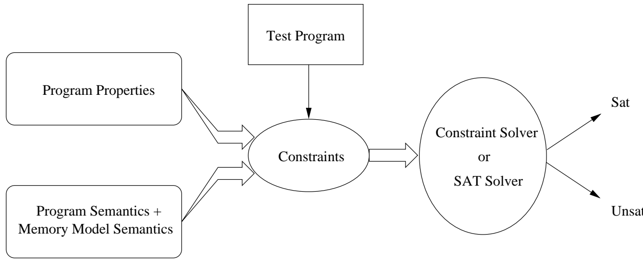


Figure 3: The processing flow of our approach.

Memory Ordering Specifications) [6, 7]. Nemos defines a memory model in a declarative style using a collection of ordering rules. The processing flow of our verification methodology is shown in Figure 3, which comprises the following steps: (i) capturing the semantics of the source language, including the thread semantics, as constraints, (ii) formalizing program properties as additional constraints, (iii) applying these constraints to a given test program and reducing the verification problem to a constraint satisfaction problem, and (iv) employing a suitable tool to solve the constraint problem automatically.

2.1 Specifying the Constraints

We use predicate logic to specify the ordering constraints imposed on a relation *order*. To make our specifications compositional and executable, our notation differs from previous formalisms in two ways. First, we employ a modest extension of predicate logic to higher order logic, i.e. *order* can be used as a parameter in a constraint definition so that new refinements to the ordering requirement can be conveniently added. This allows us to construct a complex model using simpler components. Second, our specifications are fully explicit about all ordering properties, including previously implicit requirements such as totality, transitivity, and cycle-freedom. Without explicating such "hidden" requirements, a specification is not complete for execution.

2.2 Solving the Constraints

After the language semantics and program properties are specified as constraints, they can be applied to a finite execution trace. This process converts the system requirements from higher order logic to propositional logic.

The Algorithm: Given a test program $\mathcal{P}$, we first derive its execution *ops* (defined in Section 3.1) from the program text in a preprocessing phase. The initial execution is fully *symbolic*, that is, *ops* may contain free variables, e.g., for data values and ordering relations. Suppose *ops* has n operations, there are n^2 ordering pairs among these operations. We construct a $n \times n$ adjacency matrix $\mathcal{M}$, where the element $\mathcal{M}_{ij}$ indicates whether operations i and j should be ordered. We then go through each requirement in the specification and impose the corresponding propositional constraints with respect to the elements of $\mathcal{M}$. The goal is to find a binding of the free variables in *ops* such that it satisfies the conjunction of all requirements or to conclude that no such binding exists. This is automated using a constraint solver.

3. THE SOURCE LANGUAGE

This section develops the formal semantics of sequential consistency for a source language that supports many common programming language constructs. The choice of using sequential consistency as the basis of our formal development is motivated by two factors. (i) SC is often the implicitly assumed model during software development, i.e., many algorithms and compilation techniques are developed under the assumption of SC. (ii) Many weak memory models, including the new JMM draft, define pivotal properties such as race-freedom using SC executions. Providing such an executable definition of race conditions will provide a solid foundation upon which a full-featured Java race-detector can be built.

Although this paper formalizes only SC, our framework is generic and allows an arbitrary memory model to be plugged-in for a formal comparative analysis. In our previous work, we have already applied Nemos to build a large collection of memory model specifications, including the Intel Itanium Memory Model [7] and a variety of classical memory models [6], such as sequential consistency, coherence, PRAM, causal consistency, and processor consistency.

Our previous specification of sequential consistency in [6] only deals with normal read and write operations. While this is sufficient for most processor level memory systems, it is not enough for describing language level thread activities. In order to handle more realistic programs, this paper extends the previous model by supporting a language that allows the use of local variables, computation operations, control branches, and synchronization operations.

3.1 Terminology

Variables: Variables are categorized as *global variables*, *local variables*, *control variables*, and *synchronization variables*. Global and synchronization variables are visible to all threads. Local and control variables are thread local. Control variables do not exist in the original source program — they are introduced by our system as auxiliary variables for control operations. Synchronization variables correspond to the locks employed for mutual exclusion. In our examples, we follow a convention that uses x, y for global variables, $r1$, $r2$ for local variables, $c1$, $c2$ for control variables, $l1$, $l2$ for synchronization variables, and 0, 1 for primitive data values.

Instruction: An *instruction* corresponds to a program statement from the program text. The source language has a syntax similar to Java, with $Lock$ and $Unlock$ inserted corresponding to the Java keyword *synchronized*. It supports the following instruction types:

Read:	e.g., $r1 = x$
Write:	e.g., $x = 1$, or $x = r1$
Computation:	e.g., $r1 = r2 + 1$
Control:	e.g., $if(r1 > 0)$
Lock:	e.g., $Lock\ l1$
Unlock:	e.g., $Unlock\ l1$

Execution: An *execution* consists of a set of symbolic operation instances generated by program instructions. We assume that the expression involved in a computation or control operation only uses local variables. If the original instruction performs a computation on global variables, it will be divided into read operations followed by computation operations.

Operation Tuple: An operation i is represented by a tuple: $i = \langle t, pc, op, var, data, local, localData, cmpExpr, ctrExpr, lock, matchID, id\rangle$, which we decompose using a number of selector functions (shown in boldface):

$\mathbf{t}\ i = t$:	thread ID
$\mathbf{pc}\ i = pc$:	program counter
$\mathbf{op}\ i = op$:	operation type
$\mathbf{var}\ i = var$:	global variable
$\mathbf{data}\ i = data$:	data value
$\mathbf{local}\ i = local$:	local variable
$\mathbf{localData}\ i = localData$:	data value for local variable
$\mathbf{cmpExpr}\ i = cmpExpr$:	computation expression
$\mathbf{ctrExpr}\ i = ctrExpr$:	path predicate
$\mathbf{lock}\ i = lock$:	lock
$\mathbf{matchID}\ i = matchID$:	ID of the matching lock
$\mathbf{id}\ i = id$:	global ID of the operation

For every global variable x, there is a default write operation for x, with the default value of x and a special thread ID t_{init}. We assume Lock and Unlock operations are properly nested. Each trailing Unlock stores the id of the matching Lock in its $matchID$ field.

3.2 Semantics

3.2.1 Control Flow

It is a major challenge to specify control flow in the context of nondeterministic thread interleavings. We solve this problem by (i) transforming control related operations to auxiliary reads and writes using control variables and (ii) imposing a set of consistency requirements on the "reads" and "writes" of control variables similar to that of normal reads and writes. The detailed steps are as follows:

- For each branch instruction i, say $if(p)$, add a unique auxiliary control variable c, and transform instruction i to an operation i' with the format of $c = p$. Operation i' is said to be a control operation ($\mathbf{op}\ i' = Control$), and can be regarded as an *assignment* to the control variable c.

- Every operation i has a *ctrExpr* field that stores its *path predicate*, which is a boolean expression on a set of control variables dictating the condition for i to be executed. An operation i can be regarded as a *usage* of the involved control variables in the path predicate. Without loops, the path predicate for every operation can be determined during the preprocessing phase. This can be achieved based on a thread local analysis since control variables are thread local.

- An operation i is *feasible* if its *ctrExpr* field evaluates to $True$. We define a predicate **fb** to check the feasibility of an operation.

- In the memory ordering rules, feasibility of the involved operations is checked to make sure the consistency of control flow is satisfied.

By converting control blocks to assignments and usages of control variables, we can specify consistency rules for control flow in a fashion similar to data flow.

3.2.2 Loops

Loops are not directly supported in this specification. For the purpose of defining a memory model alone, nonetheless, our mechanism for handling control operations is sufficient for loops. This is because the task of a memory model specification can be regarded as answering the question of whether a given execution is allowed by the memory model. For any concrete terminated execution, loops have already been resolved to a finite number of iterations.

However, to enable a fully automatic and exhaustive program analysis involving loops, another level of constraints need to be developed so that the path predicate of an operation can conditionally grow. Another technique, as used by tools such as Extended Static Checker for Java (ESC/Java) [8], is to rely on the user to supply loop invariants — loops without invariants are handled in a manner that is unsound but still useful. This approach can be adopted by our system as well. As a future work, we plan to investigate effective approaches for handling loops.

3.2.3 Formal Specification

The semantics of the source language is defined as a collection of constraints. The detailed specification is presented in Appendix A. This section explains each of the rules.

As shown below, predicate `legalSC` is the overall constraint that defines the requirement of sequential consistency on an execution *ops* in which the operations follow an ordering relation *order*.

legalSC *ops order* $\equiv$
  requireProgramOrder *ops order* $\wedge$
  requireReadValue *ops order* $\wedge$
  requireComputation *ops order* $\wedge$
  requireMutualExclusion *ops order* $\wedge$
  requireWeakTotalOrder *ops order* $\wedge$
  requireTransitiveOrder *ops order* $\wedge$
  requireAsymmetricOrder *ops order*

Program Order Rule (Appendix A.1):
Constraint `requireProgramOrder` specifies that operations should respect program order, which is formalized by predicate `orderedByProgram`. In addition, the default writes are ordered before other operations.

Read Value Rules (Appendix A.2):
Constraint `requireReadValue` enforces the consistency of data flow across reads and writes. Informally, it requires that for each read k: (i) there must exist a suitable write i providing the data and (ii) there does not exist an overwriting write j between i and k. The assignments and usages of local variables (local data dependence) and control variables (local control dependence) follow the similar guideline to ensure consistent data transfer. Therefore, `requireReadValue` is decomposed into three subrules for global reads, local reads, and control reads, respectively. Because we apply *unique* control variables, `controlReadValue` does not need to check the second case listed above.

Computation Rule (Appendix A.3):
Constraint `requireComputation` enforces the program semantics. It is not directly related to the memory ordering, but is needed for analyzing realistic code. It requires that for every operation involving computations (i.e., when the op-

eration type is *Computation* or *Control*), the resultant data must be obtained by properly evaluating the expression in the operation. For brevity, the Appendix omits some details of the standard program semantics that are usually well understood. For example, we use a predicate `eval` to indicate that the standard process should be followed to evaluation an expression. Similarly, `getLocals` and `getCtrs` are used to parse the *cmpExpr* and *ctrExpr* fields to obtain a set of ⟨variable, data⟩ entries involved in the expressions (these entries represent the local/control variables that the operation depends on and their associated data values), which can be subsequently processed by `getVar` and `getData`.

Mutual Exclusion Rule (Appendix A.4):
Constraint `requireMutualExclusion` enforces mutual exclusion for operations enclosed by matched Lock and Unlock operations.

General Ordering Rules (Appendix A.5):
These constraints require *order* to be transitive, total, and asymmetric (circuit-free).

4. EXECUTION VALIDATION

A direct application of the formal language specification is for execution validation. Studying thread behaviors with small code fragments (generally known as *litmus tests*) is very helpful for understanding the implications of a threading model. In fact, many memory model proposals rely on a collection of litmus tests to illustrate critical properties. In [9, 10], we have also demonstrated the effectiveness of abstracting a common programming pattern (such as the Double-Checked Locking algorithm or Peterson's algorithm) as a litmus test to support verification.

While defining the legality of thread behaviors is the common goal for all memory model specifications, the ability to automatically validate an execution has been lacking in previous declarative specification methods. Our system supports such an analysis by allowing a user to add annotations about the read values, and verifying those assertions automatically via constraint solving.

Constraint `validateExecution` verifies whether a given execution *ops* is legal under the formal model.

$$\textbf{validateExecution}\ ops\ \equiv (\exists\ order.\ \textbf{legalSC}\ ops\ order)$$

Concrete examples will be discussed in Section 7 to demonstrate how to apply such a formal specification to enable computer aided analysis.

5. RACE DETECTION

Race conditions are usually inadvertently introduced and may lead to unexpected behaviors that are hard to debug. Therefore, catching these potential defects is highly useful for developing reliable software. Furthermore, many relaxed memory systems guarantee that race-free programs behave in the same way as sequentially consistent programs, which allows programmers to rely on their intuitions about SC during software development. This also makes race-detection even more important in practice.

Our definition of a data race is according to [11], which has also been adopted by the new JMM draft [4]. In these proposals, a *happens-before* order (based on Lamport's *happened-before* order [12] for message passing systems) is used for formalizing *concurrent* memory accesses. Further, data-race-free programs (also referred to as *correctly synchronized programs*) are defined as being free of conflicting and concurrent accesses under all *sequentially consistent* executions. The reason for using SC executions to define data races is to make it easier for a programmer to determine whether a program is correctly synchronized.

We define constraint `detectDataRace` to catch any potential data races. This constraint attempts to find a total order *scOrder* and a *happens-before* order *hbOrder* such that there exists a pair of conflicting operations which are not ordered by *hbOrder*. This formalizes the notion of data races under sequentially consistent executions.

$$\begin{aligned}&\textbf{detectDataRace}\ ops\ \equiv \exists\ scOrder,\ hbOrder.\\&\quad \textbf{legalSC}\ ops\ scOrder \wedge\\&\quad \textbf{requireHbOrder}\ ops\ hbOrder\ scOrder \wedge\\&\quad \textbf{mapConstraints}\ ops\ hbOrder\ scOrder \wedge\\&\quad \textbf{existDataRace}\ ops\ hbOrder\end{aligned}$$

Happens-before order is defined in `requireHbOrder`. Intuitively, it states that two operations are ordered by happens-before order if (i) they are program ordered, (ii) they are ordered by synchronization operations, or (iii) they are transitively ordered by a third operation.

$$\begin{aligned}&\textbf{requireHbOrder}\ ops\ hbOrder\ scOrder\ \equiv\\&\quad \textbf{requireProgramOrder}\ ops\ hbOrder \wedge\\&\quad \textbf{requireSyncOrder}\ ops\ hbOrder\ scOrder \wedge\\&\quad \textbf{requireTransitiveOrder}\ ops\ hbOrder\end{aligned}$$

Since sequential consistency requires a total order among all operations, the happens-before edges induced by synchronization operations must follow this total order. This is captured by `requireSyncOrder`. Similarly, `mapConstraints` is used to make sure *scOrder* is consistent with *hbOrder*.

$$\begin{aligned}&\textbf{requireSyncOrder}\ ops\ hbOrder\ scOrder\ \equiv \forall\ i\ j \in ops.\\&\quad (\textbf{fb}\ i\ \wedge\ \textbf{fb}\ j\ \wedge\ \textbf{isSync}\ i\ \wedge\ \textbf{isSync}\ j\ \wedge\ \textbf{scOrder}\ i\ j)\\&\quad \Rightarrow \textbf{hbOrder}\ i\ j\end{aligned}$$

$$\begin{aligned}&\textbf{mapConstraints}\ ops\ hbOrder\ scOrder\ \equiv \forall\ i\ j \in ops.\\&\quad (\textbf{fb}\ i\ \wedge\ \textbf{fb}\ j\ \wedge\ \textbf{hbOrder}\ i\ j) \Rightarrow \textbf{scOrder}\ i\ j\end{aligned}$$

With a precise definition of happens-before order, we can formalize a race condition in constraint `existDataRace`. A race is caused by two feasible operation that are (i) conflicting, i.e., they access the same variable from different threads ($\textbf{t}\ i \neq \textbf{t}\ j$) and at least one of them is a write, and (ii) concurrent, i.e., they are not ordered by happens-before order.

$$\begin{aligned}&\textbf{existDataRace}\ ops\ hbOrder\ \equiv \exists\ i, j \in\ ops.\\&\quad \textbf{fb}\ i \wedge \textbf{fb}\ j\ \wedge\ \textbf{t}\ i \neq \textbf{t}\ j\ \wedge\ \textbf{var}\ i = \textbf{var}\ j\ \wedge\\&\quad (\textbf{op}\ i = Write\ \vee\ \textbf{op}\ j = Write) \wedge\\&\quad \neg(\textbf{hbOrder}\ i\ j)\ \wedge\ \neg(\textbf{hbOrder}\ j\ i)\end{aligned}$$

To support race analysis for the new JMM proposal, this race definition needs to be extended, e.g., by adding semantics for volatile variable operations.

6. ATOMICITY VERIFICATION

Atomicity ensures certain atomic transactions. If atomic-

ity can be verified, a compiler may ignore the fine-grained interleavings and apply standard sequential compilation techniques when treating an atomic block. However, race-freedom is neither *necessary* nor *sufficient* to ensure atomicity. As shown by the example in Figure 2, a monitor-style mutual exclusion mechanism, if used improperly, cannot guarantee atomicity even if the code is race-free. Therefore, a different mechanism is needed to specify and verify atomicity.

For this purpose, we allow a programmer to annotate an atomic block by enclosing it with keywords *AtomicEnter* and *AtomicExit*. To simplify some implementation details, we assume that the annotations are properly inserted. For the operation tuple, we add three more fields: *abEnter*, *abExit*, and *matchAbID*.

abEnter $i = abEnter$:	if i is the start of an atomic block;
abExit $i = abExit$:	if i is the end of an atomic block;
matchAbID $i = matchAbID$:	ID of the matching start of the atomic block.

During the preprocessing phase, we set up the operation i that immediately *follows* an *AtomicEnter* with **abEnter** $i = True$. Similarly, we set up the operation j that immediately *precedes* the matching *AtomicExit* with **abExit** $j = True$. We also record the *id* of i into the *matchAbID* field of j (**matchAbID** j = **id** i). Given an execution *ops* transformed from an annotated program, we can use constraint `verifyAtomicity` to catch atomicity violations.

verifyAtomicity *ops* $\equiv \exists$ *order*.
  legalSC *ops order* $\wedge$
  existAtomicityViolation *ops order*

existAtomicityViolation *ops order* $\equiv$
  $\exists\, i, j, k \in$ *ops*.
  (**fb** i $\wedge$ **fb** j $\wedge$ **fb** k $\wedge$
  abEnter i $\wedge$ **abExit** j $\wedge$
  id i = **matchAtID** j $\wedge$ **id** $i \neq$ **id** j $\wedge$
  isViolation k i $\wedge$
  $\neg$(**order** k i) $\wedge$ $\neg$(**order** j k))

isViolation k i $\equiv$ (**t** $k \neq$ **t** i)

The definition of `existAtomicityViolation` is generic, in that `isViolation` can be fine-tuned to capture other desired semantics. For illustration purposes, we only provide a very strong requirement here. It states that no operation from another thread can be interleaved between the atomic block. In practice, it is benign to interleave certain operations as long as the effect cannot be observed. For example, it might be desirable to define a “variable window” (a set of variables manipulated within an atomic block) and only detect an atomicity violation when the intruding operation “overlaps” the variable window.

7. IMPLEMENTATION

Constraint-based analyses can be quickly prototyped using a constraint logic programming language such as FD-Prolog[2]. We have built a tool named *DefectFinder*, written in SICStus Prolog [13], to test the proposed techniques.

7.1 Constraint Solver

Two mechanisms from FD-Prolog can be applied for solving the constraints in our specification. One applies backtracking search for all constraints expressed by logical variables, and the other uses non-backtracking constraint solving techniques such as *arc consistency* [14] for finite domain variables, which is potentially more efficient and certainly more complete (especially under the presence of negation) than with logical variables. This works by adding constraints in a monotonically increasing manner to a constraint store, with the built-in constraint propagation rules of FD-Prolog helping refine the variable ranges when constraints are asserted to the constraint store. In a sense, the built-in constraint solver from Prolog provides an effective means for bounded software model checking by explicitly exploring all program executions, but symbolically reasoning about the constraints imposed on free variables.

7.2 Constraint Generation

Translating the constraints specified in the Appendix to Prolog rules is straightforward. One caveat, however, is that most Prolog systems do not directly support quantifiers. While existential quantification can be realized via Prolog's backtracking mechanism, we need to implement universal quantification by enumerating the related finite domain. For instance, constraint `requireWeakTotalOrder` is originally specified as follows:

requireWeakTotalOrder *ops order* $\equiv \forall\, i, j \in$ *ops*.
  (**fb** i $\wedge$ **fb** j $\wedge$ **id** $i \neq$ **id** j) $\Rightarrow$ (**order** i j $\vee$ **order** j i)

In the Prolog code, predicate `forEachElem` is recursively defined to call the corresponding `elemProg` for every element in the adjacency matrix *Order* (variable names start with a capital letter in Prolog).

```
requireWeakTotalOrder(Ops,Order,FbList):-
    forEachElem(Ops,Order,FbList,doWeakTotalOrder).

elemProg(doWeakTotalOrder,Ops,Order,FbList,I,J):-
    const(feasible,Feasible),
    length(Ops,N),
    matrix_elem(Order,N,I,J,Oij),
    matrix_elem(Order,N,J,I,Oji),
    nth(I,FbList,Fi),
    nth(J,FbList,Fj),
    (Fi #= Feasible #/\ Fj #= Feasible #/\ I #\= J)
        #=> (Oij #\/ Oji).
```

One technique shown by the above example is worth noting. That is, the adjacency matrix *Order* and the feasibility list *FbList* are passed in as finite domain variables. The domain of the elements in these lists (which is *boolean* in this case) is previously set up in the top level predicate. Providing such domain information significantly reduces the solving time, hence is critical for the performance of the tool.

The search order among the constraints may also impact performance. In general, it is advantageous to let the solver satisfy the most restrictive goal first. For example, read value rules should precede the general ordering rules.

[2]FD-Prolog refers to Prolog with a finite domain (FD) constraint solver. For example, SICStus Prolog and GNU Prolog have this feature.

```
Tinit        Thread 1             Thread 2

(1)wr(x,0); (3)rd(x,r1,1);       (6)rd(y,r2,0);
(2)wr(y,0); (4)ctr(c1,[r1>0]);  (7)ctr(c2,[r2>=0]);
            (5)wr(y,1,[c1]);     (8)wr(x,1,[c2]);
```

Figure 4: The execution derived from program 2 in Figure 1 with $r1 = 1$ and $r2 = 0$.

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	0	X	1	1	1	1	1	1
2	X	0	1	1	1	1	1	1
3	0	0	0	1	1	0	0	0
4	0	0	0	0	1	0	0	0
5	0	0	0	0	0	0	0	0
6	0	0	1	1	1	0	1	1
7	0	0	1	1	1	0	0	1
8	0	0	1	1	1	0	0	0

Figure 5: The adjacency matrix for the execution shown in Figure 4 under sequential consistency.

7.3 Concurrency Analysis

DefectFinder is developed in a modular fashion and is highly configurable. It supports all three applications described in this paper. It also enables *interactive* and *incremental* analyses, meaning it allows users to selectively enable or disable certain constraints to help them understand the underlying model piece by piece.

To illustrate how the tool works, recall program 2 in Figure 1. Consider the problem of checking whether $r1 = 1$ and $r2 = 0$ is allowed by sequential consistency. Figure 4 displays the corresponding execution derived from the program text (it only shows the operation fields relevant to this example). When constraint `validateExecution` is imposed on this execution, DefectFinder immediately finds a legal order that satisfies the constraint and outputs its adjacency matrix, as shown in Figure 5. A matrix element $\mathcal{M}_{ij}$ can have a value of 0, 1, or X, where 0 indicates i is not ordered before j, 1 indicates i must precede j, and X means the ordering relation between i and j has not been instantiated based on the accumulated constraints. In general, there usually exist many X entries where alternative interleavings are allowed. If desired, a Prolog predicate *labeling* can be called to instantiate all variables. Our tool also outputs a possible interleaving *1 2 6 7 8 3 4 5* which is automatically derived from this matrix.

If the execution with $r1 = 0$ and $r2 = 1$ is checked, the tool would quickly determine that it is illegal since no ordering relation can be found to satisfy all constraints. The user can also ask "what if" queries by selectively commenting out some ordering rules to identify the root cause of a certain program behavior.

Applying DefectFinder for a different application simply involves selecting the corresponding goal. For example, if the programs in Figure 1 are checked for race conditions, the tool would report that program 1 is race-free and program 2 is not, in which case the conflicting operations and an interleaving that leads to the race conditions are displayed.

Similarly, when the program in Figure 2 is verified for race conditions, our utility would report that it is race-free. However, an atomicity violation would be detected if the transaction is annotated by an atomic block. Having detected this defect, the user can subsequently modify the code and do the test again. For instance, if a transaction is protected by a single Lock/Unlock pair and both transactions use the same lock, the bug would be removed.

7.4 Performance

Precise semantic analysis such as race detection is NP-hard in general [15]. Nonetheless, constraint-based methods have become very successful in practice, thanks to the efficient solving techniques developed in recent years.

Our tool has been applied to analyze a large collection of litmus tests — each of them is designed to reveal a certain memory model property or to simulate a common programming pattern. Figure 6 summarizes the performance results of the examples discussed in this paper. These analyses are performed using a Pentium 366 MHz PC with 128 MB of RAM running Windows 2000. SICStus Prolog is run under compiled mode. Our utility is available for download at `http://www.cs.utah.edu/~yyang/DefectFinder.zip`.

Test Program	Property	Result	Time (sec)
Program 1 in Figure 1	r1=1 and r2=0?	illegal	6.790
	Race Conditions	no races	6.810
Program 2 in Figure 1	r1=1 and r2=0?	legal	0.401
	Race Conditions	has races	0.811
Program in Figure 2	Race Conditions	no races	18.940
	Atomicity	violated	2.955

Figure 6: Performance statistics.

In terms of scalability, there are two aspects involved. One is the complexity of the shared memory system that can be modelled. The other is the size of programs that can be analyzed. For the former aspect, our system scales well with its compositional specification style. As demonstrated in [7], it is capable of formalizing memory ordering rules for cutting-edge commercial processors. As for the latter aspect, our CLP prototype tool currently only handles small litmus tests. However, there is still a lot of room for improvement, which offers an important but orthogonal task for future work. For instance, one can add a "constraint configuration" component that automatically filters out or reorders certain rules according to the input program, e.g., rules regarding control flow can be excluded if the program does not involve branch statements. Other solving techniques may also help make our approach more effective. We have shown in [7] that a slight variant of the Prolog code can let us benefit from a propositional SAT solver. In our recent work [16], we are developing efficient SAT encoding methods for analyzing larger programs. We are now in a position to handle nearly 500 memory operations.

8. RELATED WORK

Constraint solving was historically applied in AI planning problems. In recent years, it has started to show a lot of potential for program analysis as well. For example, constraints are used in [17] to analyze programs written in a factory control language called Relay Ladder Logic. A constraint system is developed in [18] for inferring static types for Java bytecode. The work in [19] performs points-to analysis for Java by employing annotated inclusion constraints.

Flanagan [20] proposed to use CLP for bounded software model checking. To the best of our knowledge, our work is the first to apply the constraint-based approach for capturing language-level memory models and reasoning about correctness properties in multithreaded programs.

Extensive research has been done in model checking Java programs, e.g., [21, 22, 23, 24]. These tools, however, do not specifically address memory model issues. Therefore, they cannot precisely analyze fine-grained thread interleavings. We can imagine our method being incorporated into these tools to make their analyses more accurate.

There is a large body of work on race detection, which can be classified as static or dynamic analysis. The latter can be further categorized as on-the-fly or post-mortem, depending on how the execution information is collected. Netzer and Miller [25] proposed a detection algorithm using the post-mortem method. Adve and Hill proposed the *data-race-free* model [26] and developed a formal definition of data races under weak memory models [11]. Lamport's happened-before relation has been applied in dynamic analysis tools, e.g., [27, 15, 28]. Several on-the-fly methods, e.g., [29, 30, 31], exploited information based on the underlying cache coherence protocol. The drawback of these dynamic techniques is that they can easily miss a data race, depending on how threads are scheduled. Our approach is based on the definition given in [11]. Our system also employs the happened-before relation, hence it is able to handle many different synchronization styles. Unlike the dynamic approaches, we use a static method that examines a symbolic execution to achieve an exhaustive coverage.

Some race detectors, e.g., [32, 33, 34], were designed specifically for the lock-based synchronization model. Tools such as ESC/Java [8] and Warlock [35] rely on user-supplied annotations to statically detect data races. Type-based approaches, e.g., [36, 37, 38], have also been proposed for object-oriented programs. While effective in practice, these tools do not address the issue that we focus on in this paper, which is how to rigorously reason about multithreaded programs running in a complex shared memory environment.

Flanagan and Qadeer [5] developed a type system to enforce atomicity based on Lipton's theory of right and left movers [39]. Since a race analysis is required to be performed in advance, the effectiveness of their approach depends on the accuracy of the race detector. It will be interesting to investigate if the requirements of movers can be captured as constraints for type inference.

9. CONCLUSION

We have presented a novel approach that handles both program semantics and memory model semantics in a declarative constraint-based framework. With three concrete applications — execution validation, race detection, and atomicity verification — we have demonstrated the feasibility and effectiveness of applying such a "memory-model-aware" analysis tool for verifying multithreaded programs that, albeit small, can be extremely difficult to analyze by hand. Our framework is particularly useful in helping people understand the underlying concurrency model and conduct verification for common programming patterns. The capability of studying program correctness under relaxed memory models is also essential in verifying critical components of important programs such as JVMs and garbage collectors that run on weak memory systems.

To summarize, our system offers the following benefits:

- It is rigorous. Based on formal definitions of program properties and memory model rules, our system enables a precise semantic analysis. Specifications developed in such a rigorous manner can also be sent to a theorem proving utility, such as the HOL theorem prover [40], for proving generic properties.

- It is automatic. Our approach allows one to take advantage of the tremendous advances in constraint/SAT solving techniques. The executable thread semantics can also be treated as a "black box" whereby the users are not necessarily required to understand all the details of the model to benefit from the tool.

- It is generic. Since our method is not limited to a specific synchronization mechanism, it can be applied to reason about various correctness properties for any threading model, all using the same framework.

Future Work: We plan to investigate divide-and-conquer style verification methods to make our system more scalable. Techniques developed in other tools, such as predicate abstraction, branch refinement, and assume-guarantee, can be integrated into our system. We also plan to explore more efficient solving techniques. In particular, the structural information of the constraints may be applied for improving the solving algorithms. We hope this paper can help pave the way towards future studies in these exciting areas.

10. REFERENCES

[1] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

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

[3] W. Pugh. The Java memory model is fatally flawed. *Concurrency: Practice and Experience*, 12(1):1–11, 2000.

[4] JSR133: Java memory model and thread specification. http://www.cs.umd.edu/˜pugh/java/memoryModel.

[5] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proceedings of PLDI*, 2003.

[6] Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, and Konrad Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. *The 18th International Parallel and Distributed Processing Symposium (IPDPS)*, to appear.

[7] Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, and Konrad Slind. Analyzing the Intel Itanium memory ordering rules using logic programming and SAT. In *Proceedings of the 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'03), LNCS 2860*, October 2003.

[8] C. Flanagan, K. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended static checking for Java, 2002.

[9] Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom. Analyzing the CRF Java Memory Model. In *Proceedings of the 8th Asia-Pacific Software Engineering Conference*, 2001.

[10] Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom. UMM: An operational memory model specification framework with integrated model checking capability. *Concurrency and Computation: Practice and Experience*, to appear.

[11] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting data races on weak memory systems. In *Proceedings of the 18th International Symposium on Computer Architecture (ISCA)*, pages 234–243, 1991.

[12] L. Lamport. Time, clocks and ordering of events in distributed systems. 21(7):558–565, July 1978.

[13] SICStus Prolog. http://www.sics.se/sicstus.

[14] J. Jaffar and J-L. Lassez. Constraint logic programming. In *Principles Of Programming Languages*, Munich, Germany, January 1987.

[15] Robert H. B. Netzer. Race condition detection for debugging shared-memory parallel programs. Technical Report CS-TR-1991-1039, 1991.

[16] Ganesh Gopalakrishnan, Yue Yang, and Hemanthkumar Sivaraj. QB or not QB: An efficient execution verification tool for memory orderings. In *Proceedings of Computer Aided Verification (CAV'04)*.

[17] Alexander Aiken, Manuel Fähndrich, and Zhendong Su. Detecting races in relay ladder logic programs. *LNCS*, 1384:184–200, 1998.

[18] Etienne Gagnon, Laurie J. Hendren, and Guillaume Marceau. Efficient inference of static types for java bytecode. In *Static Analysis Symposium*, pages 199–219, 2000.

[19] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for Java using annotated constraints. In *Proceedings of Object-Oriented Programming Systems, Lanuages, and Applications*, pages 43–55, 2001.

[20] Cormac Flanagan. Automatic software model checking using CLP. In *Proceedings of ESOP*, 2003.

[21] Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[22] W. Visser, K. Havelund, G. Brat, and S. Park. Java PathFinder - second generation of a Java Model Checker. In *Post-CAV Workshop on Advances in Verification, Chicago*, 2000.

[23] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code. In *International Conference on Software Engineering*, 2000.

[24] D. Park, U. Stern, and D. Dill. Java model checking. In *Proceedings of the First International Workshop on Automated Program Analysis, Testing and Verification, Limerick, Ireland*, 2000.

[25] R. H. Netzer and B. P. Miller. What are race conditions? Some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, 1992.

[26] S. V. Adve and M. D. Hill. A unified formalization of four shared-memory models. *IEEE Trans. on Parallel and Distributed Systems*, 4(6):613–624, 1993.

[27] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 85–96, 1991.

[28] D. Perkovic and P. Keleher. Online data-race detection via coherency guarantees. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 47–57, 1996.

[29] S. L. Min and J.-D. Choi. An efficient cache-based access anomaly detection scheme. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, pages 235–244, 1991.

[30] B. Richards and J. R. Larus. Protocol-based data-race detection. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, 1998.

[31] Edmond Schonberg. On-the-fly detection of access anomalies. In *Proceedings of PLDI*, pages 285–297, 1989.

[32] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[33] C. von Praun and T. Gross. Object-race detection. In *Proceedings of Object-Oriented Programming Systems, Lanuages, and Applications*, pages 70–82, 2001.

[34] J. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of PLDI*, 2002.

[35] N. Sterling. Warlock - a static data race analysis tool. *USENIX Winter Technical Conference*, pages 97–106, 1993.

[36] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. *Proceedings of PLDI*, pages 219–232, 2000.

[37] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: a dialect of Java without data races. In *Proceedings of Object-Oriented Programming Systems, Lanuages, and Applications*, 2000.

[38] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications*, 2001.

[39] R. Lipton. Reduction: a method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

[40] T. F. Melham M. J. C. Gordon. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

APPENDIX

A. SEQUENTIAL CONSISTENCY

legalSC *ops order* $\equiv$
requireProgramOrder *ops order* $\wedge$
requireReadValue *ops order* $\wedge$
requireComputation *ops order* $\wedge$
requireMutualExclusion *ops order* $\wedge$
requireWeakTotalOrder *ops order* $\wedge$
requireTransitiveOrder *ops order* $\wedge$
requireAsymmetricOrder *ops order*

A.1 Program Order Rule

requireProgramOrder *ops order* $\equiv \forall\, i, j \in ops.$
$(\mathbf{fb}\ i\ \wedge\ \mathbf{fb}\ j\ \wedge\ (\mathbf{orderedByProgram}\ i\ j\ \vee$
$\mathbf{t}\ i = t_{init}\ \wedge\ \mathbf{t}\ j \neq t_{init})) \Rightarrow \mathbf{order}\ i\ j$

A.2 Read Value Rules

requireReadValue *ops order* $\equiv$
globalReadValue *ops order* $\wedge$
localReadValue *ops order* $\wedge$
controlReadValue *ops order*

globalReadValue *ops order* $\equiv \forall\, k \in ops.$
$(\mathbf{fb}\ k\ \wedge\ \mathbf{isRead}\ k) \Rightarrow$
$(\exists\, i \in ops.\ \mathbf{fb}\ i\ \wedge\ \mathbf{op}\ i = Write\ \wedge\ \mathbf{var}\ i = \mathbf{var}\ k\ \wedge$
$\mathbf{data}\ i = \mathbf{data}\ k\ \wedge\ \neg(\mathbf{order}\ k\ i)\ \wedge$
$(\neg\exists\, j \in ops.\ \mathbf{fb}\ j\ \wedge\ \mathbf{op}\ j = Write\ \wedge\ \mathbf{var}\ j = \mathbf{var}\ k\ \wedge$
$\mathbf{order}\ i\ j\ \wedge\ \mathbf{order}\ j\ k))$

localReadValue *ops order* $\equiv \forall\, k \in ops.\ \mathbf{fb}\ k \Rightarrow$
$(\forall\, e \in (\mathbf{getLocals}\ k).$
$(\exists\, i \in ops.\ (\mathbf{fb}\ i \wedge \mathbf{isAssign}\ i \wedge \mathbf{local}\ i = \mathbf{getVar}\ e\ \wedge$
$\mathbf{data}\ i = \mathbf{getData}\ e\ \wedge\ \mathbf{orderedByProgram}\ i\ k)\ \wedge$
$(\neg\exists\, j \in ops.\ (\mathbf{fb}\ j \wedge \mathbf{isAssign}\ j \wedge \mathbf{local}\ j = \mathbf{getVar}\ e\ \wedge$
$\mathbf{orderedByProgram}\ i\ j \wedge \mathbf{orderedByProgram}\ j\ k))))$

controlReadValue *ops order* $\equiv \forall\, k \in ops.$
$(\forall\, e \in (\mathbf{getCtrs}\ k).$
$(\exists\, i \in ops.\ \mathbf{op}\ i = Control \wedge\ \mathbf{var}\ i = \mathbf{getVar}\ e\ \wedge$
$\mathbf{data}\ i = \mathbf{getData}\ e\ \wedge\ \mathbf{orderedByProgram}\ i\ k))$

A.3 Computation Rule

requireComputation *ops order* $\equiv \forall\, k \in ops.$
$((\mathbf{fb}\ k\ \wedge\ \mathbf{op}\ k = Computation) \Rightarrow$
$(\mathbf{data}\ k = \mathbf{eval}\ (\mathbf{cmpExpr}\ k)))\ \wedge$
$((\mathbf{fb}\ k\ \wedge\ \mathbf{op}\ k = Control) \Rightarrow$
$(\mathbf{data}\ k = \mathbf{eval}\ (\mathbf{ctrExpr}\ k)))$

A.4 Mutual Exclusion Rule

requireMutualExclusion *ops order* $\equiv \forall\, i, j \in ops.$
$(\mathbf{fb}\ i\ \wedge\ \mathbf{fb}\ j\ \wedge\ \mathbf{matchLock}\ i\ j) \Rightarrow$
$(\neg\exists\, k \in ops.\ \mathbf{fb}\ k\ \wedge\ \mathbf{isSync}\ k\ \wedge$
$\mathbf{lock}\ k = \mathbf{lock}\ i\ \wedge\ \mathbf{t}\ k \neq \mathbf{t}\ i\ \wedge\ \mathbf{order}\ i\ k\ \wedge\ \mathbf{order}\ k\ j)$

A.5 General Ordering Rules

requireWeakTotalOrder *ops order* $\equiv \forall\, i, j \in ops.$
$(\mathbf{fb}\ i\ \wedge\ \mathbf{fb}\ j\ \wedge\ \mathbf{id}\ i \neq \mathbf{id}\ j) \Rightarrow (\mathbf{order}\ i\ j\ \vee\ \mathbf{order}\ j\ i)$

requireTransitiveOrder *ops order* $\equiv \forall\, i, j, k \in ops.$
$(\mathbf{fb}\ i \wedge \mathbf{fb}\ j \wedge \mathbf{fb}\ k \wedge \mathbf{order}\ i\ j \wedge \mathbf{order}\ j\ k) \Rightarrow \mathbf{order}\ i\ k$

requireAsymmetricOrder *ops order* $\equiv \forall\, i, j \in ops.$
$(\mathbf{fb}\ i\ \wedge\ \mathbf{fb}\ j\ \wedge\ \mathbf{order}\ i\ j) \Rightarrow \neg(\mathbf{order}\ j\ i)$

A.6 Auxiliary Definitions

fb $i \equiv (\mathbf{eval}\ (\mathbf{ctrExpr}\ i) = True)$

orderedByProgram $i\ j \equiv (\mathbf{t}\ i = \mathbf{t}\ j\ \wedge\ \mathbf{pc}\ i < \mathbf{pc}\ j)$

isAssign $i \equiv (\mathbf{op}\ i = Computation\ \vee\ \mathbf{op}\ i = Read)$

isSync $i \equiv (\mathbf{op}\ i = Lock\ \vee\ \mathbf{op}\ i = Unlock)$

matchLock $i\ j \equiv$
$\mathbf{op}\ i = Lock\ \wedge\ \mathbf{op}\ j = Unlock\ \wedge\ \mathbf{matchID}\ j = \mathbf{id}\ i$

Note: for brevity, the following predicates are not explicitly defined here since they are typically well understood.

eval *exp*:	evaluate *exp* with standard program semantics;
getLocals *k*:	parse *k* and get the set of local variables that k depends on, with their associated data values;
getCtrs *k*:	parse the path predicate of *k* and get the set of control variables that k depends on, with their associated data values;
getVar *e*:	get variable from a ⟨variable, data⟩ entry;
getData *e*:	get data from a ⟨variable, data⟩ entry.

B. EXECUTION VALIDATION

validateExecution *ops* $\equiv \exists\ order.$ **legalSC** *ops order*

C. RACE DETECTION

detectDataRace *ops* $\equiv \exists\ scOrder,\ hbOrder.$
legalSC *ops scOrder* $\wedge$
requireHbOrder *ops hbOrder scOrder* $\wedge$
mapConstraints *ops hbOrder scOrder* $\wedge$
existDataRace *ops hbOrder*

requireHbOrder *ops hbOrder scOrder* $\equiv$
requireProgramOrder *ops hbOrder* $\wedge$
requireSyncOrder *ops hbOrder scOrder* $\wedge$
requireTransitiveOrder *ops hbOrder*

requireSyncOrder *ops hbOrder scOrder* $\equiv \forall\, i\ j \in ops.$
$(\mathbf{fb}\ i\ \wedge\ \mathbf{fb}\ j\ \wedge\ \mathbf{isSync}\ i\ \wedge\ \mathbf{isSync}\ j\ \wedge\ \mathbf{scOrder}\ i\ j)$
$\Rightarrow \mathbf{hbOrder}\ i\ j$

mapConstraints *ops hbOrder scOrder* $\equiv \forall\, i\ j \in ops.$
$(\mathbf{fb}\ i\ \wedge\ \mathbf{fb}\ j\ \wedge\ \mathbf{hbOrder}\ i\ j) \Rightarrow \mathbf{scOrder}\ i\ j$

existDataRace *ops hbOrder* $\equiv \exists\, i, j \in\ ops.$
$\mathbf{fb}\ i \wedge \mathbf{fb}\ j\ \wedge\ \mathbf{t}\ i \neq \mathbf{t}\ j\ \wedge\ \mathbf{var}\ i = \mathbf{var}\ j\ \wedge$
$(\mathbf{op}\ i = Write\ \vee\ \mathbf{op}\ j = Write)\ \wedge$
$\neg(\mathbf{hbOrder}\ i\ j)\ \wedge\ \neg(\mathbf{hbOrder}\ j\ i)$

D. ATOMICITY VERIFICATION

verifyAtomicity *ops* $\equiv \exists\ order.$
legalSC *ops order* $\wedge$
existAtomicityViolation *ops order*

existAtomicityViolation *ops order* $\equiv \exists\, i, j, k \in\ ops.$
$(\mathbf{fb}\ i\ \wedge\ \mathbf{fb}\ j \wedge\ \mathbf{fb}\ k\ \wedge$
$\mathbf{abEnter}\ i\ \wedge\ \mathbf{abExit}\ j\ \wedge$
$\mathbf{id}\ i\ = \mathbf{matchAtID}\ j\ \wedge\ \mathbf{id}\ i \neq \mathbf{id}\ j\ \wedge$
$\mathbf{isViolation}\ k\ i\ \wedge$
$\neg(\mathbf{order}\ k\ i)\ \wedge\ \neg(\mathbf{order}\ j\ k))$

isViolation $k\ i\ \equiv (\mathbf{t}\ k \neq \mathbf{t}\ i)$

Requirements for Programming Language Memory Models

Jeremy Manson and William Pugh
Department of Computer Science
University of Maryland, College Park
{jmanson, pugh}@cs.umd.edu

ABSTRACT

One of the goals of the designers of the Java programming language was that multithreaded programs written in Java would have consistent and well-defined behavior. This would allow Java programmers to understand how their programs might behave; it would also allow Java platform architects to develop their platforms in a flexible and efficient way, while still ensuring that Java programs ran on them correctly.

Unfortunately, Java's original *memory model*, which described the way in which Java threads interact through memory, was not defined in a way that allowed programmers and architects to understand the requirements for a Java system. As part of Java Specification Request (JSR) 133 [7], a new memory model has been defined for Java. This paper outlines how the requirements for a new memory model were established, and what those requirements are. It does not outline the model itself; it merely provides a rationale.

1. INTRODUCTION

The work in [13, 14] showed that the original semantics for Java's threading specification [6, §17] had serious problems. To address these issues, the Java programming language [6] has recently undergone a revision; it now provides greater flexibility for implementors and a clearer notion of what it means to write a correct program. The new specification is widely known as the *Java memory model*.

To provide a clearer semantics, the informal properties of the memory model had to be described. This was accomplished through a great deal of thinking, staring at white boards, and spirited debate. A careful balance had to be maintained. On one hand, it was necessary for the model to allow programmers to be able to reason carefully and correctly about their multithreaded code. On the other, it was necessary for the model to allow compiler writers, virtual machine designers and hardware architects to optimize code ruthlessly, possibly interfering with the intuitive results of a program.

At the end of this process, a consensus emerged as to what the informal requirements for a programming language memory model are. In this paper, we discuss these requirements in detail. We do not discuss how these requirements were met. For more details on the actual model, see [7].

For a more detailed record of the process of designing this memory model, it might be instructive for the reader to look at the Java memory model mailing list archives [8].

2. WHY A SEMANTICS?

In the past, multithreaded languages have not defined a full semantics for multithreaded code. Ada, for example, simply defines unsynchronized code as “erroneous” [1]. The reasoning behind this is that since such code is incorrect (on some level), no guarantees should be made when it occurs. What it means for code to be correctly synchronized should be fully defined; after that, nothing.

This is the same strategy that some languages take with array bounds overflow – unpredictable results may occur, and it is the programmer's responsibility to avoid these scenarios.

The problem with this strategy is one of security and safety. In an ideal world, all programmers would write correct code all of the time. However, this does not always happen. Programs frequently contain errors; not only does this cause code to misbehave, but it also allows attackers an easy way into a program. Buffer overflows, in particular, are frequently used to compromise a program's security. Program semantics must be carefully defined: otherwise, it becomes harder to track down errors, and easier for attackers to take advantage of those errors. If programmers don't know what their code is doing, programmers won't be able to know what their code is doing wrong.

The new Java memory model provides strong guarantees for correctly written code, but also provides a clear and definitive semantics for how code should behave when it is not correctly written.

3. SIMPLE REORDERING

Many of the most important optimizations that can be performed on a program involve reordering program statements. For example, superscalar architectures frequently reorder instructions to ensure that the execution units are all in use as much as possible. Even optimizations as ubiquitous as common subexpression elimination and redundant read elimination can be seen as reorderings: each evaluation of the common expression is conceptually “moved” to the point at which it is evaluated for the first time.

Initially, x == y == 0

Thread 1	Thread 2
1: r2 = x;	3: r1 = y
2: y = 1;	4: x = 2

May return r2 == 2, r1 == 1

Figure 1: Behaves Surprisingly

In a single threaded program, a compiler can (and, indeed, must) be careful that these program transformations not interfere with the possible results of the program. We refer to this as a compiler's maintaining of the *intra-thread semantics* of the program – a thread in isolation has to behave as if no code transformations occurred at all.

However, it is much more difficult to maintain a simple, straightforward semantics while optimizing multithreaded code. Consider Figure 1. It may appear that the result `r2 == 2, r1 == 1` is impossible. Intuitively, if `r2` is 2, then instruction 4 came before instruction 1. Further, if `r1` is 1, then instruction 2 came before instruction 3. So, if `r2 == 2` and `r1 == 1`, then instruction 4 came before instruction 1, which comes before instruction 2, which came before instruction 3, which comes before instruction 4. This is a cyclic execution, which is, on the face of it, absurd.

On the other hand, we must consider the fact that a compiler can reorder the instructions in each thread. If instruction 3 does not come before instruction 4, and instruction 1 does not come before instruction 2, then the result `r2 == 2` and `r1 == 1` is perfectly reasonable.

In fact, in Java, for performance reasons, we always allow actions that are not control or data dependent on each other in a program to be reordered. This leads us to our first requirement:

Reorder1 Independent actions can be reordered.

In a multithreaded context, doing this may lead to counter-intuitive results, like the one in Figure 1. However, it should be noted that this code is improperly synchronized: there is no ordering of the accesses by synchronization. When synchronization is missing, weird and bizarre results are allowed.

It should be noted that Reorder1 guarantees that independent actions can be reordered regardless of the order in which they appear in the program. It does **not** guarantee that two independent actions can always be reordered. For example, a write action clearly cannot be reordered out of a locking region. We shall see another example of how these reorderings are limited in Section 5.4.

4. GUARANTEES FOR CORRECTLY SYNCHRONIZED PROGRAMS

It is very difficult for programmers to reason about the kinds of transformations that compilers perform. One of the goals of the Java memory model is to provide programmers a mechanism that allows them not to have to reason about reorderings in a program.

For example, in the code in Figure 1, the programmer can only see the result of the reordering because the code is improperly synchronized. Our first goal is to ensure that this is the only reason that a programmer can see the result of a reordering.

We say that a program obeys *sequentially consistent semantics* (as defined in [10]) if the result of any execution is the same as if all of the actions in that execution took place in some total order that reflects the order of the program, and each read sees the last write to that variable that occurred in that order. If a program obeys sequentially consistent semantics, then no compiler or processor reorderings will be visible.

Initially, `x == y == 0`

Thread 1	Thread 2
`r1 = x;`	`r2 = y;`
`if (r1 != 0)`	`if (r2 != 0)`
`y = 42;`	`x = 42;`

Correctly synchronized, so `r1 == r2 == 0` is the only legal behavior

Figure 2: Surprising Correctly Synchronized Program

Two accesses (reads of or writes to) the same shared field or array element are said to be *conflicting* if at least one of the accesses is a write. A *data race* occurs in an execution of a program if there are conflicting actions in multiple threads in that execution that are not ordered by synchronization. The program in Figure 1 has data races on both `x` and `y`. A program is *correctly synchronized* if and only if all sequentially consistent executions are free of data races.

Having defined these terms, we can talk a little more about the guarantees we provide. One possibility would be that we could eliminate all reorderings. On contemporary systems, this would have too much of an impact on performance. However, it is perfectly reasonable to ensure that code reordering should only be visible between threads when those threads are involved in data races. Our first guarantee for programmers, therefore, applies to data-race-free programs:

DRF Correctly synchronized programs have sequentially consistent semantics.

Given this requirement, programmers need only worry about code transformations having an impact on their programs' results if those program contain data races.

This requirement leads to some interesting corner cases. For example, the code shown in Figure 2 (first described in [2]) is correctly synchronized. This may seem surprising, since it doesn't perform any synchronization actions. Remember, however, that a program is correctly synchronized if, when it is executed in a sequentially consistent manner, there are no data races. If this code is executed in a sequentially consistent way, each action will occur in program order, and neither of the writes will occur. Since no writes occur, there can be no data races: the program is correctly synchronized. A program transformation (such as an aggressive write speculation) that permitted this result would be disallowed.

5. SYNCHRONIZATION

We haven't really discussed how code can use explicit synchronization (in whatever form we give it) to make sure their code is correctly synchronized. The typical way of doing this is by using locking. Another way is to use *volatile* variables.

The properties of volatile variables arose from the need to provide a way to communicate between threads without the overhead of ensuring mutual exclusion. A very simple

Initially, `x == 0, ready == false`. `ready` is a volatile variable.

Thread 1	Thread 2
`x = 1;`	`if (ready)`
`ready = true`	`r1 = x;`

If `r1 = x;` executes, it will read 1.

Figure 3: Simple Use of Volatile Variables

example of their use can be seen in Figure 3. If `ready` were not volatile, the write to it in Thread 1 could be reordered with the write to `x`. This might result in `r1` containing the value 0. We define volatiles so that this reordering cannot take place; if Thread 2 reads `true` for `ready`, it must also read 1 for `x`.

Locks and unlocks work in a way similar to volatiles: actions that take place before an unlock must also take place before any subsequent locks on that monitor. The resulting property reflects the way synchronization is used to communicate between threads:

HB Volatile writes are ordered before subsequent volatile reads of the same variable. Unlocks are ordered before subsequent locks of the same monitor.

The word *subsequent* needs to be defined for HB. *Synchronization actions* include locks, unlocks, and reads of and writes to volatile variables. We have a total order over all synchronization actions in an execution of a program; this is called the *synchronization order*. An action `y` is subsequent to another action `x` if `x` comes before `y` in the synchronization order.

5.1 Happens-Before Consistency

We can describe a simple, interesting memory model using HB by abstracting a little from locks and unlocks.

A *happens-before relationship* between two actions is what enforces an ordering between those actions. For example, if one action occurs before another in the program order for a single thread, then the first action happens-before the second. The program has to be executed in a way that does not make it appear to the second that it occurred out of order with respect to the first.

This may seem at odds with the result in Figure 1. However, a "reordering" is only visible here if we assume that the program executed all of its actions in a single total order; the surprising behavior makes it appear as if the writes are occurring before the reads. If the individual threads are examined in isolation, no reordering is visible; it is simply not known where the values seen by the reads are written.

The basic principle at work here is that threads in isolation will appear to behave as if they are executing in program order; however, the memory model will tell you what values can be seen by a particular read;

Synchronization actions can create happens-before relationships between threads. In addition to the happens-before relationship between actions in a single thread, we also have (in accordance with HB)

- An unlock on a particular monitor happens-before a lock on that monitor that comes after it in the synchronization order.
- A write to a volatile variable happens-before a read of that volatile variable that comes after it in the synchronization order.
- A call to start a thread happens-before the actual start of that thread.
- The termination of a thread happens-before a join performed on that thread.
- Happens-before is transitive. That is, if `a` happens-before `b`, and `b` happens-before `c`, then `a` happens-before `c`.

We say that it is *happens-before consistent* for a read to see a write in an execution of a program in two cases. First, a read is happens-before consistent if the write happens-before the read and there is no intervening write to the same variable. So, if a write of 1 to `x` happens-before a write of 2, and the write of 2 happens-before a read, then that read cannot see the value 1. Second, it is happens-before consistent for the read to see the write if the write does not happen-before the read. If the read does not happen-before the write, then the read is allowed to see the write. This can happen, for example, if the write occurs in another thread (as in Figure 1).

If all of the reads in an execution see writes they are happens-before consistent to see, then we say that execution is happens-before consistent. Note that happens-before consistency implies that every read must see a write that occurs somewhere in the program.

Although it is simple, happens-before consistency is not a good memory model. Notice that the behavior we want to disallow in Figure 2 is happens-before consistent. If both writes occur, and both reads see them, then both reads see writes that they are allowed to see.

Nevertheless, happens-before consistency provides a good outer bound for our model; based on HB, all executions must be happens-before consistent. Later sections of this paper (mostly Section 7) discuss ways of locating a more exact bound; for now, we focus on how happens-before affects implementation.

5.2 Implementing Synchronization

At the abstract level, happens-before consistency provides a relatively simple memory model. In this section, we talk a little about how we implement happens-before guarantees.

A happens-before relationship can be thought of as an ordering edge with two points; we call the start point a *release*, and the end point an *acquire*. Unlocks and volatile writes are release actions, and locks and volatile reads are acquire actions.

An acquire ensures an ordering with a previous release. Consider an action that takes place before an acquire. It may or may not have been visible to actions that took place before the previous release, depending on how the threads are scheduled. If we move the access to after the acquire, we are simply saying that the access is definitely scheduled after the previous release. This is therefore a legal transformation. For example, in Figure 3, if there were a read of a normal variable that occurred before the read of `ready`, then it could be moved after the read of `ready`.

Similarly, the only thing that the release does is ensure an ordering with a subsequent acquire. Consider an action that takes place after a release. It may or may not be visible to

Initially, `v1 == v2 == 0`

Thread 1	Thread 2	Thread 3	Thread 4
`v1 = 1;`	`v2 = 2;`	`r1 = v1;`	`r3 = v2;`
		`r2 = v2;`	`r4 = v1;`

Is `r1 == r3 == 1, r2 == r4 == 0` legal behavior?

Figure 4: Volatiles Must Occur In A Total Order

Initially, `x == y == v == 0`, `v` is volatile.

Thread 1	Thread 2
r1 = x;	r3 = y;
v = 0;	v = 0;
r2 = v;	r4 = v;
y = 1;	x = 1;

Is the behavior `r1 == r3 == 1` possible?

Figure 5: Strong or Weak Volatiles?

Initially, `a == b == v == 0`, `v` is volatile.

Thread 1	Thread 2
`r1 = a;`	`do {`
`if (r1 == 0)`	`  r2 = b;`
`  v = 1;`	`  r3 = v;`
`else`	`} while (r2 + r3 < 1);`
`  b = 1;`	`a = 1;`

Correctly synchronized, so `r1 == 1` is illegal

Figure 6: Another Surprising Correctly Synchronized Program

particular actions after the subsequent acquire, depending on how the threads are scheduled. If we move the access to before the release, we are simply saying that the access is definitely scheduled before the next acquire. This is therefore also a legal transformation. For example, in Figure 3, if there were a write to a normal variable that occurred after the write to `ready`, then it could be moved before the write to `ready`.

All of this is simply a roundabout way of saying that accesses to normal variables can be reordered with a following volatile read or monitor enter, or a preceding volatile write or monitor exit, This implies that normal accesses can be moved inside locking regions, but not out of them; for this reason, we sometimes call this property *roach motel semantics*.

It is relatively easy for compilers to ensure this property; indeed, most do already. Processors, which also reorder instructions, often need to be given *memory barrier* instructions to execute at these points in the code to ensure that they do not perform the reordering. Processors often provide a wide variety of these barrier instructions – for information about which are needed on which processor and for which action, consult [11].

5.3 Additional Guarantees for Volatiles

Figure 4 gives us another interesting glimpse into the guarantees we provide to programmers. The reads of `v1` and `v2` should be seen in the same order by both Thread 3 and Thread 4. The memory model does not allow writes to volatiles to be seen in different orders by different threads. In fact, it makes a much stronger guarantee:

VolatileAtomicity All accesses to volatile variables are performed in a total order.

This is clear cut, implementable, and has the unique property that the original Java memory model not only came down on the same side, but was also clear on the subject.

Another issue that arises with volatiles has come to be known as *strong versus weak* volatility. There are two possible interpretations of volatile, according to the happens-before order:

- **Strong interpretation** There is a happens-before relationship from each write to each subsequent read of that volatile.
- **Weak interpretation** There is a happens-before relationship from each write to each subsequent read of that volatile that sees that write.

In Figure 5, under the weak interpretation, the read of `v` in each thread might see its own volatile write. If this were the case, then the happens-before edges would be redundant, and could be removed. The resulting code could behave much like the simple reordering example in Figure 1.

To avoid confusion stemming from when multiple writer threads are communicating to reader threads via a single volatile variable, Java supports the strong interpretation.

StrongVolatile There must be a happens-before relationship from each write to each subsequent read of that volatile.

5.4 Optimizers Must Be Careful

Optimizers have to consider volatile accesses as carefully as they consider locking. In Figure 6, we have a correctly synchronized program. When executed in a sequentially consistent way, Thread 2 will loop until Thread 1 writes to `v` or `b`. Since the only value available for the read of `a` to see is 0, `r1` will have that value. As a result, the value 1 will be written to `v`, not `b`. There will therefore be a happens-before relationship between the read of `a` in Thread 1 and the write to `a` in Thread 2.

Knowing that the write to `a` will always happen, we might want to apply the principle that we can reorder the write to `a` with the loop. In this case, Thread 1 would be able to see the value 1 for `a`, and write to `b`. Thread 2 would see the write to `b` and terminate the loop. Since `b` is not a volatile variable, there would be no ordering between the read in Thread 1 and the write in Thread 2. There would therefore be data races on both `a` and `b`.

The result of this would be a correctly synchronized program that does not behave in a sequentially consistent way. This violates DRF, so we do not allow it. The need to prevent this sort of reordering caused many difficulties in formulating a workable memory model.

Compiler writers need to be very careful when reordering code past all synchronization points, not just those involving locking and unlocking.

Before compiler transformation

Initially, `a = 0, b = 1`

Thread 1	Thread 2
`1: r1 = a;`	`5: r3 = b;`
`2: r2 = a;`	`6: a = r3;`
`3: if (r1 == r2)`	
`4:   b = 2;`	

Is `r1 == r2 == r3 == 2` possible?

After compiler transformation

Initially, `a = 0, b = 1`

Thread 1	Thread 2
`4: b = 2;`	`5: r3 = b;`
`1: r1 = a;`	`6: a = r3;`
`2: r2 = r1;`	
`3: if (true) ;`	

`r1 == r2 == r3 == 2` is sequentially consistent

Figure 7: Effects of Redundant Read Elimination

5.5 Optimizations Based on Happens-Before

Notice that lock and unlock actions only have happens-before relationships with other lock and unlock actions **on the same monitor**. Similarly, accesses to a volatile variable only create happens-before relationships with accesses to the same volatile variable.

There have been many optimizations proposed (for example, in [15]) that have tried to remove excess, or "redundant" synchronization. One of the requirements of the Java memory model was that redundant synchronization (such as locks that are only accessed in a single thread) could be removed.

One possible memory model would require that all synchronization actions have happens-before relationships with all other synchronization actions. If we forced all synchronization actions to have happens-before relationships with each other, none of them could ever be described as redundant – they would all have to interact with the synchronization actions in other threads, regardless of what variable or monitor they accessed. Java does not support this; it does not simplify the programming model sufficiently to warrant the additional synchronization costs.

This is therefore another of our guarantees:

RS Synchronization actions that only introduce redundant happens-before edges can be treated as if they don't introduce any happens-before edges.

This is reflected in the definition of happens-before. For example, a lock that is only accessed in one thread will only introduce happens-before relationships that are already captured by the program order edges. This lock is redundant, and can therefore be removed.

6. TRANSFORMATIONS THAT INVOLVE DEPENDENCIES

In Section 3, we gave Reorder1, which is a guarantee that independent actions can be reordered. Reorder1 is a strong guarantee, but not quite strong enough. Sometimes, compilers can perform transformations that have the effect of removing dependencies.

For example, the behavior shown in Figure 7 is allowed. The compiler should be allowed to

- eliminate the redundant read of `a`, replacing `r2 = a` with `r2 = r1`, then
- determine that the expression `r1 == r2` is always true, eliminating the conditional branch `3`, and finally
- move the write `4: b = 2` early.

Initially, `x == y == 0`

Thread 1	Thread 2
`r1 = x;`	`r3 = y;`
`r2 = 1 + r1*r1 - r1;`	`x = r3;`
`y = r2;`	

`r1 == r2 == r3 == 1` is legal behavior

Figure 8: Compilers Can Think Hard About When Actions Are Guaranteed to Occur

Initially, `x == y == 0`

Thread 1	Thread 2
`r1 = x;`	`r3 = y;`
`if (r1 == 1)`	`if (r2 == 1)`
`  y = 1;`	`  x = 1;`
	`if (r2 == 0)`
	`  x = 1;`

`r1 == r2 == 1` is legal behavior

Figure 9: Sometimes Dependencies are not Obvious

After the compiler does the redundant read elimination, the assignment `4: b = 2` is guaranteed to happen; the second read of `a` will always return the same value as the first. Without this information, the assignment seems to cause itself to happen. With this information, there is no dependency between the reads and the write. Thus, dependence-breaking optimizations can also lead to apparent cyclic executions.

Note that intra-thread semantics guarantee that if `r1` $\neq$ `r2`, then Thread 1 will not write to `b` and `r3 == 1`. Additionally, either `r1 == 0, r2 == 1`, or `r1 == 1, r2 == 0`.

Figure 8 shows another surprising behavior. In order to see the result `r1 == r2 == 1`, it would seem as if Thread 1 would need to write 1 to `y` before reading `x`. However, it also seems as if Thread 1 can't know what value `r2` will be until after `x` is read.

In fact, it is easy for the compiler to perform an inter-thread analysis that shows that only the values 0 and 1 will be written to `x`. Knowing that, the compiler can determine that the quadratic equation always returns 1, resulting in Thread 1's always writing 1 to `y`. Thread 1 may, therefore, write 1 to `y` before reading `x`. The write to `y` is not dependent on the values seen for `x`. Our analysis of the program reveals that there is no real dependency in Thread 1.

A similar example of an apparent dependency can be seen in the code in Figure 9. In the same way as it does for Figure 8, a compiler can determine that only the values 0 and 1 are ever written to `x`. As a result, the compiler can remove

Initially, x = 0

Thread 1	Thread 2
r1 = x;	r2 = x;
x = 1;	x = 2;

r1 == 2 and r2 == 1 is a legal behavior

Figure 10: An Unexpected Reordering

Initially, x == y == 0

Thread 1	Thread 2
r1 = x;	r2 = y;
y = r1;	x = r2;

Incorrectly Synchronized: But r1 == r2 == 42 Still Cannot Happen

Figure 11: An Out Of Thin Air Result

the dependency and move the write to x to the beginning of Thread 2. If the resulting code were executed in a sequentially consistent way, it would result in the circular behavior described.

It is clear, then, that compilers can perform many optimizations that remove dependencies. So we make another guarantee:

Reorder2 If a compiler can detect that an action will always happen (with the same value written to the same variable), it can be reordered regardless of apparent dependencies.

Like Reorder1, this guarantee does not allow an implementation to reorder actions around synchronization actions arbitrarily. In Figure 6, for example, we saw an example of this: we could not reorder the accesses to a because of the happens-before relationships.

Even though Reorder1 and Reorder2 are strong guarantees for compilers, they are not a complete set of reorderings allowed. They are simply a set that is always guaranteed to be allowed.

6.1 Reordering Not Visible to Current Thread

Figure 10 contains a small but interesting example. The behavior r1 == 2 and r2 == 1 is a legal behavior, although it may be difficult to see how it could occur. A compiler would not reorder the statements in each thread; this code must never result in r1 == 1 or r2 == 2. However, the behavior r1 == 2 and r2 == 1 might be allowed by an optimizer that performs the writes early, but does so without allowing them to be visible to local reads that came before them in program order. This behavior, while surprising, is allowed by several processor memory architectures, and therefore is one that should be allowed by a programming language memory model.

7. OUT-OF-THIN-AIR GUARANTEES

In Figure 2, the writes are control dependent on the reads. Figure 11 is a very similar example; in this case, the writes will always happen, but the values written are data dependent on the reads.

This is no longer a correctly synchronized program, because there is a data race between Thread 1 and Thread 2. However, as it is in many ways a very similar example, we would like to provide a similar guarantee. In this case, we say that the value 42 cannot appear *out of thin air*.

In fact, the behavior of this case may be even more of a cause for concern than the other. If, for example, the value that was being produced out of thin air was a reference to an object which the thread was not supposed to have, then such a transformation could be a serious security violation. There are no reasonable compiler transformations that produce this result.

An example of this can be seen in Figure 12. Let's assume that there is some object o which we do not wish Thread 1 or Thread 2 to see. o has a self-reference stored in the field f. If our compiler were to decide to perform an analysis that assumed that the reads in each thread saw the writes in the other thread, and saw a reference to o, then r1 = r2 = r3 = o would be a possible result. The value did not spring from anywhere – it is simply an arbitrary value pulled out of thin air.

Determining what constitutes an out-of-thin-air read is complicated. A first (but inaccurate) approximation would be that we don't want reads to see values that couldn't be written to the variable being read in some sequentially consistent execution. Because the value 42 is never written in Figure 11, no read can ever see it.

The problem with this solution is that a program can contain writes whose program statements don't occur in any sequentially consistent executions. Imagine, as an example, a write that is only performed if the value of r1 + r2 is equal to 3 in Figure 1. This write would not occur in any sequentially consistent execution, but we would still want a read to be able to see it.

One way to think about these issues is to consider when actions can occur in an execution. These transformations all involve moving actions earlier than they would otherwise have occurred. You can perform an action earlier in an execution than it would have otherwise occurred if, had we carried on the execution in a sequentially consistent way, it would have been possible for the action to have occurred afterward.

If we had, for example, a write that was control dependent on the value of r1 + r2 being equal to 3 in Figure 1, we would know that write could have occurred in an execution of the program that behaves in a sequentially consistent way after the result of r1 + r2 is determined.

We can apply this form of reasoning to our other example, as well. In Figure 1, the writes to x and y can occur first because they will always occur in sequentially consistent executions. In Figure 7, the write to b can occur early because it occurs in a sequentially consistent execution when r1 and r2 see the same value. In Figure 11, the writes of 42 to y and x cannot happen, because they do not occur in any sequentially consistent execution. This, then, is our first "out of thin air" guarantee:

ThinAir1 A write can occur earlier in an execution than it would have otherwise occurred. However, that write must have been able to occur without the assumption that any reads that take place after the point where the write occurs see non-sequentially consistent values.

7.1 When Actions Can Occur

7.1.1 Disallowing Some Results

It is difficult to define the boundary between the kinds of results that are reasonable and the kind that are not. The example in Figure 11 provides an example of a result

Initially, `x = null, y = null`.
`o` is an object with a field `f` that refers to `o`.

Thread 1	Thread 2
`r1 = x;`	`r3 = y;`
`r2 = x.f;`	`x = r4;`
`y = r2;`	

`r1 == r2 == o` is not an acceptable behavior

Figure 12: An Unexpected Reordering

Initially, `x == y == z == 0`

Thread 1	Thread 2
`r3 = x;`	`r2 = y;`
`if (r3 == 0)`	`x = r2;`
`  x = 42;`	
`r1 = x;`	
`y = r1;`	

`r1 == r2 == r3 == 42` is a legal behavior

Figure 15: A Complicated Inference

that is clearly unacceptable, but other examples may be less straightforward.

The examples in Figures 13 and 14 are similar to the examples in Figures 2 and 11, with one major distinction. In those examples, the value 42 could never be written to `x` in any sequentially consistent execution. In the examples in Figures 13 and 14, 42 can be written to `x` in some sequentially consistent executions. Could it be legal for the reads in Threads 1 and 2 to see the value 42 even if Thread 4 does not write that value?

This is a potential security issue. Consider what happens if, instead of 42, we write a reference to an object that Thread 4 controls, but does not want Threads 1 and 2 to see without Thread 4's first seeing 1 for `z`. If Threads 1 and 2 see this reference, they can be said to manufacture it out of thin air.

This sort of behavior is not known to result from any combination of known reasonable and desirable optimizations. However, there is also some question as to whether this reflects a real and serious security requirement. In Java, the semantics usually side with the principle of having safe, simple and unsurprising semantics when possible. Thus, the Java Memory Model prohibits the behaviors shown in Figures 13 and 14.

7.1.2 Allowing Other Results

Now consider the code in Figure 15. A compiler could determine that the only values ever assigned to `x` are 0 and 42. From that, the compiler could deduce that, at the point where we execute `r1 = x`, either we had just performed a write of 42 to `x`, or we had just read `x` and seen the value 42. In either case, it would be legal for a read of `x` to see the value 42. By the principle we articulated as Reorder2, it could then change `r1 = x` to `r1 = 42`; this would allow `y = r1` to be transformed to `y = 42` and performed earlier, resulting in the behavior in question.

This is a reasonable transformation that needs to be balanced with the out-of-thin-air requirement. Notice that the code in Figure 15 is quite similar to the code in Figures 13 and 14. The difference is that Threads 1 and 4 are now joined together; in addition, the write to `x` that was in Thread 4 is now performed in every sequentially consistent execution – it is only when we try to get non-sequentially

Initially, `x = y = 0; a[0] = 1, a[1] = 2`

Thread 1	Thread 2
`r1 = x;`	`r3 = y;`
`a[r1] = 0;`	`x = r3;`
`r2 = a[0];`	
`y = r2;`	

`r1 == r2 == r3 == 1` is unacceptable

Figure 16: Another Out Of Thin Air Example

consistent results that the write does not occur.

There is a significant difference between these two cases. One way of articulating it is that in Figure 15, we know that `r1 = x` can see 42 without reasoning about what might have occurred in another thread because of a data race. In Figures 13 and 14, we need to reason about the outcome of a data race to determine that `r1 = x` can see 42.

This is, then, what differentiates out of thin air reads from those that are allowable. A solution must be available that does not involve reasoning about what happens in the execution solely because of data races. This is also our second out of thin air principle:

ThinAir2 Actions may only be performed earlier than their original place in the program if it can be determined that they could occur in the execution without assuming that any additional reads see values via a data race.

We can use ThinAir2 as a basic principle to reason about multithreaded programs. Consider, for example, the code in Figure 16. The only way in which the unacceptable result could occur is if a write of 1 to one of the variables were performed early. However, we cannot reason that a write of 1 to `x` or `y` will occur without reasoning about data races. Therefore, this result is impossible.

7.2 Isolation

Sometimes, when debugging a program, we are given an execution trace of that program in which the error occurred. Given a particular execution of a program, the debugger can create a *partition* of the threads and variables in the program so that if a thread accessed a variable in that execution, then the thread and variable are in the same partition. Monitors can be included in with variables for the purposes of this discussion.

Given this partitioning, you can explain the behavior in the execution of the threads in each partition without having to examine the behavior or code for the other threads. If a thread or a set of threads is isolated from the other threads in an execution, the programmer can reason about that isolated set separately from the other threads. This is called the *isolation* principle:

Isolation Consider a partition P of the threads and variables in the program so that if a thread accessed a variable in that execution, then the thread and variable are in the same partition. Given P, you can explain the behavior in the execution of the threads in each partition without having to examine the behavior or code for the other threads.

How is this helpful? Consider the code in Figure 14. If we allowed the unacceptable execution, then we could say

Initially, x == y == z == 0

Thread 1	Thread 2	Thread 3	Thread 4
r1 = x;	r2 = y;	z = 42;	r0 = z;
y = r1;	x = r2;		x = r0;

Is r0 == 0, r1 == r2 == 42 legal behavior?

Figure 13: Can Threads 1 and 2 see 42, if Thread 4 didn't write 42?

Initially, x == y == z == 0

Thread 1	Thread 2	Thread 3	Thread 4
r1 = x;	r2 = y;	z = 1;	r0 = z;
if (r1 != 0)	if (r2 != 0)		if (r0 == 1)
y = r1;	x = r2;		x = 42;

Is r0 == 0, r1 == r2 == 42 legal behavior?

Figure 14: Can Threads 1 and 2 see 42, if Thread 4 didn't write to x?

Initially, a = b = c = d = 0

```
Thread 1/2/3                  Thread 4
r1 = a;
if (r1 == 0)
  b = 1;                      r4 = d;
r2 = b;                       if (r4 == 1)  {
if (r2 == 1)                    c = 1;
  c = 1;                        a = 1;
r3 = c;                       }
if (r3 == 1)
  d = 1;
```

Behavior in question: r1 == r3 == r4 == 1; r2 == 0

Figure 18: Result of thread inlining of Figure 17; behavior allowed by semantics

that the actions in Threads 3 and 4 affected the actions in Threads 1 and 2, even though they touched none of the same variables. Reasoning about this would be difficult, at best.

The Isolation principle closely interacts with our out of thin air properties. If a thread A does not access the variables accessed by a thread B, then the only way A could have really affected B is if A might have accessed those variables along another program path not taken. The compiler might speculate that the other program path would be taken, and that speculation might affect B. The speculation could only really affect B if B could happen at the same time as A. This would imply a data race between A and B, and we would be speculating about that race; this is something ThinAir2 is designed to avoid.

Isolation is not necessarily a property that should be required in all memory models. It seems to capture a property that is useful and important in a memory model, but all of the implications of it are not understood well enough for us to decide if it must be true of any acceptable memory model.

7.3 Thread Inlining

One behavior that is disallowed by a straightforward interpretation of the out of thin air property that we have developed is shown in Figure 17. An implementation that always scheduled Thread 1 before Thread 2 and Thread 2 before Thread 3 could reasonably decide that the write to `d` by Thread 3 could be performed before anything in Thread 1 (as long as the guard `r3 == 1` evaluates to true). This could lead to a result where the write to `d` occurs, then Thread 4 writes 1 to `c` and `a`. The write to `b` does not occur, so the read of `b` by Thread 2 sees 0, and does not write to `c`. The read of `c` in Thread 3 then sees the write by Thread 4.

However, this requires reasoning that Thread 3 will see a value for `c` that is given by a data race. A straightforward interpretation of ThinAir2 therefore disallows this.

In Figure 18, we have another example, similar to the one in Figure 17, where Threads 1, 2 and 3 are combined. We can use the same reasoning that we were going to use for Figure 17 to decide that the write to `d` can occur early. Here, however, it does not clash with ThinAir2: we are only reasoning about the actions in the combined Thread 1/2/3. The behavior is therefore allowed in this execution.

As a result of this distinction, the compiler writer must be careful when considering inlining threads. When a compiler does decide to inline threads, as in this example, it may not be possible to utilize the full flexibility of the Java memory model when deciding how the resulting code can execute.

8. RELATED WORK

The happens-before relationship has a long history in concurrency literature. It is first described in [9].

The notion that correctly synchronized programs should behave in a sequentially consistent way was first articulated in [3].

An earlier, substantially simpler version of this work appeared in [12]. It did not address the full range of causality issues addressed here.

Most multithreaded languages do not provide strong semantics for multithreaded programs in the presence of data races. The Ada programming language [1], for example, refers to such programs as "erroneous", and discusses them no further. The C# language's [4] underlying framework, the Common Language Infrastructure [5], is also multithreaded; it explicitly allows optimizations to take place when there are data races, but it does not offer specific semantics.

9. CONCLUSION

The adoption of a new Java memory model was a long process. In order to carefully define the requirements, the needs of programmers, compiler writers and processor architects had to be carefully balanced. The end result is a strong statement, not only of what the requirements are for Java (as listed in Figure 19), but one that identifies and classifies these issues for future memory models.

10. ACKNOWLEDGMENTS

Initially, a = b = c = d = 0

Thread 1	Thread 2	Thread 3	Thread 4
			`r4 = d;`
`r1 = a;`	`r2 = b;`	`r3 = c;`	`if (r4 == 1) {`
`if (r1 == 0)`	`if (r2 == 1)`	`if (r3 == 1)`	`c = 1;`
`b = 1;`	`c = 1;`	`d = 1;`	`a = 1;`
			`}`

Behavior in question: r1 == r3 == r4 == 1; r2 == 0

Figure 17: Behavior disallowed by semantics

The authors wish to thank the members of the Java memory model mailing list for their time and contribution to this effort. Gratitude is particularly extended to Doug Lea and Sarita Adve for their contributions. Gramercy, also, to David Hovemeyer, for his feedback on this paper.

11. REFERENCES

[1] Ada Joint Program Office. *Ada 95 Rationale*. Intermetrics, Inc., Cambridge, Massachusetts, 1995.

[2] Sarita Adve. Designing memory consistency models for shared-memory multiprocessors. Technical Report 1198, University of Wisconsin, Madison, December 1993. Ph.D. Thesis.

[3] Sarita Adve and Mark Hill. Weak ordering—A new definition. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 2–14, 1990.

[4] ECMA. C# Language Specification, December 2002. http://www.ecma-international.org/publications/standards/Ecma-334.htm.

[5] ECMA. Common Language Infrastructure (CLI), December 2002. http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[6] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.

[7] Java Specification Request (JSR) 133. Java Memory Model and Thread Specification Revision, 2004. http: //jcp.org/jsr/detail/133.jsp.

[8] The Java memory model. Mailing list and web page. http://www.cs.umd.edu/ users/ pugh/ java/ memoryModel.

[9] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–564, 1978.

[10] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 9(29):690–691, 1979.

[11] Doug Lea. JSR-133 Cookbook, 2004. Available from http: //gee.cs.oswego.edu/dl/jmm/cookbook.html.

[12] Jeremy Manson and William Pugh. Core semantics of multithreaded Java. In *ACM Java Grande Conference*, June 2001.

[13] William Pugh. Fixing the Java memory model. In *ACM Java Grande Conference*, June 1999.

[14] William Pugh. The Java memory model is fatally flawed. *Concurrency: Practice and Experience*, 12(1):1–11, 2000.

[15] Eric Ruf. Effective Synchronization Removal for Java. In *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, Vancouver, BC Canada, June 2000.

Name	Description	Exemplar
Guarantees for Optimizers		
Reorder1	Independent actions can be reordered.	Figure 1
Reorder2	If a compiler can detect that an action will always happen (with the same value written to the same variable), it can be reordered regardless of apparent dependencies.	Figures 7, 8, 9, 10, 15
RS	Synchronization actions that only introduce redundant happens-before edges can be treated as if they don't introduce any happens-before edges.	Section 5.5
Guarantees for Programmers		
DRF	Correctly synchronized programs have sequentially consistent semantics.	Figures 2, 6
HB	Volatile writes are ordered before subsequent volatile reads of the same variable. Unlocks are ordered before subsequent locks of the same monitor.	Figure 3
VolatileAtomicity	All accesses to volatile variables are performed in a total order.	Figure 4
StrongVolatile	There is a happens-before relationship from each write to each subsequent read of that volatile.	Figure 5
ThinAir1	A write can only occur earlier in an execution than it would have otherwise occurred if that write would have occurred without assuming that any additional, later reads see non-sequentially consistent values.	Figures 11, 12, 16
ThinAir2	Actions may only be performed earlier than their original place in the program if it can be determined that they could occur in the execution without assuming that any additional reads see values via a data race.	Figures 13, 14, 17, 18
Isolation	Consider a partition P of the threads and variables in the program so that if a thread accessed a variable in that execution, then the thread and variable are in the same partition. Given P, you can explain the behavior in the execution of the threads in each partition without having to examine the behavior or code for the other threads.	Figures 13, 14

Figure 19: Properties of the Java Memory Model

Exceptions and side-effects in atomic blocks

Tim Harris
University of Cambridge Computer Laboratory
15 JJ Thomson Avenue
Cambridge, UK, CB3 0FD
tim.harris@cl.cam.ac.uk

ABSTRACT

In our paper at OOPSLA 2003 we discussed the design and implementation of a new `atomic` keyword as an extension to the Java programming language. It allows programs to perform a series of heap accesses atomically without needing to use mutual exclusion locks. We showed that data structures built using it could perform well and scale to large multi-processor systems. In this paper we extend our system in two ways. Firstly, we show how to provide an explicit 'abort' operation to abandon execution of an atomic block and to automatically undo any updates made within it. Secondly, we show how to perform external I/O within an atomic block. Both extensions are based on a single 'external action' abstraction, allowing code running within an atomic block to request that a given pre-registered operation be executed outside the block.

1. INTRODUCTION

In recent work we have been investigating the use of Software Transactional Memory as a mechanism for implementing language-level concurrency control features [6]. In our system, developed as an extension to the Java programming language, we have introduced a new keyword `atomic` which allows a group of statements to execute atomically with respect to the operation of other threads. As well as updating objects' fields, these statements can perform a wide range of operations including invoking methods and instantiating new objects. We also allow `atomic` statements to be guarded by boolean conditions, with execution blocking until the condition is satisfied. Figure 1 illustrates this by showing the implementation of a single-cell shared buffer.

In this paper we expand the range of operations which can be performed within `atomic` blocks in two different ways. The first extension we consider is what behaviour to provide when an `atomic` block terminates early by an exception being thrown. The dilemma here is whether to roll-back updates made in the `atomic` block or whether to retain them and propagate the exception. Unfortunately there is a Catch-22 situation: if we roll-back the updates then the exception object itself could be lost, leaving nothing to propagate. We discuss this in Section 2 and propose a hybrid model in which certain exceptions cause `atomic` blocks to be aborted and in which the exception thrown outside the block is a deep copy of the exception raised within it.

The second area we investigate is how to deal with I/O performed within an `atomic` block: our original design forbade any native method invocations which made most I/O operations unavailable. In Section 3 we discuss a number of ways in which I/O could be supported and propose a model in which communication libraries must be adapted for use within `atomic` blocks. This places an onus on the library's implementer but, we argue, allows better performance and scalability than automatic support for native methods.

```
class Buffer {
  private boolean full;
  private int value;

  public void put(int new_value)
                  throws InterruptedException
  {
    atomic (!full) { // Wait until buffer is empty
      full = true;
      value = new_value;
    }
  }

  public int get() throws InterruptedException
  {
    atomic (full) { // Wait until buffer is full
      full = false;
      return value;
    }
  }
}
```

Figure 1: A single-cell shared buffer implemented using atomic blocks.

Our approach for supporting both of these new features is based on a single 'external action' abstraction which we introduce in Section 4. An external action object exports an operation which can be invoked from within an `atomic` block but which is executed within the context that the object was instantiated.

In Section 5 we discuss our experience using external actions to implement our exception-propagation model and I/O system. Finally, Section 6 discusses related work and Section 7 concludes, highlighting a number of areas for future work along with dead-ends we explored in developing the 'external action' abstraction.

In the remainder of this introduction we briefly review the intended semantics of atomic blocks in Section 1.1 and outline their implementation over a Software Transactional Memory in Section 1.2.

1.1 Intended semantics of atomic blocks

We informally define the semantics of non-nesting `atomic` blocks by (*i*) specifying their behaviour when executed by a single thread running in isolation and (*ii*) requiring that, in a multi-threaded system, they behave as-if the executing thread ran in isolation while within the block.

There are two cases to consider based on whether or not the atomic block contains a guard condition. If there is no guard condition then the following two code fragments are equivalent:

```
atomic {
  S;                                { S; }
}
```

Similarly, if a guard condition is present then the following two code fragements behave equivalently *after blocking until the guard* `E` *is presciently known to yield true or terminate with an exception*:

```
atomic (E) {
  S;                                { E; S; }
}
```

These definitions have three major consequences. Firstly, they mean that if a system is genuinely single-threaded then the contents of an `atomic` block can be executed directly when its guard is satisfied. Secondly, these definitions lead to the semantics for exception propagation in our original paper – that is, if `E` or `S` terminates with an exception then the updates made up to that point are retained [6]. Thirdly, these definitions allow the guard expression `E` to have side effects – this may be important in practice if, for example, the guard accesses a self-organizing data structure such as a splay tree [4].

There are numerous subtleties which we elide here. These include dynamically nesting blocks, interruption while waiting, the interaction between class-loading and atomic block execution, thread creation within atomic blocks and the use of condition variables within atomic blocks. These issues are ones which would need to be considered carefully if incorporating `atomic` blocks into the design of a new language.

1.2 Implementation overview

Although we define the semantics of `atomic` blocks in terms of single-threaded execution we do not envisage that that would form the basis of an implementation. Instead, our current implementation is designed to allow most non-conflicting atomic blocks to execute concurrently.

The system is built in two layers. The lower layer is a word-based software transactional memory (STM). This allows groups of memory accesses to be performed within transactions which commit atomically. The STM is implemented in C within the Java Virtual Machine and provides operations for starting a new transaction (`STMStart`), aborting the current transaction (`STMAbort`), committing the current transaction (`STMCommit`), for reading a word within the context of the current transaction (`STMRead`) and for updating a word within the context of the current transaction (`STMWrite`). There are two further operations to validate transactions and to block threads while waiting for conditions to become true – these are not relevant to the current paper.

```
boolean done = false;
while (!done) {
  STMStart ();
  try {
    statements;
    done = STMCommit ();
  } catch (Throwable t) {
    done = STMCommit ();
    if (done) {
      throw t;
    }
  }
}
```

Figure 2: Code of the form `atomic { statements; }` expressed using STM management operations. In practice exception propagation is complicated by the fact that the translated code must throw the same set of exceptions as the original `statements`. Heap accesses within the `statements` (and within any methods they call) are performed using the STM.

The higher layer of the implementation maps the `atomic` keyword onto a series of STM operations. For example, entering an atomic block requires `STMStart` to be invoked, and accesses to shared fields within a block require that `STMRead` and `STMWrite` be used in place of direct heap accesses. This translation is implemented in the source-to-bytecode compiler (for transaction management operations) and the bytecode-to-native compiler (for individual field accesses). The intermediate Java bytecode format is unchanged.

Our previous paper describes these two levels in detail [6]. As an example, Figure 2 summarises how a basic non-nesting `atomic` block without a guard condition may be expressed in terms of these explicit transaction management operations.

2. MANAGING EXCEPTIONS

The semantics defined in Section 1.1 mean that if an atomic block terminates with an exception, then any heap updates made within the block are retained and the exception is propagated. This allows single-threaded code to be directly re-used in a multi-threaded environment by inserting atomic blocks around related accesses to the heap. However, there are examples where it would seem more convenient for programmers to be able to *roll-back* any updates made within the `atomic` block up to the point where the exception is thrown.

For illustration, consider code to move an object between two collections. The source collection provides a `remove` method. The destination collection provides an `add` method that fails with an exception if the target collection cannot hold the item supplied.

Figure 3 shows how a move operation can be implemented using an `atomic` block. The code is not elegant; the programmer must manually implement fix-up operations if the destination cannot contain the item supplied. Furthermore, when `R1` has to be counteracted by `A2`, the underlying transaction may involve numerous updates even though the abstract state of the two collections is unchanged. This is a problem in concurrent systems because it increases contention in the memory hierarchy. It may even be necessary

```
boolean move(Collection s, Collection d, Object o)
{
  atomic {
    if (!s.remove(o)) { /* R1 */
      return false; /* Could not find object */
    } else {
      try {
        d.add(o); /* A1 */
      } catch (RuntimeException e) {
        s.add(o); /* A2 */
        throw e;  /* Move failed */
      }
      return true; /* Move succeeded */
    }
  }
}
```

Figure 3: A collection-to-collection move implemented within an atomic block using manual roll-back. The `add` operation `A2` compensates for `R1` if the object is removed from the source collection but cannot be inserted into the destination.

```
boolean move(Collection s, Collection d, Object o)
{
  try {
    atomic {
      try {
        if (!s.remove(o)) { /* R1 */
          return false; /* Could not find object */
        } else {
          d.add(o); /* A1 */
          return true; /* Move succeeded */
        }
      } catch (RuntimeException e) {
        throw new AtomicAbortException();
      }
    }
  } (catch AtomicAbortException e) {
    return false; /* Move failed */
  }
}
```

Figure 4: A collection-to-collection move using an `AtomicAbortException` for roll-back.

to consider exceptions raised by `A2` if the object is rejected by both collections.

Of course, these same observations would hold if the `move` method was implemented using mutual exclusion locks. However, building the system over a STM allows the more convenient option of replacing the compensating operation `A2` with a request that the STM simply discards any heap updates performed within the `atomic` block.

2.1 Problems

Although the underlying STM provides an abort operation, this cannot be used directly to roll-back an `atomic` block before propagating the exception which caused the block to be aborted. The problem is that aborting would undo *all* of the updates made in the transaction: if the exception object was instantiated or modified in it then retaining that object is incompatible with rolling back the modifications. In the general case the exception object could be interlinked with other data structures, making it unclear which modifications to retain and which to lose.

There are two, more subtle problems with blindly using exceptions to trigger roll-back. The first is that it could destroy invariants assumed by existing code. For example, a library may ensure that a particular kind of exception is only thrown once a data structure has reached a given state. This guarantee would be broken if changes leading up to the exception were rolled back but the exception object was retained.

The second problem is that if all exceptions trigger roll-back then it precludes alternative implementations of `atomic` blocks which, unlike our STM, do not produce the logging information necessary to abort a transaction – this might be true of a scheme based on locking rather than an STM, or a scheme which includes optimizations for single-threaded use.

2.2 Design

Our approach is to introduce a new `AtomicAbortException` class and to have instances of that, or its subclasses, trigger roll-back. This is a checked exception class and so the programmer must indicate where it may be thrown, allowing a non-abortable implementation to be used for blocks where these exceptions are not present.

Figure 4 shows how an atomic collection-to-collection move could be implemented using roll-back: it is no longer necessary to include explicit compensatory code, and failed moves will lead to aborted lower-level transactions, reducing contention.

We use object serialization to define what happens when aborting a block while retaining the exception object which triggered the abort. This is because the serialized byte-array form of an object is meaningful between JVMs and therefore meaningful between an `atomic` block and its enclosing context. If a block terminates by throwing an exception `e` whose serialized representation would be a byte-array `b` then the effect of executing the block is equivalent to de-serializing a byte-array with the same contents as `b` and then throwing the resulting exception. Of course, this 'as if' definition allows the exception object to be retained and thrown directly if it is possible to identify that as equivalent through static analysis.

3. MANAGING I/O OPERATIONS

The second area which we consider in this paper is how to support `atomic` blocks with external side effects. In our original design we prohibited blocks from invoking any `native` method – that is, any method that is not implemented in Java bytecode. This ultimately precludes the availability of most I/O operations.

3.1 Problems

It is not possible to allow native methods to be called from `atomic` blocks by simply ensuring that JNI heap accesses are performed using the STM. That would provide no control over system calls invoked from native methods, or on code within the JVM which uses internal lower-level interfaces to bypass JNI.

Of course, there are some operations for which the JVM cannot guarantee atomicity. For example, the programmer may define an `atomic` block to swap the names of two files by a series of `renameTo` method calls. Operating system support would be needed to make these operations appear atomic to other processes; all that can reasonably be provided is atomicity in the sense that either all of the operations in the block appear to occur, or none of them occurs. Again, this is consistent with our intended 'as-if single threaded' semantics from Section 1.1.

Furthermore, different behaviour is appropriate for different kinds of I/O operation. For instance, a highly stylized server implementation may be written as a loop:

```
void serverLoop(ServerSocket s) {
  while (true) {
    Socket c = s.acceptConnection(); /*M1*/
    Thread t = new Thread() {
      public void run() {
        atomic {
          try {
            dealWithClient(c); /*M2*/
          } catch (Throwable t) {
            throw new AtomicAbortException(t);
          }
        }
      }
    };
    t.start();
  }
}
```

Connections from clients are received at method call `M1` and each is dealt with in an `atomic` block in a separate thread at `M2`. If an exception occurs in `M2` then the effect of the atomic block is discarded. In this case it may be appropriate for the external interactions performed between the client and the server to be carried out directly while executing the block and for the roll-back to only discard updates to the state within the server: the exception may indicate an internal error in the server or one that has been triggered by a maliciously formed request from a client.

In other cases it might be appropriate for external interactions to be deferred until the block has completed, or for corresponding compensatory operations to be issued if it does roll back.

3.2 Design

Rather than directly supporting unmodified native methods, the approach we take is to provide a set of Java-based interfaces with which an I/O library can implement appropriate buffering semantics. These allow a thread to determine whether it is in an atomic block and to register call-backs for when the transaction underlying the block attempts to commit or abort.

This allows a wide range of behaviour to be implemented. For instance, an output library can perform its own buffering of the deferred output, register a callback on commit to flush the output and register a callback on abort to discard the buffered state. Similarly, a library performing input can register a callback on abort to re-buffer the input which had been presented to the aborted transaction. This approach allows device-specific forms of buffering to be used – for example, to distinguish between stream-based input which cannot be re-ordered and datagram-based input in which datagrams may be re-ordered.

For console I/O we have implemented simple wrapper classes `AtomicInputStream` and `AtomicOutputStream` which provide example buffering layers for use above the ordinary I/O streams. Figure 5 shows an example of how an `AtomicOutputStream` can be used. If these I/O features were integrated fully into the environment then these wrappers could be provided as the default I/O streams.

```
public class ExampleOutput {
  static PrintStream out =
     new PrintStream(
        new AtomicOutputStream(System.out));

  static void print_sum(int x, int y) {
    atomic {
      int result = x + y;
      out.println ("Result is " + result);
    }
  }
}
```

Figure 5: An example class instantiating and using an `AtomicOutputStream` wrapper to buffer output made within the `atomic` block in the `print_sum` method.

4. EXTERNAL ACTIONS

In this section we introduce the 'external action' abstraction with which we implement our exception propagation model and I/O support libraries. In Sections 4.1 and 4.2 we discuss two ways of exposing external actions to programmers; we have implemented the first of these options and, although we have a thorough design for the second option, we have not yet tested it in practice.

External actions provide a controlled way in which code within an `atomic` block can temporarily perform operations directly on the heap rather than within the context of the current transaction. External actions are used in propagating exceptions in order to marshall the exception object so that it is available after the transaction is aborted. External actions are used during I/O to invoke native operations and to perform device-specific buffering to give transactional behaviour.

The behaviour of external actions are defined in terms of *contexts* which represent the different views that threads may have on the heap at any given moment. Contexts are hierarchical and a single *global context* exists as the root. Heap updates are said to occur *within* a given context, meaning that they are guaranteed to be visible to threads executing in that context, or executing within any context nested inside it.

When a thread enters an atomic block it creates a new context nested within its current one. When a thread leaves an atomic block then the nested context is discarded after *promoting* any heap updates made within it up to its parent context. Figure 6 illustrates a set of nested contexts.

The key challenge in Java in designing a mechanism for temporarily 'stepping outside' the current context is making it impossible to circumvent encapsulation enforced by language-based protection. In particular, code executing in

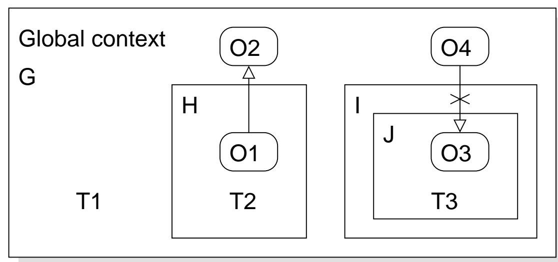


Figure 6: Thread T1 is executing in the global context G. Thread T2 is executing in context H within G. Thread T3 is executing within context J, nested two levels deep. Objects allocated in one context can only contain references to objects allocated in enclosing contexts, for instance O1 can refer to O2, but O4 cannot refer to O3.

a given context must not be able to access objects instantiated in an enclosed context – otherwise, for example, there is no guarantee that the code would even see the objects correctly initialised.

We deal with this problem by representing external actions as designated `ExternalAction` objects and ensuring that (*i*) actions are executed in the context within which the object is instantiated, and (*ii*) actions' parameters are passed by serialization. The first property ensures that free variables occurring within an action's definition will refer to data that is accessible in the context within which the action executes. The second property ensures that any incoming parameters received by the action have been copied and re-created within the context that the action executes.

We expose contexts to Java programmers as immutable `Context` objects which uniquely identify an active context and allow traversal from it to its enclosing context object. A static method returns the caller's current context. A thread can register a `ContextListener` with any context that is contained within its current one. Context listeners receive three call-backs:

```
boolean validToCommit(Context c);
void actionOnCommit(Context c);
void actionOnAbort(Context c);
```

These three operations are used to perform a two-phase commit of updates that external actions have associated with a context. The first of these, `validToCommit`, is called when deciding whether the context should be destroyed or whether, at the end of an `atomic` block, updates made within it should be merged into its parent context. If any context listener returns `false` then the context must be destroyed. The second and third call-backs are called to to inform the listener of the outcome of this voting.

External actions are implemented by extending the STM interface with two context-control operations: a method for setting the current transactional context used by STM operations and a method for doing an inter-context copy of arrays of bytes when serializing parameters to external actions. The remainder of the implementation is Java-based; the `STMCommit` operation becomes a Java method which calls `validToCommit` on any `ContextListener` objects before attempting to commit the underlying STM transaction.

The two context-control operations are available only to

```
public class ExampleActionCall {
  static int x = 0;

  static VoidExternalAction printX =
    new VoidExternalAction() {
      public void action(Context caller_context) {
        System.out.println(" x=" + x);
    }};

  static void increment_x() {
    atomic {
      printX.doAction();
    }
  }
}
```

Figure 7: An example code fragment defining and invoking an external action.

trusted code. However, we have investigated two ways of exposing them safely to ordinary code such as applications and I/O library implementations. The first of these, which we describe in Section 4.1, allows a single operation to be defined at a time. The second design, in Section 4.2, exports a whole `interface` of external actions: it is more verbose for short examples but is more convenient for non-trivial cases.

4.1 Operation-based external actions

The first way of defining external actions uses a simple mechanism in which the action is defined by overriding an `action` method on an `ExternalAction` class. A separate trusted `doAction` method uses the context-control extensions to marshal parameters for the action and to invoke it in the appropriate context.

Figure 7 illustrates this: the `VoidExternalAction` class is extended with an `action` method that is called from the context created in `increment_x` but which is executed in the global context that was active when `printX` was initialized.

Generic types and variable-length argument lists can simplify the infrastructure for defining this form of external actions by avoiding the proliferation of separate kinds of action for different parameter and return types. Aside from actions with `void` return type, a single parametric definition would suffice.

However, with this approach, defining external actions which can throw checked exceptions remains problematic: the definition cannot be made parametric on a *set* of exceptions. In general the programmer has to follow inelegant approaches such as hiding checked exceptions within unchecked wrappers.

4.2 Interface-based external actions

The second way of defining external actions is more suitable for use in larger settings where the entire set of existing methods on an object are to be encapsulated as external actions. The approach is to allow an object to be exported from one context and for *all* method invocations on it to be made via stubs which behave as external actions.

The need for this kind of interface-based design became particularly apparent while creating wrappers for use around the Java Transaction API in which large numbers of boilerplate actions otherwise had to be written to wrap exist-

```
// Definition of interface exported
interface printXIfc {
  public void printX();
}

// Signature of export operation
public class ExternalAction {
  static <F> F export(F imp) {
    ...
  }
}

// Invocation of external action
public class ExampleActionCall {
  static int x = 0;

  static printXIfc printer =
    ExternalAction.export(
      new printXIfc() {
        public void printX() {
          System.out.println(" x=" + x);
        }
      });

  static void increment_x() {
    atomic {
      printer.printX();
    }
  }
}
```

Figure 8: An external action defined using an interface.

ing implementations of interfaces such as `UserTransaction`, `PreparedStatement` and `Connection`.

Figure 8 illustrates how the earlier `increment_x` example from Figure 7 could be expressed in this alternative form. As before, the example ultimately prints the contents of a field `x` in the global context. This operation is performed by (*i*) providing an interface `printXIfc` which defines the signatures of the methods to be exported as external actions, (*ii*) defining an implementation of these operations to be exported, (*iii*) invoking `ExternalAction.export()` to produce a set of stubs to perform the inter-context calls.

The stubs are constrained to implement an identical interface to one implemented by the original, retaining `throws` clauses for checked exceptions as well as the details of return types and parameters.

5. IMPLEMENTATION EXPERIENCE

In this section we consider the use of external actions in providing a mechanism for managing exceptions (Section 5.1) and for performing external I/O operations (Section 5.2).

5.1 Propagating exceptions

The exception-propagation mechanism proposed in Section 2 can be implemented by a single external action that takes the exception object created within the `atomic` block and returns a deep copy of it created in the global context. The definition of this action is simply:

```
static ObjectExternalAction promoteException =
  new ObjectExternalAction() {
    public Object action
              (Context caller_context,
               Serializable aae) {
      return aae;
    }
  };
```

The actual copying of the exception object to the global context is performed by the marshalling of the exception object when `promoteException` is invoked. The design in Figure 2 for implementing an `atomic` block using STM operations is extended to propagate exceptions by adding an exception handler of type `AtomicAbortException` and having this promote the exception, abort the transaction and then re-throw the copy the exception.

5.2 Performing I/O

I/O operations are implemented using external actions to perform any native method invocations necessary for the I/O and using `ContextListener` call-backs to trigger re-buffering of unused input (when aborting an input operation) or to trigger the actual output of buffered data (when committing an output operation).

For example, when reading from standard input, an external action is used to perform the read. It calls a native read method from within the global context and buffers the value read, again within the global context. In this case a context listener is registered to re-buffer the data if the `atomic` block is aborted, or to discard the buffer if the `atomic` block completes successfully.

We define a set of utility classes which simplify the implementation of abstractions such as the `AtomicOutputStream` in Figure 5. These hold ordered collections of objects that are buffered until an atomic block commits, and collections of input items that have been received by an `atomic` block and must be held for potential re-buffering in case the block aborts.

Integration with external database transactions is not so straightforward. We have built a prototype system based on the Java Open Transaction Manager (JOTM)[1], although this relies on modifications to the JOTM implementation rather than being made through the established Java Transaction API (JTA) [11]. The fundamental problem is that both the STM and the JOTM system want to make the final decision of whether or not to commit a set of operations; neither allows the other to perform a separate 'prepare' phase. We chose to extend the JTA `UserTransaction` interface with an additional `prepare()` operation. This issue would have to be addressed more methodically in a full-strength implementation of our system.

6. RELATED WORK

This `atomic` construct builds on designs for Conditional Critical Regions [7] and on the concurrency control features of languages such as DP [2], Edison [3], Lynx [10] and Argus [8].

Stack-like memory usage disciplines have been investigated in several other settings, most notably region-based memory management [12]. Regions have been proposed as

[1] `http://jotm.objectweb.org`

an alternative or adjunct to traditional garbage collection, allowing objects to be allocated within a stack of regions and allowing space to be reclaimed by removing an entire region from the top of the stack. Safety requires that references do not occur from more permanent regions into less permanent ones.

The Real-Time Specification for Java (RTSJ) [1] defines a way of allocating objects within 'scoped memory areas' in order to allow storage reclamation without a run-time garbage collector. Scoped memory areas must obey similar constraints to the `Context` objects proposed here: objects within one area may not refer to objects in less permanent areas.

There are three main areas in which differences exist between our scheme, regions and scoped memory areas. The first is in whether the prevention of illegal references is done statically or dynamically: our system, as with conventional region-based ones, takes the former approach whereas RTSJ takes the latter. The second point of comparison is the direction in which contexts are entered: our system must support transitions both from an outer context to an enclosed one (by entering an atomic block) and from an enclosed context to an outer one (by invoking an external action). The final point is that the stack of `Context` objects in our system should be viewed as 'overlays' on the same heap, with objects at one layer being shadowed by objects at enclosed layers, whereas the identities of objects in different regions or scoped areas are considered distinct.

7. CONCLUSIONS AND FUTURE WORK

This paper has shown how we have extended our `atomic` regions for concurrent Java programs to suppport explicit abort operations and I/O. The design presented here introduces a notion of nested execution contexts and an abstraction for performing inter-context method calls. In this final section we highlight a number of dead-ends we followed in earlier designs (Section 7.1) and a number of extensions for future work (Section 7.2).

7.1 Early dead-ends

Although these final abstractions are individually simple, developing them has highlighted a number of problems which we had not originally foreseen. These all relate to the need to be careful about passing object references into a context in which the initialisation of the objects' fields will not have been visible.

The original design we sketched proposed control methods through which reads or writes could be performed outside the current software transaction [5].

This approach is not safe with respect to the language-based protection provided by Java: for example, `final` fields are intended to be constant once initialised, but using these methods a programmer could cause the initialisation to happen within a transactional context and subsequent accesses to take place outside that context and therefore without the initializations visible.

In subsequent designs we considered introducing a form of 'global action' which would always execute in the global context. As with our ultimate design for external actions, these would be defined by instantiating an anonymous inner class, for example:

```
atomic {
  final String s = new String("Erroneous example");
  GlobalAction g = new GlobalAction() {
    public void doAction(Context caller_context) {
      System.out.println ("s=" + s); /*P1*/
  }};
  g.doAction();
}
```

Unfortunately if `P1` is exectued in the global context then the initialization of the object `s` refers is not visible. Note how our decision to execute external actions within the context within which they are instantiated avoids this problem without the need for dynamic checks. It also deals naturally with the case of nested contexts.

7.2 Future work

Object finalizers still pose a problem: if an object is instantiated in an `atomic` block and that block is subsequently rolled back by an exception then should finalizer methods be invoked on the objects that are lost? There appear to be two options: the first is to consider the destruction of the atomic block's context to entirely undo the creation of the objects and therefore to not run finalizers on them. The second option is to execute the finalizers within the context that the objects were instantiated – i.e. to execute them just before destroying the context. These two options have different behaviour if the finalizers loop or perform external actions. We favour the first option because it is simpler to implement and because it is consistent with the semantics of Section 1.1.

The key direction for future work is evaluating the practical utility of the techniques that we have developed: we have now considered atomic blocks with an armoury of features, but we have not exercised these features in earnest in a large system. It will also be instructive to see whether the roll-back mechanisms triggered by `AtomicAbortException` objects can simplify sequential programs by automating the management of compensatory actions – this may be particularly useful when developing I/O-processing code with a wide variety of possible failure points.

A further point for future investigation will be the relationship between this work and the `java.util.concurrent` library[2] anticipated in J2SE 1.5. For instance, once there are benchmarks targeting JSR-166 features, then it will be interesting to compare the implementation of collections and queues built using `atomic` blocks with those built using the virtual machine's existing abstractions. We hope that our work is an excellent counterpart to JSR-166 and that the combination of well-engineered high-level abstractions and an effective mechanism for extending them to provide aggregate atomic operations may encourage more wide-scale adoption of concurrency in applications.

7.3 Acknowledgments

This work has been supported by a donation from the Scalable Synchronization Research Group at Sun Labs Massachusetts.

[2] JSR-166, `http://www.jcp.org/en/jsr/detail?id=166`

8. REFERENCES

[1] Bollella, G., Brosgol, B., Dibble, P., Furr, S., Gosling, J., Hardin, D., Turnbull, M., and Belliardi, R. *The Real-Time Specification for Java*. Addison Wesley, June 2000.

[2] Brinch Hansen, P. Distributed processes: A concurrent programming concept. *Communications of the ACM 21*, 11 (Nov. 1978), 934–941.

[3] Brinch Hansen, P. Edison – a multiprocessor language. *Software – Practice and Experience 11*, 4 (Apr. 1981), 325–361.

[4] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction to Algorithms*. MIT Press, Cambridge, Mass., 1990.

[5] Harris, T. Design choices for language-based transactions. Tech. Rep. UCAM-CL-TR-572, University of Cambridge, Computer Laboratory, Aug. 2003.

[6] Harris, T., and Fraser, K. Language support for lightweight transactions. In *Object-Oriented Programming, Systems, Langauges & Applications (OOPSLA '03)* (Oct. 2003), pp. 388–402.

[7] Hoare, C. A. R. Towards a theory of parallel programming. In *Operating Systems Techniques* (1972), vol. 9 of *A.P.I.C. Studies in Data Processing*, pp. 61–71.

[8] Liskov, B., and Scheifler, R. Guardians and actions: linguistic aupport for robust, distributed programs. *ACM Transactions on Programming Languages and Systems 5*, 3 (July 1983), 381–404.

[9] *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages* (Jan. 1994), ACM SIGPLAN Notices, ACM Press.

[10] Scott, M. L. Language support for loosely coupled distributed programs. *IEEE Transactions on Software Engineering SE-13*, 1 (Jan. 1987), 88–103.

[11] Singh, I., Stearns, B., and Johnson, M. *Designing enterprise applications with the J2EE platform*, 2nd ed. Addison Wesley, 2002.

[12] Tofte, M., and Talpin, J.-P. Region-based memory management. *Information and Computation* (Feb. 1997). An earlier version of this was presented at [9].

Transactional Lock-Free Objects for Real-time Java

F. Pizlo M. Prochazka S. Jagannathan J. Vitek
Purdue University

ABSTRACT

Priority inversion is an important concern in providing robust synchronization in real-time systems. When a high-priority task attempts to acquire a lock held by a low priority task, it is often necessary to momentarily resume the execution of the low priority task so as to allow it to leave the critical region safely, ensuring that shared resources are not in an inconsistent state. Once these resources are properly released, the high priority task can proceed. In pathological cases, the priority of several threads may have to be increased, and the high priority tasks can experience unbounded delays.

An alternative approach would record the original values of shared objects whenever they are modified, restoring them if the executing thread is interrupted by a higher-priority one. This approach thus treats the critical section as a lightweight *transaction*. This paper presents an extension to the Real-time Specification for Java with *transactional lock-free* (TLF) objects. Atomic methods of TLF-objects can be accessed concurrently without risking priority inversion. The semantics of our transactions are such that a high-priority thread will always succeed when trying to enter an atomic section. The time to enter is bounded by the number of locations updated within the atomic section. Experimental results undertaken in the context of Ovm, a virtual machine framework for Java that implements the Real-Time Specification for Java, indicates that transactional lock-free objects can improve the responsiveness of high priority threads compared to priority-inheritance based approaches at the cost of a reduction throughput.

1. INTRODUCTION

The Real-Time Specification for Java (RTSJ) [6] holds promise to play a key role in the construction of real-time systems in a type-safe, high-level, programming language. The RTSJ is being evaluated for use in mission-critical systems by the likes of Boeing [24] and JPL [18], and has one high-quality commercial implementation [14], as well as a number of open source incarnations [4, 8] and variants [19, 7, 30, 9, 27]. This paper focuses on the RTSJ programming model, and in particular, on how to write correct and efficient real-time code in the presence of priority preemptive threads and mutual exclusion.

Priority inversion is a well-studied problem in concurrent real-time programming. Avoiding priority inversion is especially important in mission critical or real-time applications [17, 10, 22, 23]. Priority inheritance and priority ceiling emulation are two well-known protocols that attempt to avoid priority inversion. The priority ceiling emulation (PCE) technique raises the priority of any locking thread to the highest priority of any thread that ever uses that lock (i.e., its priority ceiling). This requires the programmer to supply the priority ceiling for each lock. In contrast, the priority inheritance protocol (PIP) will raise the priority of a thread only when holding a lock causes it to block a higher priority thread. When this happens, the low priority thread inherits the priority of the higher priority thread it is blocking. Yet another alternative is to have privileged threads, for example those executing on behalf of the operating system. These threads can often disable interrupts or preemption that effectively locks lower-priority threads from acquiring critical resources. Regardless of the approach, once a thread enters a synchronized section its locks cannot be summarily relinquished without potentially violating synchronization invariants.

Even with a priority inversion avoidance protocol a high-level thread can experience an unbounded delay if a low priority thread's does not relinquish its lock, or, in less extreme cases, if it takes longer than planned to complete its work. While this situation may be described as a programming error, it is nevertheless a case that can occur, especially in large and complex embedded software such as those found in avionics systems.

The RTSJ supports priority preemptive threads with a minimum of 28 unique priorities. Furthermore, it provides, by default, an implementation of a priority inheritance algorithm and supports priority ceiling emulation. Priority inheritance may be transitive since a low-priority thread may try to acquire resources held by an even lower priority threads. In the RTSJ, application programmers are also faced with the additional problem that some low-priority whose priority is boosted may trigger a garbage collection, effectively blocking the high-priority thread from executing for the entire duration of the collection.

We propose to investigate an alternative concurrency control mechanism based on *transactional lock-free* (TLF) objects. In this scheme, concurrency control is implemented by critical sections which provide atomicity via a simple compiler-assisted transactional mechanism. Whenever a high priority thread tries to enter an atomic method of a TLF-object, the operation always succeeds in time bounded by the number of objects updated in the method. Any lower priority thread

executing within an atomic method will be evicted, and its changes undone. The lower priority thread is guaranteed to be reexecuted when the high priority thread completes its work. Since the overheads to perform rollbacks are charged only to low-priority threads, our scheme favors responsiveness over throughput.

This paper defines the semantics of TLF-objects and describes their implementation within a real-time Java virtual machine. The salient features of our proposal are as follows:

- *A simple programming model* in which priority inversion can not occur and higher priority threads are guaranteed to enter critical sections within a bounded number of steps.
- *Integration with the RTSJ* in a backwards compatible way. No changes to the Java language are required, we introduce annotations that are interpreted by the VM. Our proposal can coexist with traditional synchronization.
- *Efficient implementation* which required modest changes to an existing RTSJ virtual machine and its optimizing compiler.

Transactional extensions to programming languages have received renewed attention of late. This work is closely related to that of Harris [11] and Welc *et.al.* [32]. The main difference between our work and theirs, besides choice of implementation technique, is that we must ensure that space for a transaction is bounded.

The paper is organized as follows. We begin with an overview of relevant features of the Real-time Specification for Java. Section 3 introduces transactional lock-free objects. Section 4 describes the implementation of TLF-objects within Ovm. Section 5 reports on experimental results.

2. REAL-TIME SPECIFICATION FOR JAVA

We overview the salient features of the Real-time Specification for Java (RTSJ). The RTSJ extends Java with support for real-time programming in a backward compatible way. The RTSJ requires no changes to syntax of Java and allow real-time and plain Java codes to co-exist in one virtual machine. One design choice made in the RTSJ is to extend the Java programming model with two abstractions: (a) regions of memory, called scoped memory areas, which are not subject to garbage collection, and (b) real-time threads that never interact with the heap and thus can never interfere with, or be affected by, the garbage collector. Technically real-time threads come in two flavors, `RealtimeThread` and `NoHeapRealtimeThread`, and only the latter is guaranteed not to experience garbage collection pauses. Both kinds of threads can be created with any of the 28 priorities over and above the ten priorities defined in standard Java and be given scheduling parameters that are either periodic, aperiodic, or sporadic.

Priority inversion is avoided by defining *monitor control policies*. The RTSJ overloads the meaning of the `synchronized` keyword by allowing programmers to specify a monitor control policy for each Java language monitor. Two policy classes are provided by default: `PriorityInheritance` and `PriorityCeilingEmulation` that implement the familiar notions of priority inheritance and priority ceiling respectively. The static method `MonitorControl.setMonitorControl(p)` can be used to set the default policy for the entire virtual machine, which `setMonitorControl(target, policy)` sets the policy for a single object, `target`.

3. TRANSACTIONAL LOCK-FREE OBJECTS

An alternative approach to the above mentioned priority inversion avoidance schemes is to use lightweight transactions instead of monitors to control access to critical sections. For example, one can apply optimistic concurrency semantics [31] to any number of concurrently executing tasks, allowing these tasks to simultaneously enter the same critical section (*i.e.* a sequence of operations protected by the same lock). If the operations performed by these tasks do not conflict they will be allowed to *commit* their changes, making their updates visible to other threads in the program; otherwise, one or more task will be aborted, and their changes discarded. An aborted task may retry execution.

Our approach can be viewed as a hybrid transactional model that combines features of both pessimistic and optimistic concurrency semantics in a manner suited to real-time systems. As in a pessimistic concurrency control model, only one thread is allowed to execute within a TLF-object at any given point. As in an optimistic concurrency control model, a thread can be aborted while executing within a TLF-object, if a higher-priority thread requests access to the same object. As in these other approaches, state information is logged by each thread to ensure that when an abort does occur, updates it has performed can be reverted to ensure that consistency invariants are not violated.

We propose a simple language extension inspired by the work of Harris and Fraser [11] and Welc *et.al.* [32] geared towards priority preemptive real-time systems. We introduce a single new keyword `@atomic` used to declare transactional critical sections:

```
@atomic void update( int i) {
   if ( i > this.field )
      this.field = i;
}
```

All writes performed within the method, and the methods it invokes, are guaranteed to become visible when the current thread leaves the critical section. In the case a transaction is aborted all changes performed by the thread within the synchronized region are undone and the method is re-excuted.

We refer to objects with **@atomic** methods as *transactional lock-free* (TLF) objects[1]. A TLF-object acts as a monitor with respect to its atomic methods. There can be only one thread concurrently executing an atomic method of a given

[1]Anderson *et.al.* have proposed lock-free shared objects in [2]. The main difference between our works is that we provide a language-based solution, so none the operations performed by an aborted thread can be witnessed by other threads. This is not the case in Anderson's work where only the transactional object is protected, shared variables may be modified by an aborted thread and these will not be undone.

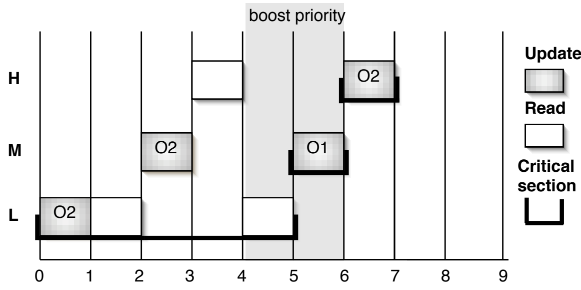


Fig. 1: A sequence of actions performed by different priority threads under priority inheritance trying to enter critical sections guarded by the same lock. Low priority thread L is released at time step 0, mid priority thread M is released at time 2, and high priority thread is released at time step 3. L and M operate with boosted priorities at time steps 5 and 6, respectively. The execution of H's critical section is delayed until time step 6.

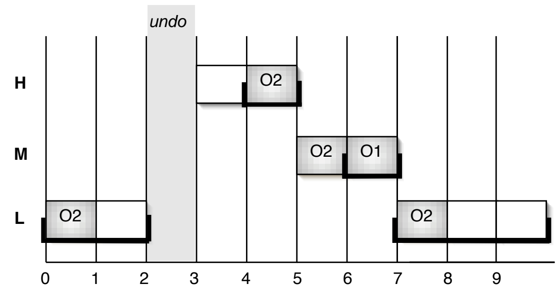


Fig. 2: A sequence of actions performed by different priority threads sharing under transactional semantics. Threads M is released at time step 2, causing thread L to abort and undo its write. Thread H is released at time step 3, preempting M. Thread L is reexecuted at time step 7. Notice that in this example transaction reduce overall throughput but increase responsiveness of the high priority thread.

TLF-object. We ensure that a thread is allowed to enter a critical section, only if the TLF-object protecting it is not currently being accessed, or if the TLF object is currently owned by a lower priority thread. In the latter case the lower priority thread is evicted from the critical section, its changes are undone, and it is permitted to re-execute whenever the high priority thread completes its work.

Fig. 2 illustrates the actions undertaken by different threads when they share access to a critical section defined as a transactional object. At time step 0, a low-priority thread transactionally executes regions of code that updates object `O2` (which is not necessarily the transactional object but can be any object visible to that thread). Before it completes execution of its transaction, the thread is interrupted by a medium-priority thread causing the modification to `O2` to be undone (time step 2). Before the medium priority thread is allowed to execute its transaction, it is interrupted itself by a high priority thread which transactionally updates `O2`. Once the high priority completes its work the medium priority thread is allowed to proceed (time step 5). Finally the low-priority thread regains control and completes its work (time step 7 to 9). Note that the low-priority thread must reexecute its actions Thus, when a lower-priority thread is evicted, it must resume execution from the beginning of the critical section; no intermediate results from its previous aborted execution are preserved.

In contrast, Fig. 1 shows a comparable scenario using a priority-inversion avoidance protocol. In this case, the interruption of the low-priority thread `L` by medium-priority thread `M` at time step 2 is allowed because M does not immediately enter the critical section. Thread H preempts M at time step 3, in order to get to execute The priority of `L` is boosted at time step 4 and `M` at time step 5. With a protocol such as priority inheritance, the maximum time that `H` must wait before it can begin execution within the critical section is only bounded if the length of individual critical sections can be bounded. In contrast, TLF-objects bound the time that a higher-priority thread must wait to enter a critical section as a function of the number of updates that must be undone. Since undoing the effects of an update involves simply restoring the original value of the object at the time the transaction was entered, this cost is significantly easier to calculate and predict than the worst case execution time (WCET) of critical sections[2]. The cost of transaction is that overall throughput is reduced due to the cost of undoing and reexecuting threads.

3.1 Example

Consider the doubly linked list data type, `DLList`, defined in Fig. 3. The class has two atomic methods, `insert()` which performs ordered insertion and `getMax()` which returns the list's largest value. An auxiliary method, `splice()`, inserts a new cell in a list. A program with two threads, a high priority thread **H** and a low priority **L**, which interact via a shared `DLList` object is given next (the actions performed by each thread are shown, the indentation hints at the relative start times):

```
L:   shared.insert( obj)
H:       shared.getMax()
```

Assume `L` is scheduled first, it acquires the shared TLF-object, enters the critical section and start executing the insertion code. If `H` is scheduled before `L` completes, the VM will notice that `shared` has been acquired by another thread. By definition of a priority preemptive system, the priority of `L` *must* be less than the `H`. This leaves no ambiguity as to what must happen next. `L` is aborted; if it has already performed some of its updates, they are undone and `H` is permitted to enter the critical section. Once `H` completes its work, `L` will be rescheduled and will re-execute the call to `insert()`.

In this example the cost aborting a call of `insert()` is bounded

[2]Practitioners still estimate WCETs by empirical means, running the program on different inputs and measuring execution time.

```
class DLList {
   Comparable value;
   DLList prev, next;

   @atomic DLList insert( Comparable v) {
      DLList cur = this, pre = null;
      while ( cur != null && v.gte( cur.value) )
         { pre = cur; cur = cur.next; }
      if ( pre == null )
         return new DDList( value).splice( this);
      else {
         pre.splice( new DDList( value));
         return this;
      }
   }

   DLList splice( DLList after) {
      DLList tmp = this.next;
      this.next = after;
      after.next = tmp;
      after.prev = this;
      return this;
   }

   @atomic Comparable getMax() {
      Comparable max = this.value;
      DLList cur = this;
      while ( cur != null ) {
         if ( max.lte( cur.value) ) max = cur.value;
         cur = cur.next;
      }
      return max;
   }
}
```

Fig. 3: TLF-object example. insert() performs an ordered insertion in the list. Calls to insert start a new transaction which protects the internal call to splice() ensuring that all updates are logged. If the transaction completes changes become visible, otherwise they are undone. The latency of calls to insert() are bounded by the cost of restoring the three cells updated within splice().

by the cost of undoing writes to three `DLList` objects, while aborting `getMax()` incurs no extra cost.

The semantics of `@atomic` methods are such that the original values of all objects updated within the method, or methods called from it, are logged including objects of other types. Thus the call to `splice()` is also protected by the transaction initiated in `insert()`. Just as with Java synchronization, calling an unprotected method (such as `splice()`) directly from another thread may reveal inconsistent state. For instance, consider the following program:

```
L:   shared.insert( obj)
M:      shared.splice( cell)
H:         shared.insert( obj)
```

If a medium priority thread is released while `L` is within its critical section, it may observes values modified by `L`. If `H` is called before `L` completes the transaction may be aborted and the values observed by `M` become stale.

3.2 Semantics

The semantics of TLF have been designed to suit the requirements of RTSJ; in particular we have tried to balance expressiveness of the programming model with space and time efficiency concerns. TLF-objects are based on a flat non-nested transactional model. Whenever a real-time thread `T` invokes a method annotated with the `@atomic` keyword on an object `tobj`, an implementation is required to checked if the `tobj` is currently owned by another thread. If it is the case the other thread is aborted and the ownership field of `tobj` is cleared. Thread `T` then *acquires* ownership of `tobj` and begins executing. When `T` *exits* from the critical section, all changes it made are finalized and ownership of `tobj` is reset. A thread may exit a critical section *normally*, when the method returns, or *exceptionally*, by throwing an exception from the method. In both cases all changes made by the thread will be finalized.

The original values of all objects updated by a thread between the time a TLF-object is acquired and the critical section is exited are recorded in a log to ensure state consistency if the thread is interrupted prior to its exit of the section. An implementation is required to allocate space for logs at virtual machine start up before any real-time thread are started. To support both plain real-time threads and `NoHeapRealtimeThreads`, logs are allocated in immortal memory. The logging policy is implementation dependent – the only requirement is that space required by the log be bounded. Since we log the original value of an object the first time it is updated, each modified object must be logged at most once. It is permissible to log partial objects. Locations in an array may be logged at most once per write to the location.

An implementation is only required to ensure that objects are in a consistent state when a thread exits its critical section. Attempts to access objects being modified are unchecked programming errors as they may yield stale value of these objects.

Aborted transactions are automatically re-executed. The process is transparent to the application. A thread is neither able observe that a transaction was re-executed nor allowed to explicitly trigger an abort.

Code Constraints. An implementation is allowed to reject any `@atomic` method that may perform blocking operations. This can be validated by an off-line static analysis of real-time code. The analysis only needs to classify methods as, either, safe or unsafe. A method is *safe* if it does not contain synchronized statements. Moreover, safe methods cannot include calls to native methods since such methods can perform blocking operations. In practice, native methods are inspected manually, and declared safe on a case-by-case basis. For the same reason, safe methods cannot make reflective calls, or refer to classes that cannot be statically guaranteed to be available and initialized[3]. Finally, a method is safe if all methods called in its body can be proved safe.

[3]Class initialization is an issue for RTSJ programs. Java semantics mandate classes to be loaded and initialized lazily, but to obtain any measure of predictability a RTSJ-virtual machine will likely load all classes agressively and initialize them ahead of time. Reflection can potentially refer to classes that have not yet been loaded and must be used carefully.

Space Bounds. The upper bound of the log size is user-specified. As the layout of objects and the logging policy are implementation dependent, space requirements are estimated based on the maximum number of objects that can be modified in a critical section. For arrays the number used is that of writes to individual locations. The RTSJ provide a class for this purpose called `SizeEstimator`[4]; we extend this notion to bound transactional object memory requirements `TLFSizeEstimator`.

```
class TLFSizeEstimator extends SizeEstimator {
   TLFSizeEstimator( Method m);
   void reserve( Class c, int count);
   void reserveArray( Class c, int count);
}
```

Each atomic method has an associated estimator. The space required to log the changes performed in the method is declared by calling the `reserve()` and `reserveArray()` methods. `reserve(C, i)` sets aside space to log i instances of class `C`. `reserveArray(A, i)` sets aside space to log i writes to an array of type `A`. Once the estimator is complete, it must be registered with the VM using `registerAtomic()` method. The following code fragment register estimators for the methods of Fig. 3.

```
tins = new TLFSizeEstimator( dllist_insert_method);
tins.reserve( DLList.class, 3);
OVM.registerAtomic( tins);

tmax = new TLFSizeEstimator( dllist_getmax_method);
OVM.registerAtomic( tmax);
```

The `insert()` method may update three objects of class `DLList`, while the `getMax()` method performs no updates.

3.3 Discussion

The design of TLF-objects is fully backwards compatible with the RTSJ. Plain RTSJ code can execute on virtual machine supporting TLF-objects. Indeed, programs can use a mixture of TLF and traditional synchronization. The `@atomic` keyword does not require change to Java syntax; it is an annotation consistent with the JSR-175 Metadata extensions to Java. The keyword can also be replaced by a marker exception[5].

The transactional model adopted here could be relaxed at some cost in complexity and perfomance. For example, providing support for nested transactions would mean that aborting the execution of one atomic method may lead to a cascade of aborts to undo the effects of other atomic methods. Logging operations become more complex as a transaction must maintain one log per nested transaction to support nested aborts. This implies that the same object may appear in the log of several transactions. Furthermore, the cost of write barriers increases as it is necessary to check in which transaction's log the object has been stored.

A more modest change would entail recursive atomic sections, i.e. allowing a thread to call atomic methods on an object if it already has acquired that object. In the example of Fig. 3, this would allow method `splice()` to be declared atomic. The drawback of this change to the semantics is that some extra checks have to be performed when a thread enters and leaves a critical section, and that it is more difficult to check statically if a method is safe. Furthermore, there would be additional runtime exceptions if a transaction tried to acquire a different object.

Extending our semantics to support a full optimistic transactional model would increase throughput as more than one thread may execute in the same critical section, but would also increase the cost of exiting the region as it would be necessary to check that no conflict on shared state has occured, and would also increase the program's space requirements as multiple logs for the same object have to be maintained at the same time.

[4]In the RTSJ size estimators are used to set bounds on the size of memory regions.

[5]Marker exceptions are well known idiom, the declaration `@atomic void f() {...}` becomes `void f() throws Atomic {...}`.

4. OVM IMPLEMENTATION

TLF-objects have been implemented in the RTSJ configuration of the Ovm virtual machine framework. Ovm is an open source framework for building language runtimes. The framework contains more than 150K lines of code and over 2000 classes, including an interpreter, a just-in-time compiler and an ahead-of-time compiler. These components can be specialized and assembled into an *Ovm configuration* customized for a particular problem domain. The RTSJ configuration yields a virtual machine implementing the Real-time Specification for Java. This configuration compiles real-time Java code ahead of time and has a fast user-level threading system based on compiler inserted polling code. The platforms currently supported are x86/RTLinux and PPC/MacOSX. The ahead-of-time compiler produces code competitive with Sun's HotSpot virtual machine.

The main design decision taken in the Ovm implementation of TLF-objects is to limit number of concurrent transactions inorder to reduce space requirements and permit a more efficient implementation. Our implementation restricts applications to have *at most one* executing transaction at any given instant. We believe this restriction to be acceptable in practice as we are targeting system with relatively small numbers of threads and where transactional sections are short.

The layout of object, described by their *object header*, is modified with an extra bit that indicates whether the object has been logged by the current transaction.

The implementation of the basic transactional primitives is described in Fig. 4. When a thread attempts to enter an atomic section, it must check that no other transaction is executing. The VM maintains a reference to the current thread within a critical section in `VM.currentTransaction`. If there is an ongoing transaction executing on behalf of another thread, it is immediately aborted. Note that the thread executing within the section must have lower-priority in order for the thread performing the **acquire** operation to be executing. In any case, the transaction ownership field is set to the new thread and the acquisition succeeds. Exiting a critical section, requires clearing all the stamp fields of all logged objects as well as the `currentTransaction` field. Every write to an object is protected by a barrier. We extend the existing barrier code (which enforces scoped memory semantics) to log the object if a transaction is active and the

```
acquire ( thread)
   owner ← VM.currentTransaction
   if owner ≠ null
      abort( owner)
   VM.currentTransaction ← thread

exit ( )
   foreach o in VM.currentLog
      o.stamp ← false
   clearLog()
   VM.currentTransaction ← null

write ( o)
   if VM.currentTransaction ≠ null ∧ o.stamp = false
      log( o)
```

```
log ( o)
   log ← VM.currentLog
   if log.length + sizeof( object) + 2 ≤ log.size
      throw logOverflowError
   addToLog( log, object, sizeof( object))
   o.stamp ← true

abort( thread)
   foreach (addr, i, size) in VM.currentLog
      copy( addr, 0, log, i, size)
      clearLog()
   thread.postAbortedTransactionException()
```

Fig. 4: Pseudo code for the basic transactional primitives in the Ovm implementation.

object's stamp is not set. The log records triples that consist of the object's address, size, and contents. An object is first added into the log before setting it's stamp field to true. In case the size of the log was misspredicted an exception is thrown. As this signals a programming error, the transaction will not be re-executed automatically. The error propagates into user code, and if not caught will terminate the current task. A transaction is aborted by rolling back the changes it performed and posting an aborted exception so that whenever the thread is sheduled next it recieve the exception which has the effect of unrolling the stack out of the critical section.

Finally, we show the translation of an atomic method, a method such as:

```
@atomic void method() {
   ...body ...
}
```

is transformed by the ahead-of-time compiler into

```
void method() throw PragmaNoPollcheck {
 while ( true ) {
   try {
      acquire( currentThread );
      raw_method(); }
   catch (AbortedTransactionException _) { continue; }
   catch (Throwable t) { exit(); throw t; }
   exit();
   break; }
}
```

In Ovm the `PragmaNoPollcheck` exception is a marker that instructs the compiler not to emit pollchecking code within the body of the translated method, thus ensuring that all transactional operation are executed atomically[6].

Note that in our implementation this translation is performed transparently within the virtual machine, in particular we do not change the signature of the method. Furthermore operations such as `abort()` are Ovm internal and not part of any Java level API.

[6]Poll checks are normally inserted at all potentially recursive method entry, and in loops. Their role is to check if a thread should yield.

5. EXPERIMENTAL RESULTS

To quantify the performance of TLF compared to priority inheritance, Figs. 5 and 6 shows the maximum execution time for high and low priority threads executing the list insertion microbenchmark described in Section 3.1[7]. The benchmark was executed on a 1.66 GHz Athlon, with 1 GB memory in single-user mode. The results were gathered by running the benchmark 7 times, discarding the first run.

The x-axis indicates the size of the list, and the y-axis mesures the maximum time necessary (in microseconds) to insert an object at each position in the list 100 times; each insertion causes the tail of the list following the insertion point to be extended to hold the new item. The lower priority thread inserts repeatedly, while the high priority thread is a periodic thread running under a 10ms period, that inserts a single value 100 times.

As the figure shows, the maximum execution time for higher priority threads under a TLF scheme is roughly a factor of 3 faster than a scheme using PIP, and is effectively the same as a scheme in which no synchronization is used to mediate access to the list. This is because the a relatively small amount of data is logged – when a thread enters the section, it needs to only record the current list element being modified. This is a three word object that contains the current value, a reference to the previous element, and a reference to the next element. If the lower priority thread is aborted, we need only revert the list by restoring this object. Note that maximum execution time for TLF does not increase linearly as the size of the list grows, unlikely the priority inheritance scheme, since the lower priority thread is evicted immediately; using priority inheritance, the wait time for a high priority thread is related to the amount of work remaining in the critical section that must be completed by the lower priority thread.

As expected, the maximum execution time for lower priority threads under TLF is worse than under a PIP scheme since such threads must reexecute their critical section in its entirety once they are aborted. However, the cost of reex-

[7]To quantify Ovm's code generation quality, the non-synchronized version of this benchmark runs roughly 10% slower than the same program in Sun's Hotspot VM.

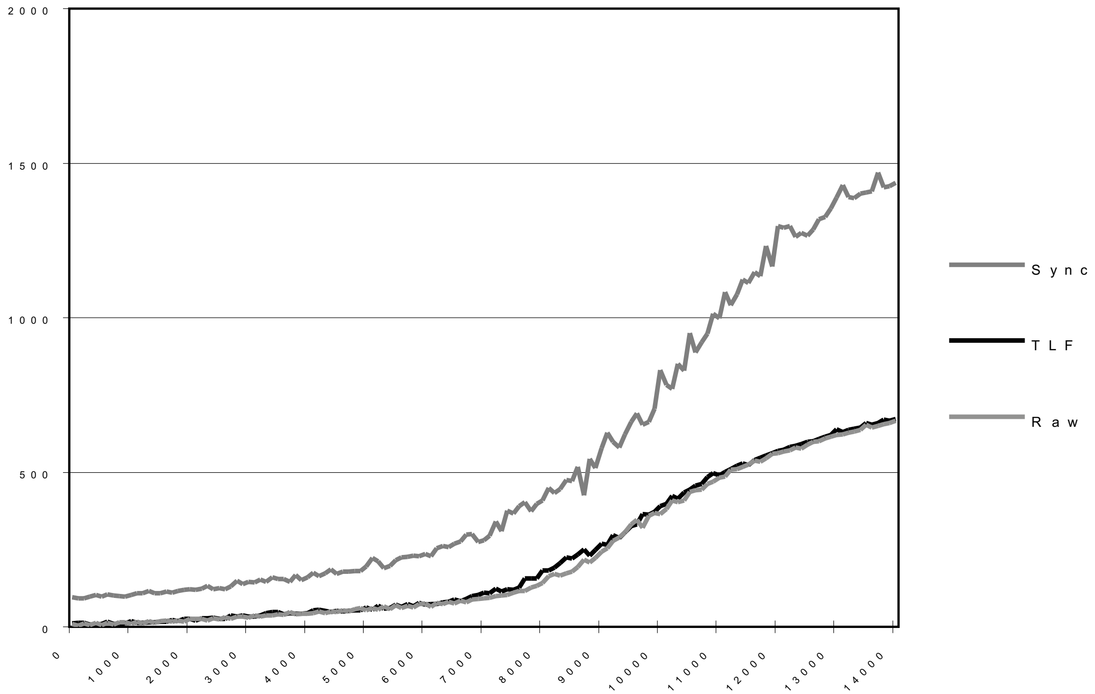


Fig. 5: Responsiveness. Maximum execution times for a high-priority thread with synchronization (Sync), TLF-objects (TLF), and no concurrency control (Raw).

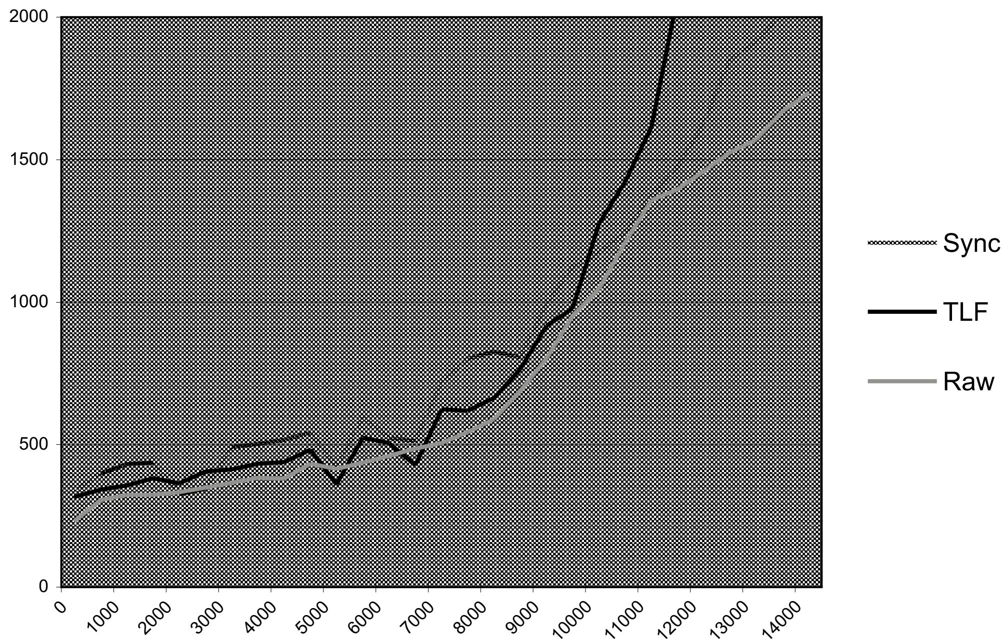


Fig. 6: Responsiveness. Maximum execution times for a low-priority thread with synchronization (Sync), TLF-objects (TLF), and no concurrency control (Raw).

ecution is small compared to the improved throughput for higher priority threads. Of course, changing the periodicity rate of higher priority threads may influence the shape of this graph; a higher-rate would imply low priority threads do less useful work and reexecute more frequently.

6. RELATED WORK

Our use of rollbacks to redo computation inside synchronized sections as a result of an undesirable scheduling is reminiscent of optimistic concurrency protocols first introduced in the 1980's [16] to improve database performance. Given a collection of transactions, the goal in an optimistic concurrency implementation is to ensure that only a serializable schedule results [1, 12, 5]. An analysis of the applicability of lock-free objects in hard realtime systems is described by Anderson *et.al.* [3]. Their focus is on defining schedules that guarantee tight bounds on the number of retries a thread may execute a critical section in the presence of higher-priority threads competing for the same resource. Transactional extensions of the priority ceiling protocol are studied in [26, 29], where a task with its priority higher then a given *abort ceiling* can abort the currently scheduled task.

Transactional lock-free objects have been presented in [2]. The emphasis of that work is to provide language independent support for transactional operations. This means that atomicity and isolation guarantees are limited to the transactional objects. In this paper we give a stronger guarantee, actions of a thread within a transactional section are only observable if the thread commits. The difference is that transaction do not only protect the transactional object but also all other objects that can be reached by following chains of references accessible to thread performing a transaction. This means that we ensure that the programmer will not be able to observe that a thread was reexecuted. A formal semantics for language-level transactions appeared in [31], interested readers are referred to that paper for a discussion of correctness.

Applying these techniques to a broader setting, researchers have also investigated lock-free objects [15, 28, 25]. Our use of logging writes inside synchronized sections distinguishes our approach from lock-free structures because synchronized sections serve as a protection mechanism for multiple (distinct) reads and writes. Conceptually, within its dynamic context, the original values of shared objects are logged and can be reverted if the section aborts. More recently, Herlihy *et.al.* describe a software transactional memory abstraction [13] for Java that allow transactional objects to be dynamically created. Harris and Fraser [11] also describe a lightweight transactional model for Java. Both these efforts share similar goals to ours, but differ in the semantics and implementations of the primitives chosen, and in the application domains addressed. Related efforts at providing hardware support for lock-free execution has been described by Rajwar and Goodman [20] and Rajwar looked at hardware assisted transactional memory [20, 21]

7. CONCLUSION

We have presented a mechanism that addresses the issue of effective scheduling of high-priority threads in priority preemptive realtime systems by introducing lightweight transactional regions that log objects accessed by a thread, and restores the contents of these logs when a thread are interrupted by a higher-priority ones. We show that the time to revert control to higher-priority threads can be bounded to be a function of the size of the transaction-maintained log, and the overheads to maintain transactional consistency is small both in space and time. The model is simple, requiring no major change in programming style or methodology, and is general, applicable to any RTSJ program. Performance results indicate that this scheme significantly outperforms lock-based and priority-inheritance based approaches in terms of responsiveness for high-priority threads.

Our design deliberately trades-off flexibility and generality for simplicity. Effectively dealing with nested transactions, allowing multiple transactional regions to execute concurrently, and reducing the amount of programmer intervention necessary to specify log sizes are several important extensions we hope to pursue.

8. REFERENCES

[1] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *sigmod*, pages 23–34, 1995.

[2] James Anderson, Srikanth Ramamurthy, Mark Moir, and Kevin Jeffay. Lock-free transactions for real-time systems. In *Real-Time Database Systems: Issues and Applications*. Kluwer Academic Publishers, Norwell, Massachusetts, 1997.

[3] James H. Anderson, Srikanth Ramamurthy, and Kevin Jeffay. Real-time computing with lock-free shared objects. *ACM Transactions on Computer Systems*, 15(2):134–165, May 1997.

[4] William S. Beebee, Jr. and Martin Rinard. An implementation of scoped memory for real-time Java. *Emsoft - LNCS*, 2211, 2001.

[5] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[6] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.
`www.javaseries.com/rtj.pdf`.

[7] Dries Buytaert, Frans Arickx, and Johan Vos. A profiler and compiler for the Wonka Virtual Machine. In *USENIX JVM'02 Work in Progress*, San Francisco, CA, August 2002.

[8] Angelo Corsaro and Doug Schmidt. The design and performace of the jRate Real-Time Java implementation. In *The 4th International Symposium on Distributed Objects and Applications (DOA'02)*, 2002.

[9] Urs Gleim. JaRTS: A portable implementation of real-time core extensions for Java. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '02): August 1–2, 2002, San Francisco, California, US*, Berkeley, CA, USA, 2002. USENIX.

[10] John B. Goodenough and Lui Sha. The priority ceiling protocol: A method for minimizing the blocking of high priority Ada tasks. *ACM SIGADA Ada Letters*, 8(7):20–31, Fall 1988.

[11] Tim Harris and Keir Fraser. Language support for lightweight transactions. *ACM SIGPLAN Notices*, 38(11):388–402, November 2003.

[12] Maurice Herlihy. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Transactions on Database Systems*, 15(1):96–124, 1990.

[13] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *ACM Conference on Principles of Distributed Computing*, pages 92–101, 2003.

[14] Timesys Inc. jTime, 2003. `http://www.timesys.com`.

[15] E. H. Jensen, G. W. Hagensen, and J. M. Broughton. A new approach to exclusive data access in shared memory multiprocessors. Technical report, Lawrence Livermore National Laboratories, 1987.

[16] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 9(4):213–226, June 1981.

[17] Douglass Locke, Lui Sha, Ragunathan Rajkumar, John Lehoczky, and Greg Burns. Priority inversion and its control: An experimental investigation. *ACM SIGADA Ada Letters*, 8(7):39–42, Fall 1988.

[18] NASA/JPL and Sun. Golden gate, 2003. `http://research.sun.com/projects/goldengate`.

[19] Kelvin Nilsen. Adding real-time capabilities to Java. *Communications of the ACM*, 41(6):49–56, June 1998.

[20] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 294–305, Austin, Texas, December 1–5, 2001. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[21] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, volume 37, 10 of *ACM SIGPLAN notices*, pages 5–17, New York, October 5–9 2002. ACM Press.

[22] Douglas C. Schmidt, Sumedh Mungee, Sergio Flores-Gaitan, and Aniruddha S. Gokhale. Alleviating priority inversion and non-determinism in real-time CORBA ORB core architectures. In *IEEE Real Time Technology and Applications Symposium*, pages 92–101, 1998.

[23] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 29(9):1175–1185, September 1990.

[24] David Sharp. Real-time distributed object computing: Ready for mission-critical embedded system applications. In *Proceeding of the Third International Symposium on Distribtued-Objects and Applications (DOA'01)*, 2001.

[25] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC '95)*, pages 204–213, New York, August 1995. ACM.

[26] LihChyun Shu, Michal young, and Ragunathan Rajkumar. An abort ceiling protocol for controlling priority inversion. In *Proceedings of the First Worskhop on Real-Time Computing and Applications (RTCSA)*, pages 202–206, 1994.

[27] Fridtjof Siebert. Hard real-time garbage collection in the Jamaica Virtual Machine. In *Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, Hong Kong, 1999.

[28] R.L. Sites. *Alpha Architecture Reference Manual*. Digital, 1992.

[29] Hiroaki Takada and Ken Sakamura. Real-time synchronization protocols with abortable critical sections. In *Proceedings of the First Worskhop on Real-Time Computing and Applications (RTCSA)*, pages 48–52, 1994.

[30] Jörgen Tryggvesson, Torbjörn Mattsson, and Hansruedi Heeb. Jbed: Java for real-time systems. *Dr. Dobb's Journal of Software Tools*, 24(11):78, 80, 82–84, 86, November 1999.

[31] Jan Vitek, Suresh Jagannathan, Adam Welc, and Antony L. Hosking. A Semantic Framework for Designer Transactions. In *Proceedings of the European Symposium on Programming*, March 2004.

[32] Adam Welc, Antony L. Hosking, and Suresh Jagannathan. Transactional Monitors for Concurrent Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'04)*, June 2004.

Snapshots and Software Transactional Memory

Christopher Cole
Northrop Grumman Corporation
chris.cole@ngc.com

Maurice Herlihy
Brown University
Computer Science Department
herlihy@cs.brown.edu.

ABSTRACT

One way that software transactional memory implementations attempt to reduce synchronization conflicts among transactions is by supporting different kinds of access modes. One such implementation, Dynamic Software Transactional Memory (DSTM), supports three kinds of memory access: WRITE mode, which allows an object to be observed and modified, READ mode, which allows an object to be observed but not modified, and TEMP mode, which allows an object to be observed for a limited duration.

In this paper, we examine the relative performance of these modes for simple benchmarks on a small-scale multiprocessor. We find that on this platform and for these benchmarks, the READ and TEMP mode implementations do not substantially increase transaction throughput (and sometimes reduce it). We blame the extra bookkeeping inherent in these modes.

In response, we propose a new SNAP access mode. This mode provides almost the same behavior as TEMP mode, but admits much more efficient implementations.

1. INTRODUCTION

Dynamic Software Transactional Memory (DSTM) [7] is an application programming interface for concurrent computations in which shared data is synchronized without using locks. DSTM manages a collection of *transactional objects*, which are accessed by *transactions*. A transaction is a short-lived, single-threaded computation that either *commits* or *aborts*. If the transaction commits, then these changes take effect; otherwise, they are discarded. A *transactional object* is a container for a regular Java object. A transaction can access the contained object by *opening* the transactional object, and then reading or modifying the regular object. Changes to objects opened by a transaction are not visible to other transactions until the transaction commits. (Changes are discarded if the transaction aborts.) Transactions are *linearizable* [8]: they appear to take effect in a one-at-a-time order.

PODC Workshop on Concurrency and Synchronization in Java Programs, July 25-26, 2004, St. John's, Newfoundland, Canada.

Copyright 2004 Christopher Cole and Maurice Herlihy.

If two transactions open the same object at the same time, a *synchronization conflict* occurs, and one of the conflicting transactions must be aborted. To reduce synchronization conflicts, an object can be opened in one of several *access modes*. An object opened in *WRITE* mode can be read or modified, while an object opened in *READ* mode can only be read. WRITE mode conflicts with both READ and WRITE modes, while READ mode conflicts only with WRITE.

DSTM also provides *TEMP* mode, a special kind of read-only mode that indicates that the transaction may *release* the object before it commits. Once such an object has been released, concurrent accesses of any kind do not cause synchronization conflicts. It is the programmer's responsibility to ensure that releasing objects does not violate transaction linearizability.

The contribution of this paper is to examine the effectiveness of these access modes on a small-scale multiprocessor. We find that the overhead associated with READ and TEMP modes mostly outweighs any advantage in reducing synchronization conflict. To address this issue, we introduce a novel *SNAP* (snapshot) mode, an alternative to TEMP mode with much lower overhead. SNAP mode provides almost the same behavior as TEMP, but much more efficiently.

2. RELATED WORK

Transactional memory was originally proposed as a hardware architecture [6, 16], and continues to be the focus of hardware-oriented research [13]. There have also been several proposals for software transactional memory and similar constructs [2, 1, 9, 12, 15]. Others [10, 14] have studied the performance of read/write locks.

An alternative approach to software transactional memory (STM) is due to Harris and Fraser [5]. Their STM implementation is *word-based*: the unit of synchronization is a single word of memory. An uncontended transaction that modifies N words requires $2N + 1$ compare-and-swap calls. Fraser [4] has proposed a FSTM implementation that is *object-based*: the unit of synchronization is an object of arbitrary size. Here, an uncontended transaction that modifies N objects also requires $2N+1$ compare-and-swap calls. Herlihy *et al.* [7] have proposed an object-based DSTM implementation, described below, in which an uncontended transaction that modifies N objects requires $N+1$ compare-and-swap calls, but sometimes requires traversing an additional level of indirection. In both object-oriented STM implementations, objects must be copied before they can be

modified. Marathe and Scott [11] give a more detailed comparison of these STM implementations.

3. DSTM IMPLEMENTATION

Here we summarize the relevant aspects of the DSTM implementation (a more complete description appears elsewhere [7]).

When transaction A attempts to open an object, it may discover that the object has already been opened by a transaction B. A can decide either to back off and give B a chance to complete, or it can proceed, forcing B to abort. The policy decision is handled by a separate *Contention Manager* module.

Each time a transaction opens an object, it checks whether it has been aborted by a synchronization conflict, a process called *validation*. This check prevents an aborted transaction from wasting resources, and also ensures that each transaction has a consistent view of the transactional objects.

When a transaction opens a transactional object, it acquires a reference to a *version* of that object. If the object is opened in WRITE mode, the transaction may modify that version, and otherwise it may only observe that version.

Opening an object in WRITE mode requires creating a new version (by copying the old one) and executing a compare-and-swap instruction. When an object is opened in READ mode, the transaction simply returns a reference to the most recent committed version. The transaction records that reference in a private *read table*. To validate, the transaction checks whether each of its version references is still current. This implementation has the advantage that reading does not require an expensive compare-and-swap instruction. It has two disadvantages: validation takes time linear in the number of objects read, and the contention manager cannot tell whether an object is open in READ mode. For this reason, we call this implementation the *invisible read*.

Because of these disadvantages, we devised an alternative READ mode implementation, which we call the *visible read*. This implementation is similar to WRITE mode, except that it does not copy the current version, and the object keeps a list of reading transactions. Validating a transaction takes constant time, and reads are visible to the contention manager. Each read does require a compare-and-swap, and opening an object in WRITE mode may require traversing a list of prior readers.

Similarly, TEMP mode also has both visible and invisible implementations. Releasing an object either causes the version to be discarded (invisible) or the reader removed from the list (visible).

4. BENCHMARKS

An `IntSet` is an ordered linked list of integers providing `insert()` and `delete()` methods. We created three benchmarks: WRITE, READ, and RELEASE. Each benchmark runs for twenty seconds randomly inserting or deleting values from the list. The WRITE benchmark opens each list element in WRITE mode. The READ benchmark opens each list element in READ mode until it discovers the element to modify, which it reopens in WRITE mode. The RELEASE benchmark opens each element in TEMP mode, releasing each element after opening its successor (similar to lock coupling). Each experiment was run using the `Polite` contention manager which uses exponential back-off when conflicts arise. For example, when transaction A is about to open an object already opened by transaction B, the `Polite` contention manager backs off several times, doubling each expected duration, to give B a chance to finish. If B does not finish in that duration, then A aborts B, and proceeds.

The benchmarks were run on a machine with four Intel Xeon processors. Each processor runs at 2.0 GHz and has 1 GB of RAM. The machine was running Debian Linux and each experiment was run 100 times for twenty seconds each. The performance data associated with individual method calls was extracted using the Extensible Java Profiler [3]. Each benchmark was run using 1, 4, 16, 32, and 64 threads. The single-thread case is interesting because it provides insight into the amount of overhead the experiment incurred. In the four-thread benchmarks, the number of threads matches the number of processors, while the benchmarks using 16, 32, and 64 thread show how the transactions behave when they share a processor. To control the list size, the integer values range only from 0 to 255.

	Invisible		Visible	
WRITE	36.6		22.3	
READ	4.9	(13.5%)	23.9	(107.3%)
RELEASE	19.9	(54.5%)	21.2	(95.2%)

Table 1: Single-Thread Throughput

5. BENCHMARK RESULTS

Table 1 shows the single-processor throughput (transactions committed per millisecond) for both the invisible and visible implementations. In the single-thread benchmarks, there is no concurrency, and hence no synchronization conflicts, so the throughput numbers reflect the modes' inherent overheads.

To ease comparisons, the READ and RELEASE throughput numbers are labeled with their percentages of the comparable WRITE benchmark. (For example, the invisible READ's throughput of 4.9 is 13.5% of the invisible WRITE's throughput.)

The invisible WRITE had better throughput than the visible WRITE, because when the visible WRITE opens an object, it checks whether any transaction has the object open in READ mode. Even though there are no such transactions (in a single-threaded benchmark), the check takes time. The invisible READ performed poorly because it validates each object previously open for READ each time a new object is opened. The visible READ performed slightly better than WRITE because it does not need to copy the object version. The invisible RELEASE performed better than the invisible READ because it releases objects, and once an object is released, it no longer needs to be validated.

Table 2 shows the time (in nanoseconds) for common method calls. The WRITE and "READ & TEMP" rows show the time needed to open an object in those modes, the UPGRADE row shows the time needed to upgrade from READ or TEMP mode to WRITE mode, and the RELEASE line shows the time needed to release an object opened in TEMP mode. These timings to not always mirror the benchmark throughput numbers because the visible implementation incurs all its overhead in calls to the `open()` method,

	Invisible	Visible
WRITE	180	730
READ &TEMP	280	135
UPGRADE	250	160
RELEASE	90	40

Table 2: Common Method Call Timings (nanoseconds)

while the invisible implementation incurs costs each time the current transaction is validated as a side-effect of other DSTM calls.

We now turn our attention from single-thread executions, where overhead dominates, to multi-thread executions, where we hope to see gains in READ or RELEASE mode due to reduced synchronization conflicts.

Table 3 shows the transactions-per-millisecond throughput of the invisible implementation for varying numbers of threads, and Table 4 does the same for the visible implementation.

Surprisingly, perhaps, the concurrency allowed in READ and TEMP did not overcome the overhead in either implementation (with one minor exception). In the invisible implementation, a transaction takes an excessive amount of time to traverse the list because it must validate its read-only table with each DSTM API call. A transaction attempting to insert a large integer may never find the integer's position in the list before being aborted. In the visible implementation, the single-threaded benchmark has a slight advantage because it does not need to copy the version being opened. In the multithreaded benchmarks, however, the visible implementation incurs overhead because it must traverse and prune a non-trivial list of readers.

We ran a number of other experiments, including longer lists and adding additional delays ("work") to transactions. The results, omitted here for brevity, are essentially unchanged: overall, READ and TEMP modes do not enhance throughput.

Naturally, these results are valid only for the specific implementation and platform tested here. It may be that platforms with more processors, or a different contention manager, or different internals would behave differently. Nevertheless, to address the problem of increasing throughput on our four-processor platform, we devised a new SNAP mode described in the next section.

6. SNAPSHOT MODE

In an attempt to find a low-overhead alternative, we devised a new *snapshot* mode for opening an object.

```
public TMCloneable open(SNAP)

  throws DeniedException
```

This method returns a reference to the version that would have been returned by a call to `open(READ)`. It does not actually open the object for reading, and the DSTM does not keep any record of the snapshot. All methods throw `DeniedException` if the current transaction has been aborted.

The `version` argument to the next three methods is a version reference returned by a prior call to `open(SNAP)`.

```
public void snapValidate(TMCloneable version)
  throws DeniedException, SnapshotException
```

The call returns normally if a call to `open(SNAP)` (or `open(READ)`) would have returned the same version reference. Otherwise, the call throws a `SnapshotException`. Throwing this exception does not abort the current transaction, allowing the transaction to retry another snapshot.

```
public TMCloneable
  snapUpgradeRead(TMCloneable version)
  throws SnapshotException, DeniedException
```

If the version argument is still current, this method opens the object in READ mode, and otherwise throws an exception (`SnapshotException`).

```
public TMCloneable snapUpgradeWrite(TMCloneable)
  throws SnapshotException, DeniedException
```

If the version argument is still current, this method opens the object in WRITE mode, and otherwise throws an exception (`SnapshotException`).

Objects opened in TEMP mode are typically used in one of the following three ways. Most commonly, an object is opened in TEMP mode and later released. The transaction will be aborted if the object is modified in the interval between when it is opened and when it is released, but the transaction will be unaffected by modifications that occur after the release.

```
Entry entry = (Entry)tmObject.open(TEMP);
...
entry.release();
```

The same effect is achieved by the following code fragment:

```
Entry entry = (Entry)tmObject.open(SNAP);
...
tmObject.snapValidate(entry);
```

The first call returns a reference to the object version that would have been returned by `open(TEMP)` (or `open(READ)`), and the second call checks that the version is still valid. There is no need for an explicit release because the transaction will be unaffected if that version is changed (assuming it does not validate again).

Sometimes an object is opened in TEMP mode and never released (which is equivalent to opening the object in READ mode). To get the same effect in SNAP mode, the transaction must apply `snapUpgradeRead` to the object, atomically validating the snapshot and acquiring READ access.

Finally, an object may be opened in TEMP mode and later upgraded to WRITE mode. The snapUpgradeWrite() method provides the same effect.

To illustrate how one might use SNAP mode, figure 1 shows the code for a `insert()` method based on SNAP mode. It is not necessary to understand this code in detail, but there are three lines that merit attention. As the transaction traverses the list, `prevObject` is a reference to the last transactional object accessed, and `lastObject` is a reference to that object's predecessor in the list. In the line marked A, the method validates for the last time that `lastObject` is still current, effectively releasing it. If the method discovers that the value to be inserted is already present, then in the line marked B, it upgrades access to the predecessor entry to READ, ensuring that no other transaction deletes that value. Similarly, if the method discovers that the value to be

	1 Thread		4 Threads		16 Threads		32 Threads		64 Threads	
WRITE	36.6		35.7		32.9		29.6		24.7	
READ	4.9	(13.5%)	1.7	(4.7%)	0.6	(1.7%)	0.5	(1.8%)	0.5	(2.2%)
RELEASE	19.9	(54.5%)	8.5	(23.7%)	4.2	(12.6%)	3.8	(12.8%)	3.7	(15.1%)

Table 3: Invisible Implementation

	1 Thread		4 Threads		16 Threads		32 Threads		64 Threads	
WRITE	22.3		23.1		21.4		20.0		17.6	
READ	23.9	(107.3%)	0.1	(0.3%)	0.2	(0.9%)	0.2	(1.1%)	0.2	(1.2%)
RELEASE	21.2	(95.2%)	0.03	(0.1%)	0.1	(0.7%)	0.2	(1.0%)	0.3	(1.6%)

Table 4: Visible Implementation

inserted is not present, it upgrades access to the predecessor entry to WRITE, so it can insert the new entry.

The principal benefit of SNAP mode is that it can be implemented very efficiently. This mode is "stateless", in the sense that the DSTM run-time does not need to keep track of versions opened in SNAP mode (unlike READ mode). The `snapValidate`, `snapUpgradeRead` and `snapUpgradeWrite` calls simply compare their arguments to the object's current version. Moreover, SNAP mode adds no overhead to transaction validation.

7. SNAP BENCHMARKS

The results of running the same benchmark in SNAP mode instead of TEMP mode are shown in Tables 5 (invisible) and 6 (visible). For both visible and invisible implementations, SNAP mode has substantially higher throughput than both READ and TEMP mode. Opening an object in SNAP mode takes about 100ns, including validation. It takes about 125ns to upgrade an object opened in SNAP mode to to WRITE mode.

Even though invisible SNAP mode outperforms invisible READ and TEMP, it still has lower throughput than invisible WRITE. We believe this disparity reflects inherent inefficiencies in the invisible READ implementation. The invisible SNAP implementation must upgrade to invisible READ mode whenever it observes that a value is absent (to ensure it is not inserted), but transactions that open objects in invisible READ mode are often aborted, precisely because they are invisible to the contention manager.

While the result of combining invisible READ and SNAP modes is disappointing, the result of combining visible READ and SNAP modes is dramatic: here is the first alternative mode that outperforms WRITE mode across the board.

To investigate further, we implemented some benchmarks that mixed "modifying" method calls with "observer" (read-only) method calls. We introduced a `contains()` method that searches the list for a value. We tested benchmarks in which the percentages of modifying calls (`insert()` and `delete()`) varied were 50% (Table 7), 10% (Table 8), 1% (Table 9), and 0% (Table 10). Each of the SNAP mode benchmarks had higher throughput than its WRITE counterpart, and was the only benchmark to do so.

8. CONCLUSIONS

More research is needed to determine the most effective methods for opening objects concurrently in software transactional memory. We were surprised by how poorly READ and TEMP modes performed on our small-scale benchmarks. While our SNAP mode implementation substantially outperforms both READ and TEMP modes, it is probably appropriate only for advanced programmers. It would be worthwhile investigating whether or not a contention management scheme could increase the throughput of read transactions, or if there are more efficient designs for tracking objects open for reading.

Notice that the DSTM guarantees that every transaction, even ones that are doomed to abort, sees a consistent set of objects. For the invisible read, this guarantee is expensive, because each object read must be revalidated every time a new object is opened. An alternative approach, used in Fraser's FSTM [4], does not guarantee that transactions see consistent states, but uses periodic checks and handlers to protect against memory faults and unbounded looping due to inconsistencies. The relative merits of these two approaches remains an open topic for further research.

```
  public boolean insert(int v) {
    List newList = new List(v);
    TMObject newNode = new TMObject(newList);
    TMThread thread = (TMThread)Thread.currentThread();
    while (thread.shouldBegin()) {
      thread.beginTransaction();
      boolean result = true;
      try {
        TMObject lastNode = null;
        List lastList = null;
        TMObject prevNode = this.first;
        List prevList = (List)prevNode.openSnap();
        TMObject currNode = prevList.next;
        List currList = (List)currNode.openSnap();
        while  (currList.value < v) {
          if (lastNode != null)
/*A*/         lastNode.snapValid(lastList);
          lastNode = prevNode;
          lastList = prevList;
          prevNode = currNode;
          prevList = currList;
          currNode = currList.next;
          currList = (List)currNode.openSnap();
        }
        if (currList.value == v) {
/*B*/      prevNode.snapUpgradeRead(prevList);
          result = false;
        } else {
          result = true;
/*C*/     prevList = (List)prevNode.snapUpgradeWrite(prevList);
          newList.next = prevList.next;
          prevList.next = newNode;
        }
        // final validations
        if (lastNode != null)
          lastNode.snapValid(lastList);
        currNode.snapValid(currList);
      } catch (SnapshotException s) {
        thread.getTransaction().abort();
      } catch (DeniedException d) {
      }
      if (thread.commitTransaction()) {
        return result;
      }
    }
    return false;
  }
```

Figure 1: SNAP-mode insert method

	1 Thread		4 Threads		16 Threads		32 Threads		64 Threads	
WRITE	36.6		35.7		32.9		29.6		24.7	
READ	4.9	(13.5%)	1.7	(4.7%)	0.6	(1.7%)	0.5	(1.8%)	0.5	(2.2%)
RELEASE	19.9	(54.5%)	8.5	(23.7%)	4.2	(12.6%)	3.8	(12.8%)	3.7	(15.1%)
SNAP	62.4	(170.7%)	16.7	(46.8%)	10.9	(33.2%)	10.2	(34.6%)	9.6	(39.0%)

Table 5: SNAP with Invisible

	1 Thread		4 Threads		16 Threads		32 Threads		64 Threads	
WRITE	22.3		23.1		21.4		20.0		17.6	
READ	23.9	(107.3%)	0.1	(0.3%)	0.2	(0.9%)	0.2	(1.1%)	0.2	(1.2%)
RELEASE	21.2	(95.2%)	0.03	(0.1%)	0.1	(0.7%)	0.2	(1.0%)	0.3	(1.6%)
SNAP	104.8	(469.9%)	62.3	(269.6%)	56.4	(263.2%)	42.2	(210.7%)	37.8	(214.9%)

Table 6: SNAP with Visible

	1 Thread		4 Threads		16 Threads		32 Threads		64 Threads	
WRITE	22.1		23.0		21.4		19.6		17.2	
READ	23.3	(105.6%)	0.2	(0.8%)	0.5	(2.3%)	0.9	(4.8%)	1.1	(6.6%)
RELEASE	20.8	(94.2%)	0.1	(0.4%)	0.5	(2.2%)	0.8	(4.3%)	1.1	(6.3%)
SNAP	108.4	(491.5%)	81.0	(352.1%)	72.4	(337.7%)	59.5	(302.9%)	31.7	(184.0%)

Table 7: Visible with 50% Modification

	1 Thread		4 Threads		16 Threads		32 Threads		64 Threads	
WRITE	22.3		22.9		20.9		19.4		17.2	
READ	23.6	(106.0%)	0.5	(2.3%)	2.8	(13.4%)	3.9	(20.1%)	3.1	(18.2%)
RELEASE	21.1	(94.7%)	0.4	(1.7%)	2.4	(11.7%)	3.2	(16.7%)	3.0	(17.6%)
SNAP	109.7	(491.6%)	86.9	(379.4%)	88.1	(420.7%)	58.3	(300.7%)	19.8	(115.1%)

Table 8: Visible with 10% Modification

9. REFERENCES

[1] Yehuda Afek, Dalia Dauber, and Dan Touitou. Wait-free made fast. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 538–547. ACM Press, 1995.

[2] Anderson and Moir. Universal constructions for large objects. In *WDAG: International Workshop on Distributed Algorithms*. LNCS, Springer-Verlag, 1995.

[3] `http://ejp.sourceforge.net/`.

[4] Keir Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge Computer Laboratory, February 2004.

[5] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402. ACM Press, 2003.

[6] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.

[7] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101. ACM Press, 2003.

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

[9] Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, pages 151–160. ACM Press, 1994.

[10] Theodore Johnson. Approximate analysis of reader and writer access to a shared resource. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 106–114. ACM Press, 1990.

[11] Virendra J. Marathe and Michael L. Scott. A qualitative survey of modern software transactional memory systems. Technical Report 839, Department of Computer Science University of Rochester, June 2004.

[12] Mark Moir. Transparent support for wait-free

	1 Thread		4 Threads		16 Threads		32 Threads		64 Threads	
WRITE	20.9		20.9		20.3		18.0		15.9	
READ	22.2	(105.9%)	5.1	(24.4%)	11.5	(56.8%)	7.8	(43.3%)	1.9	(11.8%)
RELEASE	20.8	(99.4%)	5.4	(25.7%)	3.4	(16.9%)	2.9	(16.3%)	2.7	(16.7%)
SNAP	105.0	(501.2%)	97.4	(466.0%)	98.1	(483.4%)	62.8	(348.9%)	24.6	(154.7%)

Table 9: Visible with 1% Modification

	1 Thread		4 Threads		16 Threads		32 Threads		64 Threads	
WRITE	11.5		11.8		11.0		10.2		9.0	
READ	12.4	(107.7%)	9.0	(76.1%)	4.6	(42.3%)	2.4	(23.6%)	0.5	(6.0%)
RELEASE	11.0	(95.6%)	3.2	(27.5%)	2.4	(21.6%)	2.2	(21.5%)	2.0	(22.4%)
SNAP	61.3	(531.0%)	55.6	(472.4%)	61.0	(555.9%)	67.3	(662.5%)	71.3	(790.0%)

Table 10: Visible with 0% Modification

transactions. In *Proceedings of the 11th International Workshop on Distributed Algorithms*, pages 305–319, 1997.

[13] Ravi Rajwar and James R Goodman. Transactional lock-free execution of lock-based programs.

[14] Martin Reiman and Paul E. Wright. Performance analysis of concurrent-read exclusive-write. In *Proceedings of the 1991 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 168–177. ACM Press, 1991.

[15] Nir Shavit and Dan Touitou. Software transactional memory. In *Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[16] Janice M. Stone, Harold S. Stone, Philip Heidelberger, and John Turek. Multiple reservations and the Oklahoma update. *IEEE Parallel Distrib. Technol.*, 1(4):58–71, 1993.

Contention Management in Dynamic Software Transactional Memory*

William N. Scherer III and Michael L. Scott
Department of Computer Science
University of Rochester
Rochester, NY 14627-0226
{scherer,scott}@cs.rochester.edu

Abstract

Obstruction-free concurrent algorithms differ from those with stronger nonblocking conditions in that they separate progress from correctness. While it must always maintain data invariants, an obstruction-free algorithm need only guarantee progress in the absence of contention. The programmer can (and indeed must) address progress as an out-of-band, orthogonal concern.

In this work we consider the Java-based obstruction-free Dynamic Software Transaction Memory (DSTM) system of Herlihy et al. When two or more transactions attempt to access the same block of transactional memory concurrently, at least one transaction must be aborted. The decision of which transaction to abort, and under what conditions, is the *contention management* problem. We introduce several novel policies for contention management, and evaluate their performance on a variety of benchmarks, all running on a 16-processor SunFire 6800. We also evaluate the marginal utility of earlier, but somewhat more expensive detection of conflicts between readers and writers.

1 Introduction

Non-blocking algorithms are notoriously difficult to design and implement. Although this difficulty is partially inherent to asynchronous interleavings due to concurrency, it may also be ascribed to the many different concerns that must be addressed in the design process. With lock-free synchronization, for example, one must not only ensure that the algorithm functions correctly, but also guard against livelock. With wait-free synchronization one must additionally ensure that every thread makes progress in bounded time; in general this requires that one "help" conflicting transactions rather than aborting them.

Obstruction-free concurrent algorithms[3] lighten the burden by separating progress from correctness, allowing programmers to address progress as an out-of-band, orthogonal concern. The core of an obstruction-free algorithm need only guarantee progress when only one thread is running (though other threads may be in arbitrary states).

Dynamic software transactional memory (DSTM) [4] is a *general purpose* system for obstruction-free implementation of arbitrary concurrent data structures. Though applicable in principle to many programming environments, it is currently targeted at Java, where automatic garbage collection simplifies storage management concerns. DSTM is novel in its support for dynamically allocated objects and transactions, and for its use of modular contention managers to separate issues of progress from the correctness of a given data structure.

Contention management in DSTM may be summed up as the question: what do we do when two transactions have conflicting needs to access a single block of memory? At one extreme, a policy that never aborts an "enemy" transaction can lead to deadlock in the event of priority inversion or mutual blocking, to starvation if a transaction deterministically encounters enemies, and to a major loss of performance in the face of page faults and preemptive scheduling. At the other extreme, a policy that always aborts an enemy may also lead to starvation,

*This work was supported in part by NSF grants numbers EIA-0080124, CCR-9988361, and CCR-0204344, by DARPA/AFRL contract number F29601-00-K-0182, and by financial and equipment grants from Sun Microsystems Laboratories.

or to livelock if transactions repeatedly restart and then at the same step encounter and abort each other. A good contention manager must lie somewhere in between, aborting enemy transactions often enough to tolerate page faults and preemption, yet seldom enough to make starvation unlikely in practice. We take the position that policies must also be provably deadlock free. It is the duty of the contention manager to ensure progress; we say that it does so out-of-band because its code is entirely separate from that of the transactions it manages, and contributes nothing to their conceptual complexity.

Section 2 begins our study with an overview of DSTM. Section 3 then describes our contention management interface and presents several novel contention management policies. Section 4 evaluates the performance of these policies on a suite of benchmark applications. Our principal finding is that different policies perform best for different combinations of application, workload, and level of contention. The out-of-band nature of contention managers in DSTM is thus quite valuable: it allows the programmer to choose the policy best suited to a given situation. We also find that early detection of conflicts between readers and writers can be either helpful or harmful, again depending on application, workload, and level of contention. This suggests that it may be desirable to provide both "visible" and "invisible" reads in future versions of DSTM. We summarize our conclusions in Section 5.

2 Dynamic STM

DSTM transactions operate on blocks of memory. Typically, each block corresponds to one Java object. Each transaction performs a standard sequence of steps: initialize; open and update one or more blocks (possibly choosing later blocks based on data in earlier blocks); attempt to commit; if committing fails, retry. Blocks can be opened for full read-write access, read-only access, or for temporary access (where the block can later be discarded if changes to by other transactions won't affect the viability of current one).

Under the hood, each block is represented by a *TMObject* data structure that consists of a pointer to a *Locator* object. The Locator in turn has pointers to the transaction that has most recently opened the TMObject, together with old and new data object pointers (see Figure 1).

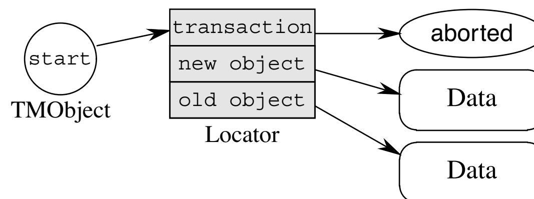


Figure 1: Transactional object structure

When a transaction attempts to open a block, we first read the Locator pointer in the TMObject for the block. We then read the status word for the transaction that has most recently updated the block to determine whether the old or the new data object pointer is current. If this status word is `committed`, then the new object is current; otherwise the old one is. Next, we build a new Locator that points to our transaction and has the active version of the data as its old object. We copy the data for the new object and then atomically update the TMObject to point to our new Locator. Finally, we store the new Locator and its corresponding TMObject in our transaction record.

To validate that a transaction is still viable, we verify that each Locator in it is still the current Locator for the appropriate TMObject. Finally, to commit a transaction, we atomically update our transaction's status word from `active` to `committed`. This update, if successful, signals that all of our updated versions of the data objects are the ones that are current.

With this implementation, only one transaction at a time can have a block open for write access, because only one can have its Locator pointed to by the block's TMObject. If another transaction wishes to write an already-opened block, it must first abort the "enemy" transaction. This is done by atomically updating that transaction's status field from `active` to `aborted`. Once this is done, the aborted transaction's attempt to commit is doomed to fail.

2.1 Visible and Invisible Reads

In the original version of the DSTM, read-only access to blocks was achieved by creating a private copy. The Locator for the block was then stored with the transaction record. At validation time, a conflict would be detected if the current and stored Locators did not match. We term this implementation an *invisible* read because it associates no artifact

from the reading transaction with the block. A competing transaction attempting to open the block for write access cannot tell that readers exist, so there is no "hook" through which contention management can address the potential conflict.

An alternative implementation of read-only access adds a pointer to the transaction to a linked list of readers for the block. This implementation adds some overhead to read operations and increases the complexity of subsequently opening the block for read-write access: the writer must traverse the list and explicitly abort the readers. In exchange for this overhead, however, we gain the ability to explicitly manage conflicts between readers and writers, and to abort doomed readers early.

2.2 Limiting Mutual Abortion

If a thread decides to abort another transaction in the current DSTM implementation, it does so without first checking to see whether its own transaction remains viable. There is thus a significant window during which two transactions can detect a mutual conflict and decide to abort each other. To narrow (though not eliminate) this window, we propose checking the status of the current transaction immediately before aborting an enemy. This is a very low-overhead change: it consists of a single read of the transaction's status word.

3 Contention Management Policies

The contention management interface for the DSTM includes notification methods for about various events that can occur during the processing of transactions, plus two request methods that ask the manager to make a decision. Notifications include

- Beginning a transaction
- Successfully committing a transaction
- Failing to commit a transaction
- Self-abortion of a transaction
- Beginning an attempt to open a block (for read-only, temporary, or read-write access)
- Successfully opening a block (3 variants)
- Failing to open a block (3 variants) due to failed transaction validation
- Successfully changing access to read-only/temporary/read-write on a block already open in another mode (6 total variants)

Requests are

- Should the transaction (re)start at this time?
- Should the transaction abort an enemy?

Because the contention management methods are called in response to DSTM operations, they must themselves be non-blocking. Additionally, a contention manager must always (eventually) abort a competing transaction (else deadlock could result). There are no further correctness considerations for contention managers: one is free to design them as needed for overall efficiency. As illustrated by the sample managers presented here and in the original DSTM paper [4], the design space is quite large. In this work, we begin to explore that space by adapting policies used in a variety of related problem domains.

3.1 Aggressive

The Aggressive manager ignores all notification methods, and always chooses to abort an enemy transaction at conflict time. Although this makes it highly prone to livelock, it forms a useful baseline against which to compare other policies.

3.2 Polite

The Polite contention manager uses exponential backoff to resolve conflicts encountered when opening blocks. Upon detecting contention, it spins for a period of time proportional to 2^nns, where n is the number of retries that have been necessary so far for access to a block. After a maximum of 8 retries, the polite manager unconditionally aborts an enemy transaction. One might expect the Polite manager to be particularly vulnerable to performance loss due to preemption and page faults.

3.3 Randomized

A very simple contention manager, the Randomized policy ignores all notification methods. When it encounters contention, it flips a coin to decide between aborting the other transaction and waiting for a random interval of up to a certain length. The coin's bias and the maximum waiting interval are tunable parameters; we used 50% and 64ns, respectively.

3.4 Karma

The Karma manager attempts to judge the amount of work that a transaction has done so far when deciding whether to abort it. Although it is hard to estimate the amount of work that a transaction performs on the data contained in a block, the number of blocks the transaction has opened may be viewed as a rough

indication of investment. For system throughput, aborting a transaction that has just started is preferable to aborting one that is in the final stages of an update spanning tens (or hundreds) of blocks.

The Karma manager tracks the cumulative number of blocks opened by a transaction as its priority. More specifically, it resets the priority of the current thread to zero when a transaction commits and increments that priority when the thread successfully opens a block. When a thread encounters a conflict, the manager compares priorities and aborts the enemy if the current thread's priority is higher. Otherwise, the manager waits for a fixed amount of time to see if the enemy has finished. Once the number of retries plus the thread's current priority exceeds the enemy's priority, the manager kills the it.

What about the thread whose transaction was aborted and has to start over? In a way, we owe it a karmic debt: it was killed before it had a chance to finish its work. We thus allow it to keep the priority ("karma") that it had accumulated before being killed, so it will have a better chance of being able to finish its work in its "next life". Note that every thread necessarily gains at least one point in each unsuccessful attempt. This allows short transactions to gain enough priority to compete with others of much greater lengths.

3.5 Eruption

The Eruption manager is similar to the Karma manager in that both use the number of opened blocks as a rough measure of investment. It resolves conflicts, however, by increasing pressure on the transactions that a blocked transaction is waiting on, eventually causing them to "erupt" through to completion. Each time a block is successfully opened, the transaction gains one point of "momentum" (priority). When a transaction finds itself blocked by one of higher priority, it adds its momentum to the conflicting transaction and then waits for it to complete. Like the Karma manager, Eruption waits for time proportional to the difference in priorities before killing an enemy transaction.

The reasoning behind this management policy is that if a particular transaction is blocking resources critical to many other transactions, it will gain all of their priority in addition to its own and thus be much more likely to finish quickly and stop blocking the others. Hence, resources critical to many transactions will be held (ideally) for short periods of time. Note that while a transaction is blocked, other transactions can accumulate behind it and increase its priority enough to outweigh the transaction blocking it.

Mutually blocking transactions are a potential problem, since one will have to time out before either can progress. To keep this problem from recurring, the Eruption manager halves the accumulated priority of an aborted transaction.

In addition to the Karma manager, Eruption draws on Tune *et al.*'s `QOldDep` and `QCons` techniques for marking instructions in the issue queue of a superscalar out-of-order microprocessor to predict instructions most likely to lie on the critical path of execution [8].

3.6 KillBlocked

Adapted from McWherter *et al.*'s POW lock prioritization policy [6], the KillBlocked manager is less complex than Karma or Eruption, and features rapid elimination of cyclic blocking. The manager marks a transaction as blocked when first notified of an (unsuccessful) non-initial attempt to open a block. The manager aborts an enemy transaction whenever (a) the enemy is also blocked, or (b) a maximum waiting time has expired.

3.7 Kindergarten

Based loosely on the conflict resolution rule in Chandy and Misra' Drinking Philosophers problem [2], the Kindergarten manager encourages transactions to take turns accessing a block. For each transaction T, the manager maintains a list (initially empty) of enemy transactions in favor of which T has previously aborted. At conflict time, the manager checks the enemy transaction and aborts it if present in the list; otherwise it adds the enemy to the list and backs off for a short length of time. It also stores the enemy's hash code as the transaction on which T is currently waiting. If after a fixed number of backoff intervals it is still waiting on the same enemy, the Kindergarten manager aborts transaction T. When the calling thread retries T, the Kindergarten manager will find the enemy in its list and abort it.

3.8 Timestamp

The Timestamp manager is an attempt to be as fair as possible to transactions. The manager records

the current time at the beginning of each transaction. When it encounters contention between transaction T and some enemy, it compares timestamps. If T's timestamp is earlier, the manager aborts it. Otherwise, it begins waiting for a series of fixed intervals. After half the maximum number of these intervals, it flags the enemy transaction as potentially defunct. After the maximum number of intervals, if the defunct flag has been set all along, the manager aborts the enemy. If the flag has ever been reset, however, the manager doubles the wait period and starts over. Meanwhile, if the enemy transaction performs any transaction-related operations, its manager will see and clear the defunct flag.

Timestamp's goal is to avoid aborting an earlier-started transaction regardless of how slowly it runs or how much work it performs. The defunct flag provides a feedback mechanism for the other transaction to enable us to distinguish a dead transaction from one that is still active. Of course, the use of timestamps to resolve contention is hardly new to this context; similar algorithms have been in use in the database community for almost 25 years [1].

3.9 QueueOnBlock

The QueueOnBlock manager reacts to contention by linking itself into a queue hosted by the enemy transaction. It then spins on a "finished" flag that is eventually set by the enemy transaction's manager at completion time. Alternatively, if it has waited for too long, it aborts the enemy transaction and continues; this is necessary to preserve obstruction freedom. For its part, the enemy transaction walks through the queue setting flags for competitors when it is either finished or aborted. Note that not all of these competitors need have been waiting for the same block. If more than one was, any that lose the race to next open it will enqueue themselves with the winner.

Clearly, QueueOnBlock does not effectively deal with block dependency cycles: at least one transaction must time out before either can progress. On the other hand, if the block access pattern is free of such dependencies, this manager will usually avoid aborting another transaction.

4 Experimental Results

4.1 Benchmarks

We present experimental results for five benchmarks. Three implementations of an integer set (IntSet, IntSetRelease, RBTree) are drawn from the original DSTM paper [4]. These three repeatedly but randomly insert or delete integers in the range 0..255 (keeping the range restricted increases the probability of contention). The total number of successful operations completed in a fixed period of time is reported as the overall throughput for the benchmark. The first implementation uses a sorted linked list in which every block is opened for write access; the second uses a sorted linked list in which blocks are first opened transiently and then released as the transaction approaches its insertion/deletion point; the third uses a red-black tree in which blocks are first opened for read-only access, then upgraded to read-write access when changes are necessary.

The fourth benchmark (Counter) is a simple shared counter that threads increment via transactions. The fifth (LFUCache) is a simulation of cache replacement in an HTTP web proxy using the least-frequently used (LFU) algorithm [7]. caching community, this algorithm is treated as folklore; however, an algorithm assumes that frequency (rather than recency) of web page access is the best predictor for whether a web page is likely to be accessed again in the future (and thus, worth caching).

The simulation uses a two-part data structure to emulate the cache. The first part is a lookup table of 2048 integers, each of which represents the hash code for an individual HTML page. These are stored as a single array of `TMObjects`. Each contains the key value for the object (an integer in the simulation) and a pointer to the page's location in the main portion of the cache. The pointers are null if the page is not currently cached.

The second, main part of the cache consists of a fixed size priority queue heap of 255 entries (a binary tree, 8 layers deep), with lower frequency values near the root. Each priority queue heap node contains a frequency (total number of times the cached page has been accessed) and a page hash code (effectively, a backpointer to the lookup table).

Worker threads repeatedly access a page. To approximate the workload for a real web cache, we pick

pages randomly from a Zipf distribution with exponent 2. So, for page i, the cumulative probability $p_c(i) \propto \sum_{0 \leq j \leq i} 1/j^2$. We precompute this distribution normalized to a sum of one million so that a page can be chosen with a flat random number.

The algorithm for "accessing a page" first finds the page in the lookup table and reads its heap pointer. If that pointer is non-null, we increment the frequency count for the cache entry in the heap and then reheapify the cache using backpointers to update lookup table entries for data that moves. If the heap pointer is null, we replace the root node of the heap (guaranteed by heap properties to be least-frequently accessed) with a node for the newly accessed page. In order to induce hysteresis and give pages a chance to accumulate cache hits, we perform a modified reheapification in which the new node switches place with any children that have the *same* frequency count (of one).

4.2 Methodology

Our results were obtained on a 16-processor SunFire 6800, a cache-coherent multiprocessor with 1.2Ghz UltraSPARC III processors. Our test environment was Sun's Java 1.5 beta 1 HotSpot JVM, augmented with a JSR 166 update jar file obtained from Doug Lea's web site [5]. We ran each benchmark with each of the contention management policies described in Section 3 for 10 seconds. We completed four passes of this test regime for both visible and invisible read implementations, varying the level of concurrency from 1 to 128 threads. We also repeated the tests both with and without our optimization to limit mutual abortion of transactions described in Section 2.2. Although we do not compare our results to a lock-based system, this comparison may be found in the original DSTM paper [4].

Figures 2–6 show averaged results for the Counter and LFUCache benchmarks, the read-black tree-based integer set benchmark, and the two linked list-based integer set benchmarks. Each graph is shown both in total and zoomed in on the first 16 threads (where multiprogramming does not occur). We present results only for tests with the reduced window for mutual abortion. Only one of our benchmarks (the red-black tree) is susceptible to mutual blocking, and even here the optimization does not produce a significant difference in results. On the other hand, there is also no noticeable overhead for the optimization.

4.3 Comparison Among Managers

The graphs illustrate that the choice of contention manager is crucial. For every configuration of every benchmark, the difference between a top-performing and a bottom-performing manager is at least a factor of 4, and for all but the IntSetRelease benchmark a factor of 10.

In the Counter benchmark, where every transaction conflicts with every other, the Kindergarten manager performs best. This effect can probably be attributed to the delay that is introduced when a Kindergarten manager aborts its own transaction before flipping state to abort an opposing transaction; transactions in this benchmark are short enough that the opposing transaction has a chance to complete in that window. The Timestamp manager also does well in the Counter benchmark. Here, there is no potential concurrency to be lost to serialization from the implicit queue formed by transaction start times.

In the non-release variant of the IntSet benchmark, again every transaction conflicts with every other transaction. Mirroring the Counter benchmark, the Kindergarten manager gives best performance, though by a much larger margin. The other managers perform very badly, though Karma gives some throughput at low contention levels.

For the IntSetRelease benchmark, managers separate into a few levels of performance. In the case with invisible reads, Timestamp performs badly, but the others are roughly comparable, with a slight edge to the Kindergarten manager. With visible reads, however, Karma achieves a substantial gain over all other managers tested, averaging about a factor of two in non-preemptive thread counts. Interestingly, the single worst performer is the Kindergarten manager; here, it virtually livelocks.

Greater disparity between managers can be found in the LFUCache benchmark. Before multiprogramming, with invisible reads, managers either perform well (Karma, Kindergarten, Polite, KillBlocked) or livelock at four threads (all others). With preemption, however, only Karma is able to sustain top performance; the others drop off to varying extents. With visible reads, on the other hand, there is a clear performance advantage for the Kindergarten

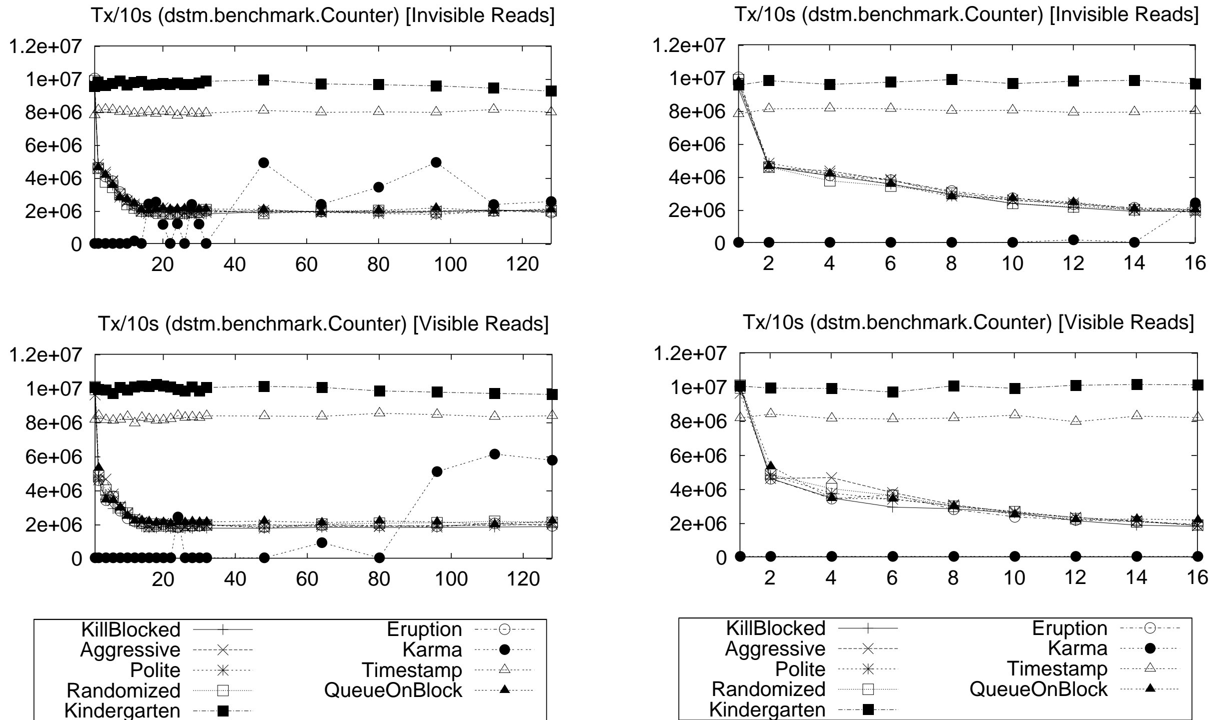


Figure 2: Counter benchmark results

manager. All others drop to low performance very quickly, although Karma does not do as poorly as the others and QueueOnBlock seems to do well at high levels of multiprogramming.

We also see much disparity in the RBTree benchmark. With visible reads, Karma outperforms all other managers by a wide margin, beginning at two threads; it is the only manager that does not virtually livelock by six threads. For invisible reads, Karma still gives top performance, but the Aggressive and Polite managers perform equally well, and QueueOnBlock is strong except in the 8–32 thread range. Interestingly, most of the remaining managers improve performance as the level of multiprogramming increases, with a plateau around 80 threads.

Across benchmarks, no single manager gives good performance in all cases. Karma and Kindergarten, however, are frequently top performers. Although overall throughput never increases with increasing numbers of threads, each benchmark has some management policy that does not degrade throughput. Of course, the limited set size we use in the benchmarks is designed to artificially increase contention, so opportunities for parallelism are limited anyway.

4.4 Visibility of Reads

In both the Counter and non-release IntSet benchmarks, there are no read accesses to blocks. As expected, we see no performance difference between visible and invisible read implementations.

In the IntSetRelease benchmark, however, there is a significant difference. While more of the managers do well with invisible reads, visible reads enable top performers to achieve almost 15 times the throughput that top performers manage with invisible reads. Middle-of-the-road managers with visible reads far outperform themselves with invisible reads. Only the Kindergarten manager does worse with visible than invisible reads.

With the RBTree benchmark, however, the situation is reversed: Karma does well with either read implementation, but all other managers perform worse, dramatically so in most cases. Similarly, in the LFUCache benchmark, managers universally do worse with visible than with invisible reads.

Why does this happen? In IntSetRelease, most reads are temporary, lasting just long enough for a thread to find the next element in the linked list; true conflict only occurs when two threads need to update the same node. Visible reads allow writers to

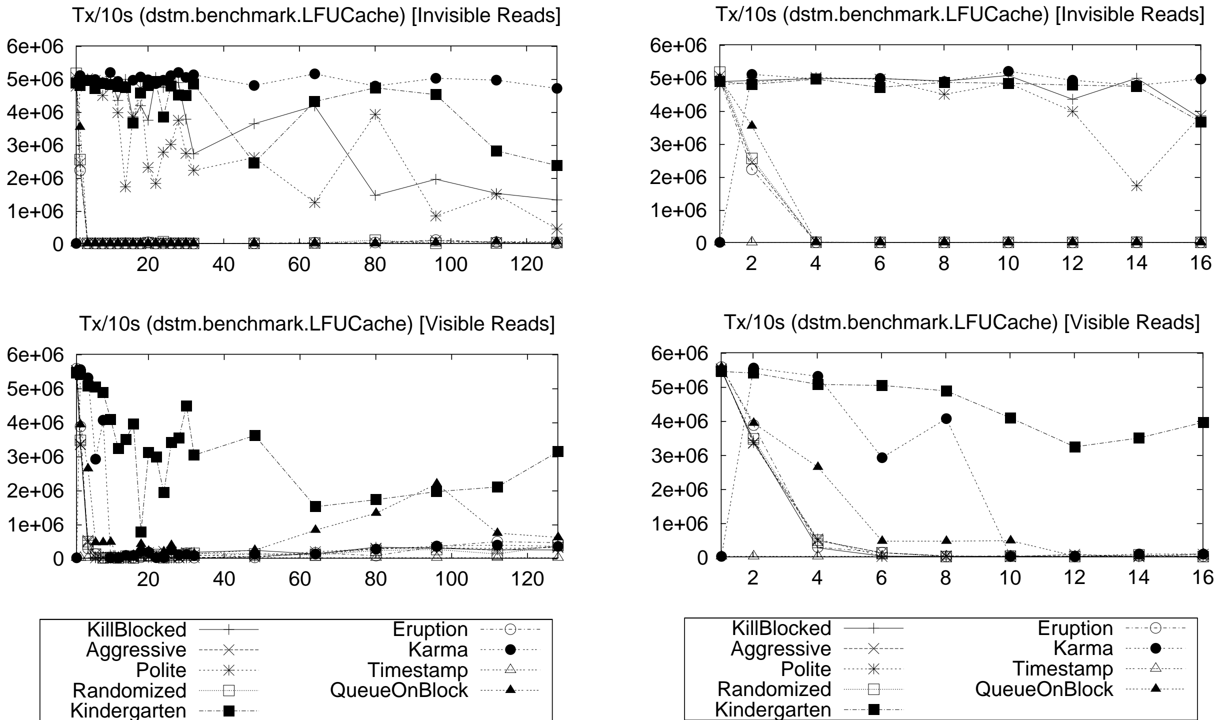


Figure 3: LFUCache benchmark results

stall and let the readers move on instead of forcing them to restart from the beginning.

In the RBTree benchmark, by comparison, conflicts between readers and writers are typically between a reading thread that is working its way from the root of the red-black tree towards an insertion/deletion point and a writing thread that is restoring the red-black tree properties upwards to the root after an insertion/deletion. If we make the reads visible, not only do the writers get delayed repeatedly (all transactions start at the tree root), but each time a writer clobbers an enemy transaction, they are likely to meet again, closer to the root. This especially explains the performance of the Kindergarten manager here: if a writer meets the same enemy twice, the other thread will "get a turn" and abort it.

5 Conclusions

In this paper we have presented a variety of contention management policies embodied in contention managers for use with dynamic software transactional memory. We have evaluated each of these managers against a variety of benchmark applications, including one novel benchmark (LFUCache) created specifically for this purpose. We have further evaluated each combination of benchmark and manager with each of two different implementations of read access in the DSTM, and with and without an optimization designed to the limit the window during which two transactions can mutually abort.

We found that different contention management policies work better for different benchmark applications, and that no single manager provides all-around best results. In fact, every manager that does well in any one benchmark does abysmally in one of the others we tested. Since the difference in throughput performance can span several orders of magnitude, gaining better understanding of when and why various policies do well is a crucial open problem.

The choice between visible and invisible reads is similarly difficult: different benchmarks perform better with different implementations. Again, further research is needed to understand when to use each type of reads. We speculate that it may be helpful to allow applications or contention managers to choose between implementations. For example, a transaction that tends to succeed almost all the time with little contention might be better served with lower-overhead invisible reads, but if it fails several times in a row, visible reads could be used to signal other transactions not to abort it.

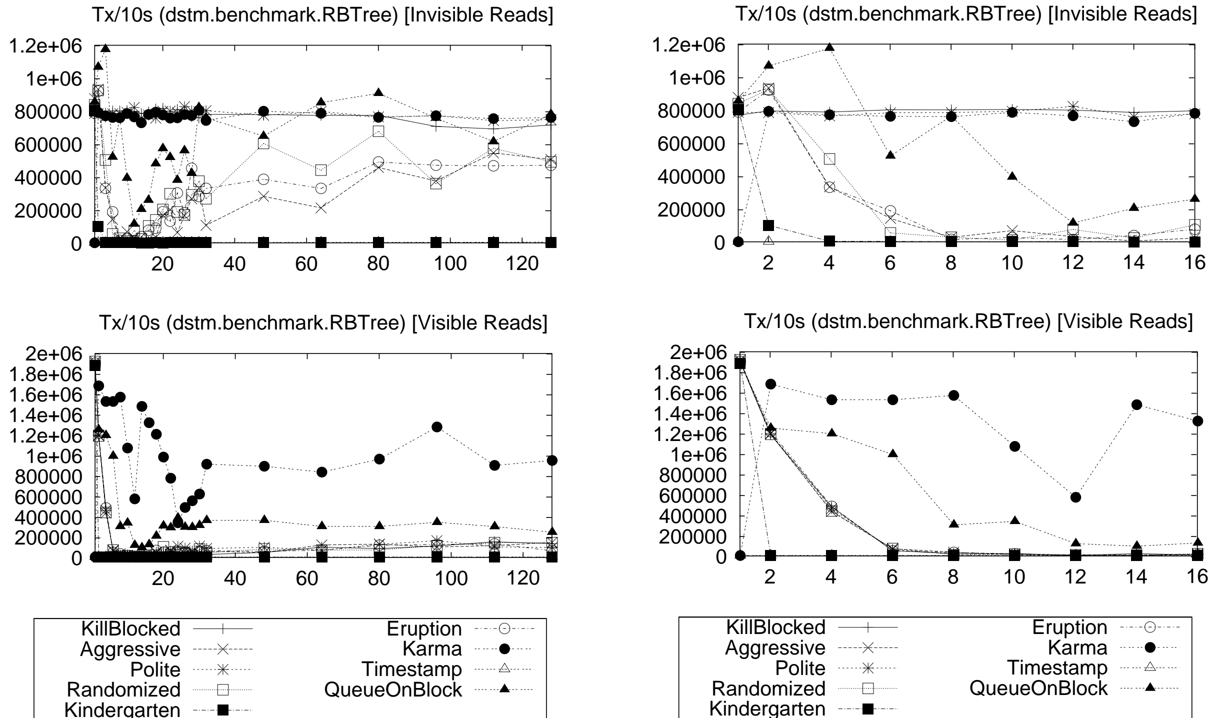


Figure 4: RBTree benchmark results

Our benchmark suite provides little opportunity to assess the value of a narrowed window for mutual aborts. Further experimentation is needed with applications in which mutual blocking may arise.

6 Acknowledgments

We are indebted to Maurice Herlihy, Victor Luchangco, and Mark Moir for various useful and productive conversations on the topic of contention management, and for providing a version of the DSTM that supports both visible and invisible reads.

References

[1] P. A. Bernstein and N. Goodman. Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems. In *Proceedings of the Sixth VLDB*, pages 285–300, Montreal, Canada, October 1980.

[2] K. M. Chandy and J. Misra. The Drinking Philosophers Problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, October 1984.

[3] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *Proceedings of the Twenty-Third International Conference on Distributed Computing Systems*, Providence, RI, May, 2003.

[4] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the Twenty-Second ACM Symposium on Principles of Distributed Computing*, pages 92–101, Boston, MA, July 2003.

[5] D. Lea. Concurrency JSR-166 Interest Site. http://gee.cs.oswego.edu/dl/concurrency-interest/.

[6] D. T. McWherter, B. Schroeder, A. Ailamaki, and M. Harchol-Balter. The Case for Preemptive Priority Scheduling in Transactional Database Workloads. Submitted to VLDB 2004.

[7] J. T. Robinson and N. V. Devarakonda. Data Cache Management Using Frequency-Based Replacement.

[8] E. Tune, D. Liang, D. M. Tullsen, and B. Calder. Dynamic Prediction of Critical Path Instructions. In *Proceedings of the Seventh International Symposium on High Performance Computer Architecture*, pages 185–196, January 2001.

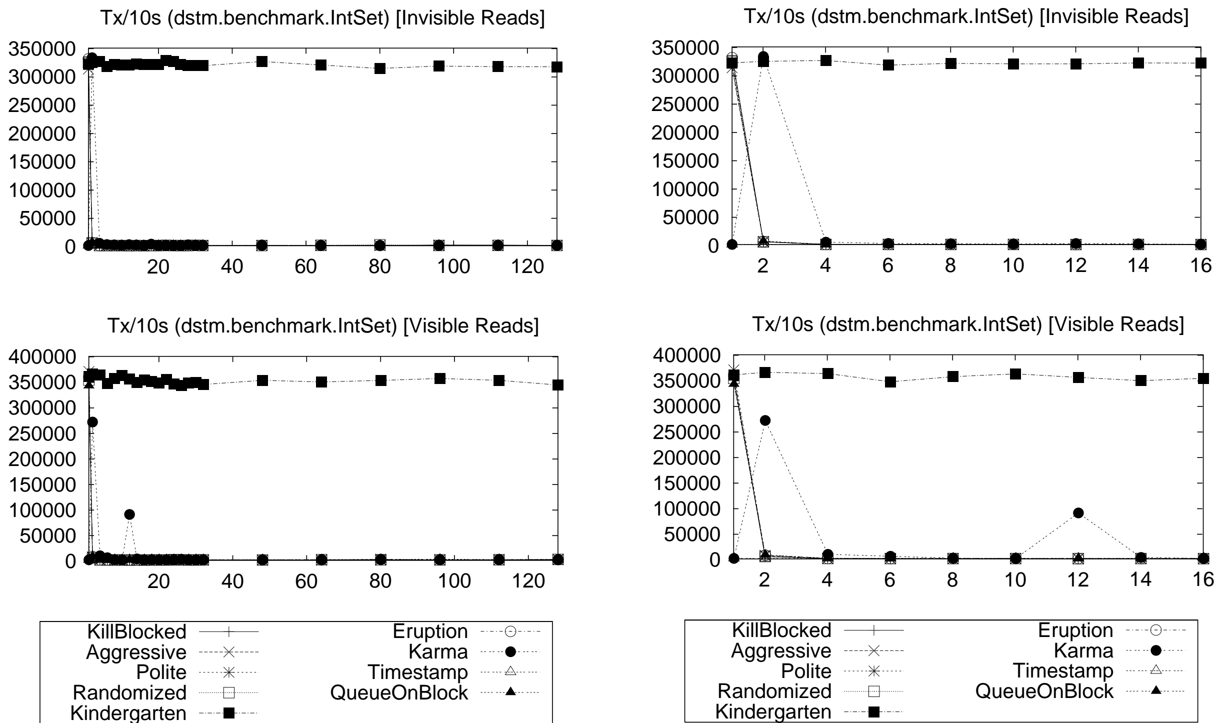


Figure 5: IntSet benchmark results

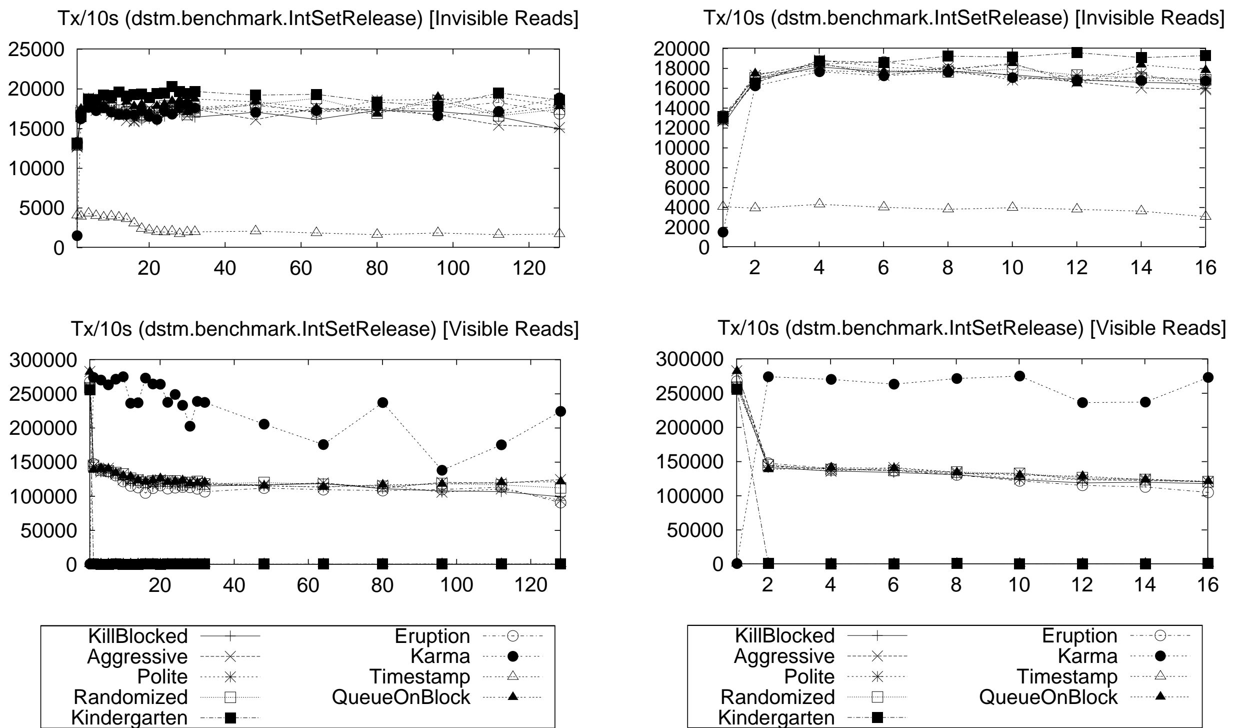


Figure 6: IntSetRelease benchmark results

Finding Concurrency Bugs In Java

David Hovemeyer and William Pugh
Dept. of Computer Science, University of Maryland
College Park, Maryland 20742 USA
{daveho,pugh}@cs.umd.edu

ABSTRACT

Because threads are a core feature of the Java language, the widespread adoption of Java has exposed a much wider audience to concurrency than previous languages have. Concurrent programs are notoriously difficult to write correctly, and many subtle bugs can result from incorrect use of threads and synchronization. Therefore, finding techniques to find concurrency bugs is an important problem.

Through development and use of an automatic static analysis tool, we have found a significant number of concurrency bugs in widely used Java applications and libraries. Interestingly, we have found that race conditions abound in concurrent Java programs; underuse of synchronization is the rule rather than the exception. We have also found many examples of other kinds of concurrency errors, suggesting that many Java programmers have fundamental misconceptions about how to write correct multithreaded programs.

This paper makes two main contributions. First, it describes simple analysis techniques that are effective at finding concurrency errors in real programs. Second, it provides evidence that threads and concurrency are widely misused in Java, even in programs written by experienced programmers.

1. INTRODUCTION

When used correctly, threads are an elegant mechanism to express concurrency and parallelism. Many applications, especially network server applications, have significant concurrency, and can benefit from parallel execution in multiprocessor systems. The Java language, by making threads a core language feature, has achieved wide popularity for writing concurrent programs.

However, threads and concurrency can be difficult to use correctly. Concurrency bugs, such as race conditions and deadlocks, can result in program misbehavior that is very difficult to diagnose. In general, reasoning about the possible behavior of a multithreaded program is difficult.

For some kinds of programs, events can be used in place of threads to avoid some of the issues that make threaded programming difficult [19]. However, events alone cannot be used to express parallelism, which is an important requirement for many applications. Several race-free and deadlock-free dialects of Java have been proposed [5, 6]; however, these dialects have not yet been widely adopted.

To better understand the kinds of concurrency bugs affecting Java programs, we have studied a variety of real applications and libraries using a static analysis tool which we developed. The tool, called FindBugs, searches for instances of *bug patterns*—code idioms that are likely to be errors. Our experience in developing the tool and using it on real software has lead us to several interesting conclusions:

1. Concurrency errors are very common, even in software widely used in production environments
2. Many programmers have fundamental misconceptions about concurrency in Java
3. Many serious bugs can be found with simple analysis techniques

The structure of this paper is as follows. In Section 2, we give an overview of our findings. In Section 3, we describe concurrency bug patterns, how our tool recognizes instances of those patterns, and how we tuned those detectors to yield more accurate results with fewer false positives. In Section 4, we present empirical data on the effectiveness of the detectors on several real Java applications and libraries, as well as anecdotal experience using the tool. In Section 5, we discuss related work. In Section 6, we present conclusions from this study, and describe possibilities for future work.

2. CONCURRENCY IN JAVA PROGRAMS

Writing correct concurrent programs is difficult [19]. The inherent nondeterminism of threaded programs, as well as the complexity of the semantics governing multithreaded execution, makes it difficult for programmers to reason about possible program behaviors. Small errors or inconsistencies, such as forgetting to obtain a lock before accessing a field, can lead to runtime errors that are very hard to debug.

Static analysis techniques have been very successful at finding many kinds of bugs in real software [16, 12, 10, 7], including errors in multithreaded programs [21, 3, 13, 11]. Bug checkers based on static analysis represent an important component of quality assurance, and can be very effective at finding potential bugs.

When we began working on techniques to find bugs in Java programs, we assumed that we would need to use so-

phisticated analysis techniques, especially in the case of concurrency bugs, where the prerequisites for a bug to manifest may involve subtle interactions between multiple threads. Instead, we found that simple analysis techniques were very effective, because all of the multithreaded software we looked at contained very obvious synchronization errors. We attribute this to two main causes.

First, many programmers, especially beginning programmers, do not understand the fundamentals of thread synchronization. We have seen many examples of code idioms, such as spin loops and race-prone condition waits, that indicate basic misconceptions about how threads work. Surprisingly, we see these idioms even in widely-used software, suggesting that these misconceptions are common even among experienced programmers.

Second, many programmers tend to view synchronization as something to be avoided whenever possible, presumably due to the assumption that synchronization is "slow". This attitude is surprising. Recent research [4, 18] has greatly reduced the overhead of locking for the uncontended case. Like any performance optimization, eliminating performance bottlenecks caused by locks must be done using careful profiling and analysis which takes the expected workload into account. Many programmers, however, appear to believe that good performance can be ensured by making the most frequent accesses to shared mutable data structures unsynchronized, even at the expense of correctness. In some cases, we even see comments indicating that the programmer was aware that he or she was writing incorrect code.

It should also be noted that our methods are not intended to be complete. There are many kinds of concurrency errors we do not attempt to detect, and our failing to detect concurrency errors should give no confidence that concurrency errors do not occur.

2.1 Why Java Concurrency is Difficult

Writing correct concurrent or multithreaded programs is exceeding difficult in any language that allows for explicit concurrency and admits the possibility of incorrect synchronization [19]. However, there are perhaps more problems in Java than in other languages, paradoxically because programmers are not as scared of writing multithreaded Java programs. All of the problems we cite also arise in other languages, such as C, C++ or C#. However, few people write multithreaded C programs unless they have carefully studied operating systems and/or concurrency, while high school students write multithreaded Java programs. Also, the consequences of incorrect programs in Java are not as severe. For example, a race condition in a Java program cannot cause the program to violate type safety, and faults arising when objects are accessed in an inconsistent state, such as null pointer dereferences and out of bound array accesses, are guaranteed to trapped and propagated as exceptions.

A more fundamental problem is that programmers often have an incorrect mental model of what behaviors are possible in concurrent Java programs. From the concurrency errors we have found in the applications and libraries we examined, it is apparent that many programmers believe that their program will execute in a sequentially consistent [1] manner. In sequential consistency, there is a global total order over all memory accesses, consistent with program order for all threads. Sequential consistency would allow some uses of data races to have useful semantics. For example, use of the double checked locking idiom [8] to avoid acquiring a lock works as expected under sequential consistency.

Unfortunately, the Java memory model [15] is not sequentially consistent. Processors and compiler optimizations may reorder memory accesses in ways that violate sequential consistency. This issue is compounded by the fact that aggressive inlining of methods, performed by most modern JVMs, can make the scope of optimizations non-local. This makes it very difficult to guess the possible behavior of code with a race condition by simply eyeballing the code.

In this paper we will show some examples of code where it is clear that the programmer had a serious misunderstanding of the semantics of concurrency in Java. Many programmers are trapped in the second order of ignorance [2] with respect to concurrent programming in Java: they do not understand how to write correct multithreaded programs, and they are not aware that they do not understand.

3. CONCURRENCY BUG PATTERNS

Our work has focused on finding simple, effective analysis techniques to find bugs in Java programs, including concurrency bugs. We start by identifying *bug patterns*, which are code idioms that are often errors. We have used many sources to find bug patterns: some have come from books and articles, while many have been suggested by other researchers and Java developers, or have been found through our own experience.

Detecting bug patterns is the automated equivalent of a code review—bug pattern detectors look for code that deviates from good practice. While to goal is to find real errors, bug pattern detectors can also produce warnings that do not correspond to real problems. Sometimes, false warnings arise because the analysis technique used by the bug pattern detector is inherently imprecise. Other times, a detector will make non-conservative assumptions about the likely behavior of the program in order to retain precision in the face of difficult program analysis problems, such as pointer aliasing or heap modeling.

Once we have identified a new bug pattern, we start with the simplest technique we can think of to detect instances of the pattern in Java code. Often, we will start out using simple state-machine driven pattern-matching techniques. We then try applying the new detector to real programs. If it produces too many false warnings, we either add heuristics, or move to a more sophisticated analysis technique (such as dataflow analysis). Our target for accuracy is that at least 50% of the warnings produced by the detector should represent genuine bugs.

We have implemented bug detectors for over 45 bug patterns, including 21 multithreaded bug patterns, in a tool called FindBugs. All of the detectors operate on Java bytecode, using the Apache Byte Code Engineering Library[1]. FindBugs is distributed under an open source license, and may be downloaded from the FindBugs website:

`http://findbugs.sourceforge.net`

This section describes some of the concurrency bug patterns implemented in FindBugs, discusses some of the implementation decisions and tradeoffs we made in writing detectors for those patterns, and presents some examples of bugs found by the tool.

[1] `http://jakarta.apache.org/bcel`

3.1 Inconsistent Synchronization

When mutable state is accessed by multiple threads, it generally needs to be protected by synchronization. A very common technique in Java is to protect the mutable fields of an object by locking on the object itself. A method may be defined with the `synchronized` keyword, in which case a lock on the receiver object is obtained for the scope of the method. Or, if finer grained synchronization is desired, a `synchronized(this)` block may be used to acquire the lock within a block scope. Classes whose instances are intended to be thread safe should generally only access shared fields while the instance lock is held. Unsynchronized field accesses often are race conditions that can lead to incorrect behavior at runtime. We refer to unsynchronized accesses in classes intended to be thread safe as *inconsistent synchronization*.

To detect inconsistent synchronization, the FindBugs tool tracks the scope of locked objects[2]. For every instance field access, the tool records whether or not a lock is held on the instance through which the field is accessed. Fields that are not consistently locked are reported as potential bugs.

We use a variety of heuristics to reduce false positives. Field accesses in object lifecycle methods, such as constructors and finalizers, are ignored, because it is unlikely that the object is visible to multiple threads in those methods. We ignore non-final public fields, on the assumption users must be responsible for guarding synchronization of such fields. Volatile fields are also ignored, because under the proposed Java memory model [15], reads and writes of volatile fields can be used to enforce visibility and ordering guarantees between threads. Similarly, final fields are ignored, since they are largely thread safe (the only exception being cases where objects are made visible to other threads before construction is complete).

Initially, we assumed that shared fields of objects intended by programmers to be thread-safe would generally be synchronized consistently, and that bugs would usually be the result of oversight by the programmer. For example, a programmer might add a public method to a thread-safe class, but forget to make the method synchronized. Under this assumption, we used the frequency of unsynchronized accesses to prioritize the warnings generated for inconsistently synchronized fields. Fields with 25% or fewer unsynchronized accesses (but at least one unsynchronized access) were assigned medium or high priority. Fields with 25-50% unsynchronized accesses were assigned low priority, on the assumption that such fields were likely to be only incidentally (not intentionally) synchronized.

To evaluate the appropriateness of our ranking heuristic, we manually categorized inconsistent synchronization warnings for several applications and libraries. We used three categories to describe the accuracy of the warnings:

Serious An accurate warning, where the unsynchronized accesses might result in incorrect behavior at runtime.

Harmless An accurate warning, where the unsynchronized accesses would be unlikely to result in incorrect behavior. An example would be an unsynchronized getter method that returns the value of an integer field.

False An inaccurate warning: either the analysis performed by the tool was incorrect, or any unsynchronized accesses would be guaranteed to behave correctly.

Our decisions were based on manual inspection of the of the code identified by each warning. While our judgment is fallible, we tried to err on the side of classifying warnings as false or harmless if we could not see a scenario that would lead to unintended behavior.

We then studied the number of serious, harmless, and false warnings that would be reported by the tool for varying cutoff values for the minimum percentage of unsynchronized field accesses. For example, for a cutoff value of 75%, only fields whose accesses were synchronized at least 75% of the time would be reported. By graphing the number of warnings in these categories, we were able to evaluate the validity of the hypothesis that most of the serious bugs would have a high percentage of synchronized accesses. We combined the 'harmless' and 'false' categories because together they represent the set of warnings we believed would not be of interest to developers. Figure 1 shows these graphs for several applications and libraries.

We were surprised to find that the likelihood of an inconsistently synchronized field being a serious bug was not strongly related to the percentage of synchronized accesses for the range of cutoff values we examined. In other words, the inconsistent synchronization bugs we found were not generally the result of the programmer simply forgetting to synchronize a particular field access or method. Instead, we found that the lack of synchronization was almost always intentional—the programmer had deliberately used a race condition to communicate between threads. The data suggests that we should try even lower cutoff values (below 50%), since many genuine bugs were found for fields synchronized only 50% of the time. The message to take away here is that lack of synchronization is not exceptional; for many classes, it is the norm.

3.1.1 Forms of Inconsistent Synchronization

As we examined examples of inconsistent synchronization, we noticed several common forms:

Synchronized field assignment, unsynchronized field use. In this form, locking is used whenever a field is set, but not when the field is read. There are two potential problems here.

First, the value read is not guaranteed to be up to date when it is eventually used; if the read is part of an operation with multiple steps, the operation will not be guaranteed to take place atomically. A more subtle problem is that if a reference is read from a field without synchronization, there is generally no guarantee that the object will be completely initialized.

This was the most common form of inconsistent synchronization. An example is shown in Figure 2.

Object pair operation. In this form, an operation is performed involving two objects, each of which can be accessed by multiple threads. However, a lock is acquired on only one of the objects, allowing the other to be vulnerable to concurrent modification. This problem is especially prevalent in `equals()` methods. It can be quite hard to fix because of the potential deadlock issues: if two threads try to lock both objects, but use different lock acquisition orders, a deadlock can result.

[2]The analysis is intraprocedural, with the addition that calls to non-public methods within a class are analyzed, and non-public methods called only from locked contexts are considered to be synchronized as well.

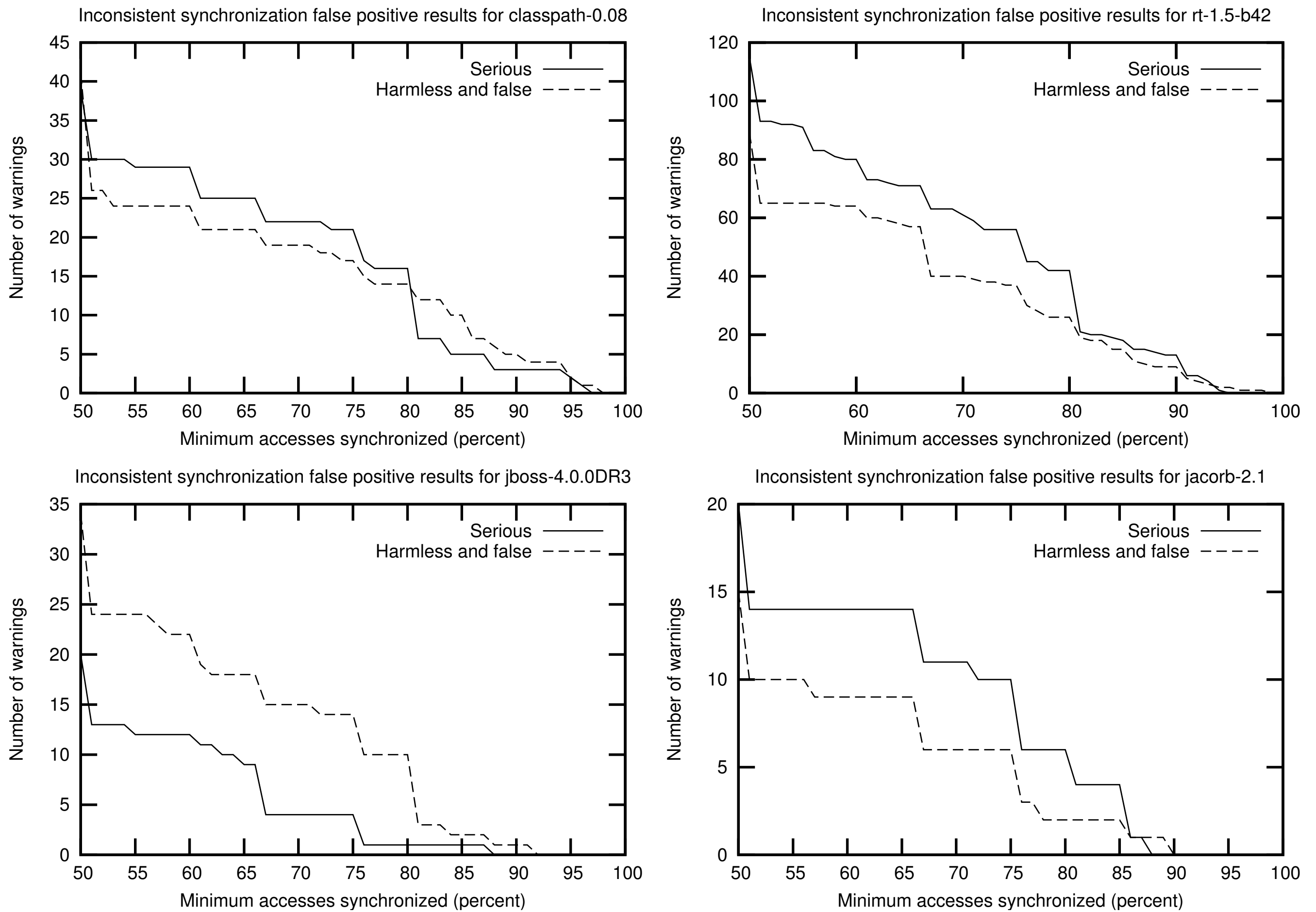


Figure 1: Serious bugs and false and harmless warnings for varying values for minimum percentage of synchronized accesses. The applications are two implementations of the J2SE core libraries (rt-1.5-b42 and classpath-0.08), a open source Java application server (jboss-4.0.0DR3), and an open source CORBA ORB (jacorb-2.1). Our hypothesis that fields with lower (but nonzero) percentages of unsynchronized accesses would be more likely to be errors was found to be incorrect.

```
// java.lang, StringBuffer.java, line 825

public int lastIndexOf(String str)
{
  return lastIndexOf(str, count - str.count);
}
```

Figure 2: An atomicity bug in GNU Classpath 0.08. The `count` field is read without synchronization and then passed to a synchronized method. Because the value may be out of date, an `ArrayIndexOutOfBounds` exception can result.

3.1.2 Limitations of the Inconsistent Synchronization Detector

While it is effective at finding many concurrency bugs, the inconsistent synchronization detector has some important limitations. First, programs that are free of race conditions may still have atomicity bugs. A naïve approach to synchronization is to make every method of a class synchronized. However, successive calls to methods in such a class will not occur atomically unless a lock on the receiver object is explicitly held in a scope surrounding both calls. Another limitation is that the detector only works when shared fields are synchronized by locking the object instance. Although this is a common technique in Java, it is also common to use explicit lock objects. Modifying the detector to handle arbitrary lock objects would require more sophisticated analysis, including some form of heap analysis.

3.2 Double Checked Locking

Lazy initialization is a common performance optimization used to create singleton objects only as they are needed. In a multithreaded program, some form of synchronization is needed to ensure that the singleton object is created only once, and that the object is always fully initialized before it is used.

A common idiom for lazy initialization of singleton objects is *double checked locking.* Synchronization is performed only if the object has not yet been created:

```
static SomeClass field;

static SomeClass createSingleton() {
  if (field == null) {
    synchronized (lock) {
      if (field == null) {
        SomeClass obj = new SomeClass();
        // ...initialize obj...
        field = obj;
      }
    }
  }
  return field;
}
```

The intent of double checked locking is that the overhead of lock acquisition is only incurred if the singleton object is observed as not having been created yet.

Unfortunately, this form of double checked locking is not correct. Although the idiom guarantees that the object is created only once, the Java memory model does not guarantee that the threads that see a non-null field value (but do not acquire the lock) will see all of the writes used to initialize the object. For example, the JIT compiler may inline the call to the constructor and reorder some of the writes initializing the object so that they occur after the write to the field storing the object instance.

Double checked locking is a good illustration of the gulf between how multithreaded code looks like it should behave and the behaviors actually allowed by the language. Most programmers can easily understand how synchronization can be used to guarantee the atomicity of a sequence of operations. However, it is much harder to understand the subtle interaction of compiler and processor optimizations. Memory model issues are challenging even for experts: double checked locking has been advocated in a number of books and articles (as mentioned in [8]), showing that even experts do not always understand the consequences of omitting proper synchronization.

Under the proposed Java Memory Model [15], it is possible to fix instances of double checked locking by making the field volatile. The volatile qualifier causes the compiler to insert the necessary optimization and memory barriers needed to ensure that all threads will see a completely initialized object, even if they don't acquire the lock. However, volatiles must be used with caution; almost all programmers should forego the use of volatiles in favor of using locking.

Our detector for this bug pattern looks for sequences of bytecode containing an `ifnull` instruction, followed by a `monitorenter` instruction, followed by another `ifnull`. This implementation catches many instances of double checked locking, with a low false positive rate. We have experimented with more sophisticated detectors for this pattern, but we found that they were not significantly more accurate.

An example of incorrect double checked locking found in JBoss-4.0.0DR3 is shown in Figure 3.

3.3 Wait Not In Loop

Java monitors support notify and wait operations to allow threads to wait for a condition related to the state of a shared object. For example, in a multithreaded blocking queue class, the `dequeue()` method would wait for the queue to become nonempty, and the `enqueue()` method would perform a notify to wake up any threads waiting for the queue to become nonempty.

Often, a single Java monitor is used for multiple conditions. For such classes, the correct idiom is to surround the call to wait with a loop which repeatedly checks the condition. Without the loop, the thread would not know whether the condition is actually true when the call to wait returns.

The condition can fail to be true for several reasons:

- The monitor is being used for multiple conditions, so the condition set by the notifying thread may not be the one the waiting thread is waiting for.
- In between the notification and the return from wait, another thread obtains the lock and changes the condition. For example, a thread might be waiting for a queue to become non-empty. A thread inserts a new element into the queue and notifies the waiting thread. Before the waiting thread acquires the lock, another thread removes the element from the queue.
- The specification for the wait method allows it to spuriously return for no reason. This can arise due to

```
// org.jboss.net.axis.server, JBossAuthenticationHandler.java,
// line 178

public void invoke(MessageContext msgContext) throws AxisFault {
   // double check does not work on multiple processors, unfortunately
   if (!isInitialised) {
      synchronized (this) {
         if (!isInitialised) {
            initialise();
         }
      }
   }
   ...
```

Figure 3: A double checked locking bug in JBoss-4.0.0DR3. Not only does the comment indicate that the programmer was aware the idiom is incorrect, the `initialise()` method writes a true value to the `isInitialised` field before the object has been completely initialized, meaning that the code would not be correct on a *single* processor system, even if the Java memory model were sequentially consistent.

```
if (!book.isReady()) {
  DebugInfo.println("book not ready");

  synchronized (book) {
    DebugInfo.println("waiting for book");
    book.wait();
  }
  ...
```

Figure 4: An unconditional wait bug in an early version of the International Children's Digital Library. If the book becomes ready after `isReady()` is called and before the lock is acquired, the notification could be missed and the thread could block forever.

special handling needed for the interaction of interrupts and waiting, and because underlying operating system synchronization primitives used by the JVM, such as pthreads, allow spurious wakeups.

The detector for this bug pattern examines bytecode for a call to `wait()` which is not in close proximity to the target of a backwards branch (i.e., a loop head).

3.4 Unconditional Wait

The Unconditional Wait pattern is a special case of Wait Not In Loop. In this bug pattern, a wait is performed immediately (and unconditionally) upon entering a synchronized block. Often, this indicates that the programmer did not include the test for the waited-for condition as part of the scope of the lock, which could lead to a missed notification. An example of this bug pattern is shown in Figure 4.

3.5 Other Concurrency Bug Patterns

This section presents several other concurrency bug patterns. These detectors are generally not effective at finding bugs in mature software. However, they are often useful in finding bugs in code written by inexperienced programmers.

Mismatched Wait and Notify. In this bug pattern, the programmer calls a wait or notify method without holding a lock on the monitor object. This will result in an `IllegalMonitorStateException` being thrown. Our implementation intraprocedurally tracks the scopes of locks, and emit a warning when a call to a wait or notify method is seen without a lock being held on the receiver object.

Two Locks Held While Waiting. Signaling between threads is often handled by using `wait()` and `notify()`/ `notifyAll()`. When `wait()` is invoked, the thread invoking wait must hold a lock on the object on which wait is invoked, and all locks on that object are released while the thread is waiting. However, locks on other objects are not released. This can cause poor performance, and can cause deadlock if they thread that is trying to perform a notify needs to acquire that lock.

This detector performs an intraprocedural analysis to find the scope of locks, and emits a warning whenever a method holds multiple locks when wait is invoked.

Calling notify() instead of notifyAll(). When a shared object is visible to multiple threads, it is possible for more than one thread to wait on the object's monitor. When multiple conditions are associated with a single monitor, the threads waiting on the monitor may be waiting for more than one distinct condition. Because Java makes no guarantee about which thread is woken when the `notify()` method is called, it is good practice to use `notifyAll()` instead, so that all threads are woken. Otherwise, the thread woken might not be the one waiting for the condition that the notifier has made true.

Invoking Thread.run() instead of Thread.start(). The `run()` method of classes extending the `Thread` class defines the code to be executed as the body of a new thread. Inexperienced programmers sometimes call the `run()` method directly, not realizing that it will not actually start the thread.

Mutable lock. In order for lock objects to be used correctly by multiple threads, they need to be created before the threads that use them for synchronization. Therefore, it is suspicious when a method performs synchronization on an object loaded from a field that was assigned earlier in the method.

Naked Notify. In general, calling a notify method on a monitor is done because some condition another thread is waiting for has become true. To correctly notify the waiting thread (or threads) of a change in the condition, the noti-

```
// get next items from queue if any
while (listlock) {
    ;
}
listlock = true;
```

Figure 5: A spin loop in an early version of the International Children's Digital Library.

fying thread must make an update to shared state before performing the notification. The detector for this bug pattern looks for calls to notify methods which do not appear to be associated with earlier updates to shared mutable state.

Because calls to notify methods always increase the liveness of threads in a program, instances of this pattern are not always genuine bugs. However, misunderstanding the correct use of wait and notify is common for inexperienced programmers, and this detector helps inform them when their code deviates from good practice.

Spin Wait. In this bug pattern, a method executes a loop which reads a non-volatile field until an expected value is seen. Aside from the obvious waste of CPU time, there is a more subtle issue. If the field is never assigned in the body of the loop, the compiler may legally hoist the read out of the loop entirely, resulting in an infinite loop. An example of a spin loop is shown in Figure 5.

4. EVALUATION

Because the FindBugs tool uses heuristics, it may report warnings that are not real errors. In order to evaluate the accuracy of the warnings produced by the tool, we applied it to several applications and libraries:

- rt.jar, from Sun JDK 1.5 beta build 42: this is Sun's implementation of the core J2SE libraries
- classpath-0.08: an open source implementation of a subset of the core J2SE libraries from the GNU project
- jboss-4.0.0DR3: a popular open source application server for Enterprise Java Beans
- JacORB-2.1: an open source implementation of a CORBA Object Request Broker (ORB)
- International Children's Digital Library 0.1: an early version of a web-based digital library application from the University of Maryland Human-Computer Interaction Laboratory [9]

With the exception of ICDL, These are mature, production quality applications and libraries. We manually classified each warning as serious (a real bug), harmless (a bug unlikely to cause incorrect behavior), and false (an inaccurate warning). The results are shown in Table 1.

In evaluating the accuracy of the detectors, we tried to err on the side of not marking a warning as serious unless we felt confident that it could result in undesirable behavior at runtime. For example, we marked some of the 'Wait not in loop' warnings as harmless because they were used to implement an infinite wait, so liveness was not an issue.[3]

In general, the detectors achieved our target of no more than 50% false and harmless warnings. Some detectors were very accurate: for example, the warnings generated by the Double Checked Locking detector were almost always accurate. The Inconsistent Synchronization detector was somewhat less accurate, although still within the acceptable range, especially considering that our tool operates without explicit specifications of which classes and methods are intended to be thread safe. The Unconditional Wait and Wait Not In Loop detectors were less accurate than desired. However, they produce only a small number of warnings, and genuine instances of these bug patterns tend to be critical bugs that can be very hard to debug.

4.1 Anecdotal Experience

This section describes some of our experience in applying the FindBugs tool.

One of the applications we studied as we were developing FindBugs was an early version of the International Children's Digital Library [9]. In conversations with the authors, we found out that they had spent several months tracking down a threading bug. When we applied FindBugs to the buggy version, it immediately found the problem. A similar problem in a different version of the ICDL software is shown in Figure 4.

4.1.1 Reporting Bugs Found by FindBugs

We have submitted some of the most serious bugs found by our tool in the Java core libraries to Sun's Java bug database.

Using our detector for Two Lock Wait, we found a serious potential deadlock in the `com.sun.corba.se.impl.orb.ORBImpl` class of Sun's JDK 1.5 build 32. In the `get_next_response()` method, two locks are held when wait is called. Only one of these locks is released while the threads is waiting. The thread can be notified by calling the `notifyORB()` method. Unfortunately, before the notification can be performed, the thread must obtain the lock still held by the thread awaiting notification, resulting is deadlock. We reported the problem to Sun, it was confirmed to be a bug, fixed internally, and the fix is scheduled to be part of a future beta release.

In prerelese versions of Sun's JDK 1.4.2, we found serious inconsistent synchronization bugs in the `append(boolean)` method of `StringBuffer` and the `removeRange(int,int)` method of `Vector`. Both classes are meant to be thread-safe, and these methods were left unsynchronized, resulting in exploitable race conditions. Even though these were acknowledged to be genuine bugs by sources at Sun, the `removeRange` error will only be fixed in the 1.5 branch, not in the 1.4 branch of Java. This illustrates an interesting asymmetry about the software engineering issues surrounding concurrency bugs in commercial software:

- It is easy to introduce concurrency errors in new code
- It is difficult to fix these bugs once they are introduced

The reason for the asymmetry is that maintenance engineers are understandably reluctant to fix bugs which cannot be easily reproduced with a simple test case.[4] Also, mainte-

[3]Interestingly, spurious wakeups will be allowed in the revised Java memory model[15], meaning that an unconditional call to `wait()` does not correctly implement an infinite wait.

[4]An interesting problem here is that many concurrency errors *cannot* be reliably reproduced by a test case.

	rt-1.5-b42				classpath-0.08			
	warnings	serious	harmless	false pos	warnings	serious	harmless	false pos
Double check	78	92%	0%	7%	0	—	—	—
Lazy static initialization	146	100%	0%	0%	10	100%	0 %	0 %
Double check	78	92%	0%	7%	0	—	—	—
Inconsistent sync	204	56%	31%	11%	80	48%	30%	21%
Mutable lock	1	100%	0%	0%	0	—	—	—
Running, not starting a thread	1	0%	0%	100%	1	0%	0%	100%
Unconditional wait	4	0%	25%	75%	2	0%	0%	100%
Wait not in loop	6	0%	16%	83%	3	0%	0%	100%

	jboss-4.0.0DR3				jacorb-2.1			
	warnings	serious	harmless	false pos	warnings	serious	harmless	false pos
Double check	5	80%	0%	20%	1	100%	0%	0%
Lazy static initialization	643	100%	0%	0%	579	100%	0 %	0 %
Inconsistent sync	54	37%	24%	38%	35	57%	17%	25%
Unconditional wait	3	66%	0%	33%	5	60%	0%	40%
Wait not in loop	4	0%	0%	100%	5	20%	0%	80%

	icdl			
	warnings	serious	harmless	false pos
Lazy static initialization	10	100%	0%	0%
Inconsistent sync	3	100%	0%	0%
Spin Wait	4	100%	0%	0%
Unconditional wait	3	66%	0%	33%
Wait not in loop	3	100%	0%	0%

Table 1: False positive rates for concurrent bug pattern detectors.

nance engineers must be concerned as to whether introducing missing synchronization could possibly introduce deadlock. Because concurrency errors are almost always difficult to reproduce, they can linger for a long time without being fixed. This highlights the usefulness of running static tools to catch bugs *before* they are introduced into a deployed code base.

4.1.2 Finding Bugs in Student Projects

Programmers with different skill levels tend to make different kinds of mistakes. We found that bug detectors such as Wait Not In Loop and Naked Notify did not find many serious bugs in production quality software. However, these detectors were very effective at finding bugs in projects written by students in an undergraduate advanced Java programming course; for many of the students, the course is their first significant exposure to concurrent programming.

Table 2 shows, for a programming project (assigned when we taught the course in Spring 2001) where students used threads for the first time, the number of student projects for which FindBugs reported various kinds of concurrency warnings. Many of these bugs are ones that we would not expect to see in production code. For example, Mismatched Wait/Notify bugs manifest by throwing a runtime exception (specifically, `IllegalMonitorStateException`), which should be diagnosed and fixed during testing. However, since students do not always understand the meaning of these exceptions, they may be tempted to ignore them, or even write handlers for them. Because students do not understand threads well, these warnings typically indicate serious errors in their code.

In recent semesters when we have taught the same course,

Bug Type	Number of Students
Inconsistent synchronization	7
Mutable Lock	1
Mismatched Wait/Notify	4
Naked Notify	4
Notify instead of NotifyAll	4
Running, not starting a thread	1
Unconditional Wait	5
Any of the above	19

Table 2: Number of student projects in an undergraduate advanced Java programming course for which FindBugs generated various kinds of concurrency warnings.

we have given students access to both FindBugs and a dynamic data race detection tool, both of which have been successful at helping students find and fix concurrency problems in their projects. However, giving students access to these tools has also made it more difficult for us evaluate the effectiveness of FindBugs by applying it to submitted programming assignments. We are working to develop infrastructure that will allow us to record the effectiveness of FindBugs as students develop code. Ensuring that FindBugs reports problems in a way that enhances students' understanding of concurrency issues is something we are actively pursuing.

5. RELATED WORK

Static bug checkers have a long history. The original pro-

gram checker is Lint [16], which uses simple analyses to find common errors in C programs. LCLint [12] is similar in spirit to the original Lint, with the addition of checking code for consistency with specifications supplied by the programmer. PREfix [7] symbolically executes C and C++ programs to find a variety of dynamic errors, such as memory corruption and out of bound array accesses. MC (for "metacompilation") [10] uses a sophisticated interprocedural analysis to check code over large numbers of paths through an entire system; state machines driven by program statements are used to check correctness properties on those paths. MC uses a novel language, called Metal, to encode the state machines, allowing checks for new properties to be added easily. The SABER project at IBM[14] uses an approach very similar to FindBugs in order to find errors in J2EE applications.

Many static bug checkers have focused on finding concurrency errors in software. Warlock [21] checks variables in multithreaded C programs to determine if they are protected by a consistent set of locks; accesses to variables with an inconsistent lockset are flagged as potential race conditions. JLint [3] performs an interprocedural analysis on Java programs to find potential deadlocks and race conditions. In [13], Flanagan and Qadeer describe a static analysis to find methods that are not atomic; as noted earlier, programs free of race conditions can still have atomicity bugs. RacerX [11] is system for finding race conditions and deadlocks in C programs. Its analysis is very similar to that performed by MC; however, some new analysis techniques are introduced, including "unlockset" analysis, which can be thought of as lockset analysis backward in time. Using unlocksets can increase the confidence of reports for unsynchronized field accesses over using locksets alone.

A variety of dynamic techniques and tools have been developed to help find and diagnose concurrency errors. Eraser [20] dynamically computes the set of locks held during accesses to shared data. Accesses to the same location with inconsistent lock sets are potential bugs. JProbe [17] dynamically monitors a running Java program to detect race conditions and deadlocks.

6. CONCLUSIONS

From our studies of concurrency bugs, we conclude that many programmers have fundamental misconceptions about how to write correct programs using threads. The intuition many programmers have about how multithreaded programs ought to work is flawed. Some of this can be attributed to inaccurate information (such as the books and articles advocating double checked locking). Some can be attributed to inadequate educations—threads and concurrency are generally considered only briefly in the undergraduate Computer Science curriculum, with more in-depth treatment coming only in electives. Finally, modern multiprocessor architecture and aggressive optimizing compilers can lead to surprising and subtle behaviors in multithreaded programs.

We believe that static checking tools can aid programmers in two important ways. First, they can help find bugs in software. Second, and perhaps more importantly, they can help educate programmers about error-prone idioms arising from misconceptions about threads and concurrency.

In future work, we would like to develop detectors for other kinds of concurrency errors, and continue to improve the accuracy of the existing detectors. We would also like to evaluate the extent to which static tools can help inexperience programmers learn to use threads correctly.

7. ACKNOWLEDGMENTS

Some of the bug detectors implemented in FindBugs were suggested by Doug Lea and Josh Bloch. We would like to thank Jeremy Manson and Jaime Spacco for helpful feedback on this paper. Finally, we would like to thank the anonymous reviewers for their insights and suggestions.

8. REFERENCES

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

[2] Phillip G. Armour. The five orders of ignorance. *Commun. ACM*, 43(10):17–20, 2000.

[3] Cyrille Artho and Armin Biere. Applying static analysis to large-scale, multi-threaded Java. In *Proceedings of the 13th Australian Software Engineering Conference*, pages 68–75, August 2001.

[4] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: featherweight synchronization for Java. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 258–268. ACM Press, 1998.

[5] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: a dialect of Java without data races. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 382–400. ACM Press, 2000.

[6] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 211–230. ACM Press, 2002.

[7] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A Static Analyzer for Finding Dynamic Programming Errors. *Software—Practice and Experience*, 30:775–802, 2000.

[8] The "double-checked locking is broken" declaration. http://www.cs.umd.edu/users/pugh/java/memoryModel/DoubleCheckedLocking.html.

[9] A. Druin, Ben Bederson, A. Weeks, A. Farber, J. Grosjean, M.L. Guha, J.P. Hourcade, J. Lee, S. Liao, K. Reuter, A. Rose, Y. Takayama, L., and L Zhang. The international children's digital library: Description and analysis of first use. Technical Report HCIL-2003-02, Human-Computer Interaction Lab, Univ. of Maryland, January 2003.

[10] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation, San Diego, CA*, October 2000.

[11] Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 237–252, 2003.

[12] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A tool for using

specifications to check code. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 87–96, 1994.

[13] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 338–349, 2003.

[14] J2EE Code Validation Preview for WebSphere Studio. http://www-106.ibm.com/developerworks/websphere/downloads/j2ee_code_validation.html.

[15] Java Specification Request (JSR) 133. Java memory model and thread specification revision, 2004. http://jcp.org/jsr/detail/133.jsp.

[16] S. Johnson. Lint, a C Program Checker, Unix Programmer's Manual, AT&T Bell Laboratories, 1978.

[17] Quest Software — JProbe Threadalyzer. http://www.quest.com/threadalyzer.jsp.

[18] Kiyokuni Kawachiya, Akira Koseki, and Tamiya Onodera. Lock reservation: Java locks can mostly do without atomic operations. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 130–141. ACM Press, 2002.

[19] John K. Ousterhout. Why threads are a bad idea (for most purposes). Invited talk, USENIX 1996 Technical Conference.

[20] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[21] Nicholas Sterling. WARLOCK: A Static Data Race Analysis Tool. In *Proceedings of the USENIX Winter Technical Conference*, pages 97–106, January 1993.

Observations on the Assured Evolution of Concurrent Java Programs

Aaron Greenhouse
Software Engineering Institute
Carnegie Mellon University
4500 Fifth Avenue
Pittsburgh, PA 15213
aarong@sei.cmu.edu

T. J. Halloran
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
thallora@cs.cmu.edu

William L. Scherlis
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
wls@cs.cmu.edu

ABSTRACT

Evolving and refactoring concurrent Java software can be error-prone, resulting in race conditions and other concurrency difficulties. We suggest that there are two principal causes: Concurrency design intent is often not explicit in code and, additionally, consistency of intent and code cannot easily be established through either testing or inspection.

We explore several aspects of this issue in this paper. First, we describe a tool-assisted approach to modeling and assurance for concurrent programs. Second, we give an account of recent case-study experience on larger-scale production Java systems. Third, we suggest an approach to scalable co-evolution of code and models that is designed to support working programmers without special training or incentives. Fourth, we propose some concurrency-related refactorings that, with suitable analysis and tool support, can potentially offer assurances of soundness.

1. INTRODUCTION

Reasoning about concurrent Java programs is a challenge for both programmers and software tool designers. Evolving concurrent programs can be even a greater challenge. Developers must constantly raise and address questions such as: *What data is shared by multiple threads? Is it accessed safely? What locks should be held when particular portions of shared state are accessed? Whose responsibility is it to acquire the lock? Is this delegate object uniquely referenced by its referring object?*

Two questions. These bread-and-butter development questions are instances of two kinds of general questions:

The first question is what is the concurrency-related design intent. An example of design intent is the association of lock objects with particular regions of shared state.

Developers rely on conventions, for example, that an object is used as a lock for its own state, and takes care of its own locking. These conventions are, however, not universally applicable and must sometimes be breached. For example, the lock of a pool object may protect the state of the pool object itself plus just those portions of the state of *pooled* objects that represents the backbone of the linked list of the objects in the pool. Another example is the use of a lock of a referring object to protect the state of uniquely referenced delegate objects, particularly arrays. This is typical in event queue representations. Another issue is who has responsibility for lock acquisition. There are many cases where a client of a shared object is meant to be responsible for acquiring and releasing locks—this is particularly useful when a lock must be held over a sequence of method calls.

Another example of design intent is the decision, typical in GUI frameworks, to follow a single-thread policy to protect the integrity of shared data. APIs often have threading policies associated with them: the Java AWT, for example, has certain rules regarding the appropriate use of its event thread to support non-lock concurrency. These rules include restrictions on which threads may invoke callback method definitions.

The second question is whether there is consistency between code and the stated design intent. An example of consistency for a lock-based model is an assurance that the correct lock is *always* acquired prior to accessing shared state [16, 13]. For non-lock concurrency, the assurance pertains to the identity of thread that touch critical state [22]. Failures to achieve consistency thus reflect either flaws in the model, or flaws in the code, or inadequacy of the verification and analysis tools.

Adoptability and scalability. Answering these questions, particularly in the case of fast-paced iterative developments, is highly problematic. While programmers may communicate design intent informally among themselves, there is relatively little success in capturing intent in a sufficiently precise representation that tools can be used to assist in assessing consistency. The high "expression cost" creates an effective adoption barrier, and the well known difficulties in general-purpose assurance for larger systems create an effective scaling barrier.

These perceptions are generally well founded: both the expression of model information and the assurance of its consistency with the code present an overwhelming barrier to entry for routine Java programming and any kind of formal assurance approach is suitable only for critical systems at small scale. There has been recent experience, however, in both modeling and assurance that suggests the possibility that both of these barriers can be removed in a wide range of interesting cases [2].

We present here an approach to concurrency-related assurance and evolution that is designed with these as the principal challenges. We do this by titrating back on inferential and expressive power to achieve goals related to adoptability by working programmers and scalability to subsystems of realistic size.

In this paper, we present our approach to modeling and

reasoning about Java concurrency. We identify the specific limits we accept on expressiveness and power to achieve this. For example, rather than requiring a programmer to express and verify full representation invariants for data structures, we instead substitute a model of "guilt-by-association," in which those constituents of shared state are merely associated with each other into "regions" [16, 14]. Race conditions, by definition, occur when a representation invariant is expected to hold at a place where it in fact does not due to the interleaved execution of a separate thread. In our approach, we focus on identifying the constituents of state that might be related by a putative invariant, but not elucidating their relationship. The hypothesis underlying this approach is that modeling at this level of abstraction is sufficient for the purposes of establishing safe concurrency. This has been reinforced through extensive case study experience.

Java concurrency. One of the challenges in modeling and reasoning about concurrency is that most interesting concurrency-related properties defy both traditional testing and inspection techniques. There is no single place in the code to look to find either expression of model information or evidence of compliance. In particular, the reasoning process to ensure consistency of code and model is almost always non-local in the structure of the code. This raises a challenge for programmers attempting to accomplish informal reverse engineering—often of their own code—and also for tool developers seeking to assist in evolving and assuring concurrent Java programs. Assuring programs, in this case, means establishing consistency between the code and concurrency design intent, as noted above. This implies that the model of intent must somehow be expressed.

Approach and prior work. This paper addresses these challenges. Our approach has four elements: (1) The incremental expression of "mechanical" design intent (what some call "non-functional" requirements) for Java concurrency, (2) An incremental and composable approach to analysis-based assurance of consistency of that intent with code, and (3) Support for rapid iteration in the co-evolution of code and design intent, including (4) Support for semi-automated refactorings.

Underlying this approach is a tool based on a suite of composable semantic program analyses supported by a complex assertion- and proof-management scheme that supports composition and incrementality.

We have previously described some results that contribute to elements (1) and (2) of our approach above—expressing design intent that enables programmers to capture model information in a way that enables assured consistency between the expressed design intent and code [7, 6, 14, 16, 22, 13]. In this paper we summarize recent experience regarding these first two elements, and offer some potential directions for addressing elements (3) and (4). This builds on extensive case study experience using our Eclipse-based tool.

Practicability. Our overall approach is designed from the outset to be practicable—feasibly adoptable by real programmers working on deadline. One of the lessons we have learned is that considerations of practicability have a significant influence on every aspect of the approach, including even the design of underlying analysis algorithms—for example, to support component composition, explicit cutpoints, programmer debugging, interactions among analyses, *etc.*

There is also a perception, which is also generally well founded, that assurance raises a formidable barrier to evolution of code/models. This includes manual evolution of code and models as well as the means by which their consistency can be managed. It is important to recognize that in production development efforts consistency is almost always partial, as are the associated models. That is, programmers must therefore be able, incrementally, to accrete model information, to reason about models and code, and to evolve both models and code. This evolution can be entirely manual or it can be tool assisted, as in the semi-automated refactorings implemented in tools such as Eclipse.

Our approach to practicability is based on three key principles:

1. *Incrementality and early gratification.* Any increment of effort we ask programmers to undertake should yield a generally immediate reward in the form of bug finding, assurance creation, guidance in evolution, or model expression. The intent is that useful assurances can be obtained with minimal or no annotation effort, and additional increments of annotation yield additional increments of assurance. This is one of the reasons why we have avoided any requirement for explicit expression of representation invariants.
2. *Familiar expression.* Properties should be expressed tersely and using terminology already familiar to programmers. We cannot require programmers on deadline to become expert in unfamiliar verification formalisms. This is a challenge, because representation invariants underly the semantic distinction between races and desired concurrency. Generally, our more abstract proxies have proven to be sufficient.
3. *Cut points and composability.* We should be able to handle individual components separately from each other, developing composable assurances, which can be linked together to form "chains of evidence" supporting an overall system-level claim. There are two challenges: First, can cutpoints be expressed in a way that satisfies the first two principles above. And, second, can the analyses be accomplished without excessive conservatism—that is, can useful analysis results be obtained for a broad range of existing Java code.

Refactoring. Rapid development iteration is increasingly a tool-supported activity, with tools used to assist in restructuring or *refactoring* of code. Refactorings are systematic plans for transforming source code, generally in ways that preserve behavior. For example, one kind of refactoring is the extraction of a new abstract superclass from two similar classes—without changing the behavior of the program. This refactoring can eliminate redundancy and improve ability to understand and evolve the program. Another example of a refactoring is extraction of a new method definition from one or more existing method definitions. This can involve subtle reorderings of computation, for example, of the code "left behind" at the call site to calculate values of actual parameters.

Refactoring is a challenge from the standpoint of bug-prevention and program assurance. If one starts with a correct (assured) program with respect to a set of models, then the refactored program should still be correct after applying the refactoring. This is particularly important when refactoring results in broad structural changes. This means that refactorings may have associated soundness preconditions.

A use of refactoring is to make programs easier to under-

stand by making intent more self-evident. Programs thus become safer to evolve [12]. There is an unfortunate irony: the refactoring process can itself be unsafe. When soundness preconditions are not identified or cannot be assured, for example, due to the lack of models, both automated and manual refactoring can be risky. Refactorings implemented in tools may be unsound, particularly with respect to concurrency and other non-local program attributes.

Determining whether the program satisfies the preconditions for a refactoring rule may require explicit knowledge of non-local design intent: *e.g.*, *What state might be read or written by this method? Is this field aliased? Is this class intended to be subclassed? Who are the clients of this class?* [19, 21]. Best practice for manual refactoring is generally considered to require explicit reverse engineering, possibly using programming tools to search program text, and then applying a refactoring either by manually manipulating program text or using a tool [12]. Most automated refactorings will result in compilable code. But this is not sufficient to guarantee that program behavior is preserved.

Indeed, it appears that there is a kind of pragmatic tradeoff between soundness and "manipulative power" evident in the present generation of tool-implemented refactorings. If so, increments in our ability to assure soundness of refactorings could have a significant impact on the range and sophistication of transformations available through mainstream code development tools.

1.1 Outline

In this paper we (1) summarize our approach to tool-assisted modeling and assurance for concurrent programs, and (2) provide an account of recent case-study experience on larger-scale production Java systems such as jEdit, Log4J, `util.concurrent` (a widely used concurrency library), as well as several commercial and government systems. We then (3) offer an approach to principled co-evolution of code and models that is intended to meet the practicability criteria above, and (4) propose some concurrency-related refactorings that, on the basis of models and analyses, can be implemented soundly in tools. Because models are integral to our approach, it is important to avoid any requirement for programmers to reinvent models after restructuring code. Therefore, in our approach to refactoring, code and models are manipulated simultaneously. The intent of this approach is to enable a more powerful generation of refactorings, including refactorings whose purpose is to assist developers in making effective use of concurrency.

2. OUR ASSURANCE TOOL

We have implemented a prototype tool, sketched in [15], within the Eclipse IDE. This seemingly benign plug-in embodies the program analysis and assurance techniques described in previous work [14, 6, 16, 13]. It has its own internal representation that supports a variety of views and analyses, detailed below. We have applied this prototype tool to a number of mid-scale production concurrent Java programs, and have had success in recovering and capturing portions of concurrency-related design intent for these systems. The tool provides both bad news and good news: We have uncovered a number of previously unknown concurrency errors, and we have provided analytic assurances regarding consistency of code and models. In this section, we provide a brief overview of the capabilities of our tool, describe programmer interaction with our tool, and report on case study experience with our tool.

2.1 Tool Capability

Programmers using our tool record design intent in terms of properties the programmer is usually already concerned with. Models of design intent are expressed as source-code program annotations in a format familiar to users of Javadoc and JML. A design goal is for each annotation to provide some immediate value by answering a question about the code. This is a crude incentive system: Working programmers on deadline should want to introduce annotations because they receive near-immediate benefits that are useful to ongoing development activity—as well as to overall quality assurance of the evolving system.

Our tool allows analysis to proceed in increments across the code base and associated models. An unannotated class is merely a class that has no models against which it can be verified. Unannotated Java code is not somehow wrong; it just lacks claims of consistency with design intent. There are tools, including RaceFreeJava [11] and Guava [1] that can assist in providing assurance of thread safety. But these tools generally require the entire program to be assured thread-safe at once. While the modular type systems they use allow the program to be analyzed on a per-class basis, they nonetheless require the whole program to be annotated before meaningful analysis can be performed.

Our annotations for expressing design intent and their associated analyses can be categorized as follows:

- **Aggregations of state.** These declarations enable a programmer to declare abstract hierarchical "regions" of state that can both subdivide and span across objects. These state models can exploit uniqueness of references to aggregate uniquely referenced objects into the state of other objects.
- **Effects.** A programmer can declare the upper bounds of a method's effects—the state it reads and writes—in terms of regions. Analysis can verify that implementations respect the declarations and suggest appropriate declarations for unannotated methods [14, 13].
- **Aliasing intent.** These declarations enable a programmer to declare that a field or return value is intended to be unaliased. Parameters that are not aliased by methods can also be declared and verified [6].
- **Locking intent.** These declarations enable programmers to declare models that associate locks with state. Analyses can verify that state is accessed only when the appropriate lock is held [16, 13]. These annotations use the models of state provided by regions. In addition, the programmer can declare that a method requires a particular lock to be held by the caller, and can declare that a method returns a particular lock.
- **Concurrency policy.** These declarations enable identification of methods that may be safely executed concurrently [16, 13]. In general, the programmer declares which methods have safe interleavings based on their critical sections. We hypothesize that, for lock-based concurrency, concurrency policy combined with models of locking intent is a suitable surrogate for representation invariants.
- **Thread identification.** These declarations provide a way to associate particular threads with code seg-

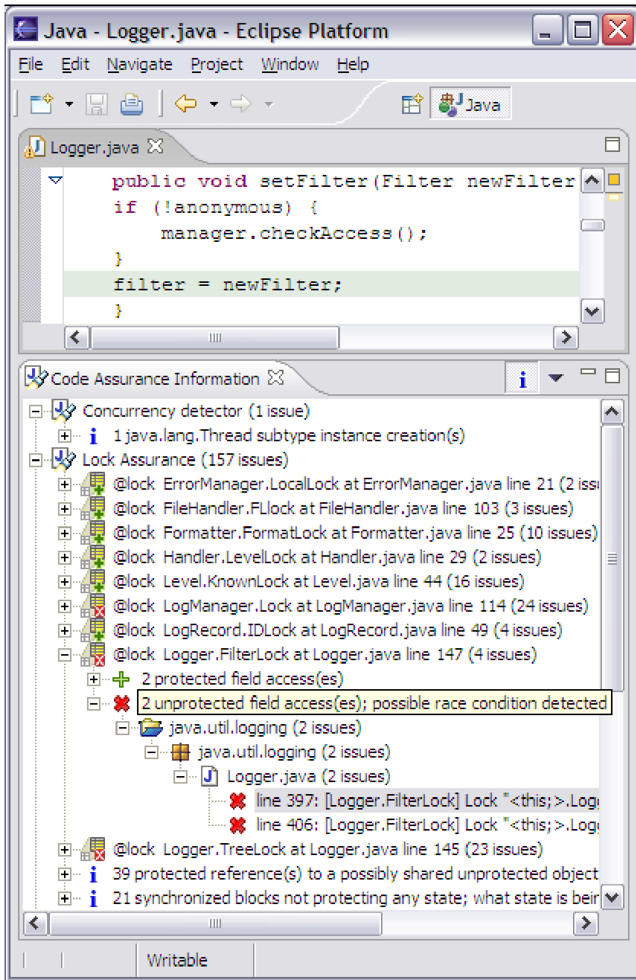


Figure 1: Our prototype concurrency assurance tool

ments and regions of state [22]. We do not elaborate these ideas here in this paper, except to note that this provides an approach to management of the non-lock concurrency typical of GUIs, as well as support for static assurances regarding appropriate use of threads in real-time Java code [5].

These models of design intent have been sufficient to capture the majority of Java concurrent programming idioms we have encountered in a broad sampling of production systems, including both commercial code and widely-adopted (high-quality) open source code. But these models are not sufficient to capture all concurrency design patterns. For example, we presently do not model and reason about thread-local objects, and we cannot yet describe designs that use arrays of locks or other indirect means of referring to locks.

2.2 Programmer–Tool Interaction

Figure 1 shows a portion of the user interface of our prototype Eclipse-based tool. The programmer enters annotations (examples of which are provided below) within his or her code using the normal Eclipse Java editor. As soon as a compilation unit containing annotations is saved, a build is done, analyses are executed, and tool results are displayed.

System	kSLOC	Annotations
jEdit v4.1	72.3	36
log4j v1.2.8	19.8	43
util.concurrent v1.3.2	10.3	158
util.logging v1.4.1_01	2.3	45
sponsor program	7.4	12

Table 1: Programs used as case studies.

Our tool, similar to the Eclipse Java compiler, is incremental and runs in the background while the programmer continues his or her work. Thus, the tool unobtrusively monitors model–code consistency as a programmer works on code and provides quick feedback as a programmer works to express models. Generally speaking, the time to complete the analyses is a function of the number of models and the complexity of the code they are associated with.

The results window shown at the bottom of Figure 1 reports model–code assurance results with a green "+" to indicate consistency and a red "×" to indicate inconsistency. A blue "**i**" highlights potential next steps for the programmer—inferred from the existing code and models. For example, the bottom "**i**" result in Figure 1 highlights 21 cases where locks that have not been explicitly associated, via a model, with lock-protected state are used within segments of code. Because these "**i**" results, like any inference not based upon explicit design intent, are subject to false positives, they can be filtered out by toggling the blue "**i**" button in the upper right of the "Code Assurance Information" window. The lower right corner of the model icons indicate the overall status of verification with respect to that model.

2.3 Tool Experience

We have applied our tool to a number of concurrent Java programs from established open source projects as well as industry and government systems. We elaborate four examples below: `java.util.logging`, the Apache Jakarta Log4j library, the open source text editor jEdit[1], and the well known concurrency utilities package `util.concurrent`.[2] In most of the systems we examined we both uncovered race conditions and obtained many positive results. The positive results are in the form of captured design intent coupled with analyses that verify consistency of code with the models.

The size and number of annotations added to each program discussed below are shown in Table 1. Our first example illustrates the kinds of errors well-intentioned developers can make in evolving a class definition with a non-trivial and unstated concurrency model.

2.3.1 *Case Study: Log4j's* `BoundedFIFO`

Our annotation language, similar to Javadoc and JML, is described in detail in [16, 13]. We introduce our technique here using the `BoundedFIFO` class, which implements a shared buffer between two threads, taken from the Apache Jakarta Log4j source code.[3] Log4j is a widely adopted library for

[1] `http://www.jedit.org/`. Bugs 893519 and 893735.

[2] `http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html`.

[3] `http://logging.apache.org/log4j/docs/index.html`. Log4j is ©1999 Apache Software Foundation.

event logging within Java programs (with capabilities similar to its SDK successor `java.util.logging`). Consider the annotated and elided code fragment below:

```
1 /** @lock BufLock is this protects Instance */
2 public class BoundedFIFO { ...
3   /** @aggregate [] into Instance
4    *  @unshared */
5   LoggingEvent[] buf;
6   int numElts = 0, first = 0, next = 0, size;
7
8   /** @singleThreaded
9    *  @borrowed this */
10  public BoundedFIFO(int size) {...}
11
12  /** @requiresLock BufLock */
13  public LoggingEvent get() {...}
14
15  /** @requiresLock BufLock */
16  public void put(LoggingEvent o) {...}
17
18  public synchronized void resize(int newSize)
19 }
```

Recall that Java has no way to express the association of locks with shared state, though there are conventions. The `@lock` annotation on line 1 declares that the `BoundedFIFO` object instance, `this`, is intended to protect all the object's fields. Specifically, this `@lock` annotation declares that the region `Instance` is protected by the object referenced by `this`. The lock is given a name, `BufLock`, because the object may otherwise be anonymous. The region `Instance` is a default region declared in `Object` and is automatically populated with the instance fields. Fields may alternatively have an explicit parent region declared via `@mapInto` (an example use is in a separate example below).

The `@requiresLock` annotation on lines 12 and 15 indicates that holding a lock on the `BoundedFIFO` object (*e.g.*, `synchronized (fifo) { e = fifo.get(); }`) is intended to be a prerequisite for callers invoking these methods. The contents of the buffer are actually a separate object, the array `buf`. Lines 3 and 4 aggregate the elements of `buf` into the state of the `BoundedFIFO` object instance. They also declare that references to the array are not "leaked" to other objects within the program—that the only references are from within the `BoundedFIFO` object. Leakage may also occur from constructors. Line 8 declares that during construction, the new object is only accessed by a single thread, *i.e.*, the one that invoked the constructor. Knowing this, our tool does not have to enforce the use of a locking protocol within the constructor's implementation. Finally, line 9 declares that the constructor does not "leak" a reference to the object itself; this is used to assure the consistency of the `@singleThreaded` annotation. On the basis of this model, our tool provides an assurance of consistency of code and model. Multiple program analyses contribute to this, including binding, typing, unique references (a specialized alias analysis), may-equal (another specialized alias analysis), effects, and our special-purpose lock analysis.

Between versions 1.0.4 and 1.1b1 of Log4j, a `resize()` method, declared on line 18 above, was added to enable resizing of the buffer. Unlike the other methods in the class, `resize()` is `synchronized`. Our tool assures the safety of this new method without new annotation. (As an aside, we note that introducing this method led to a policy failure relating to the use a wait–notify protocol—see our reports in Apache bugs 1505, 1507, 23912, and 26224.)

2.3.2 *Case Study:* `java.util.logging.Logger`

The class `java.util.logging.Logger` was introduced in Java 2 SDK version 1.4, and was designed for multi-threading. The JavaDoc claims that "All methods on Logger are multi-thread safe." We introduced a single annotation to express the locking model.

```
1 /** @lock FilterLock is this protects filter */
2 public class Logger { ...
3   private Filter filter;
4
5   public void setFilter(Filter newFilter)
6   throws SecurityException {
7     if (!anonymous) manager.checkAccess();
8     filter = newFilter;
9   }
10
11  public void log(LogRecord record) { ...
12    synchronized (this) {
13      if (filter != null && !filter.isLoggable(record))
14        return;
15    } ...
16  }
17 }
```

The tool fails to establish assurance because method `setFilter` does not acquire `FilterLock` before writing to field `filter`. This enables a race with the `log` method in which `log` checks that `filter` is non-`null`, `setFilter` writes `null` to `filter` and then `log` dereferences the now-`null` `filter` resulting in an exception. This example highlights the non-local character of concurrent programming that makes it so difficult to debug: even though `log` is written correctly, it is compromised by the incorrect `setFilter` method. The race can be fixed by enclosing the assignment to `filter` at line 8 above within a `synchronized (this)` block (see Java Bug Parade ID 4779253).

2.3.3 *Case Study: Wrong lock*

Many classes maintain static fields and methods to support unique identification of instance objects. Here is a code pattern based on code from a corporate partner:

```
1 public class C { ...
2   private static int nextID = 0;
3   private int id;
4
5   protected C() { assignID(); }
6
7   private synchronized void assignID() {
8     id = nextID++;
9   }
10 }
```

In our reverse engineering process, we first attempted to model the policy using an instance lock to protect the field `id`:

`@lock IdLock is this protects id`

We inferred this from the lock used in the code above. The tool informed that none of the other (many) uses of the field `id` were protected by this lock. We concluded (1) the actual state protected by `IdLock` is the `static` field `nextID`, and (2) that state should actually be protected by locking on the class `C` rather than on the diversity of instances:

```
1 /** @lock IdLock is class protects nextID */
2 public class C { ...
3   private static int nextID = 0;
```

```
4   private int id;
5
6   protected MemoryType() { id = getNextID(); }
7
8   private static synchronized int getNextID() {
9     return nextID++;
10  }
11 }
```

2.3.4 Case Study: Util.concurrent

The `util.concurrent` library contains a set of sophisticated synchronized wrapper classes for scalar types that additionally include suites of common operations that are meant to be atomic. We investigated this library with our tool. So far, we have developed 21 models within this library, and we have been able to provide positive assurance for all but two of these. One of the two exploits subtleties of the Java Memory Model and is beyond the present capabilities of the tool. The other model is, in fact, not consistent with code due to a subtle race condition detected by our tool.

This model involves the class `SynchronizedVariable` and its subclasses `SynchronizedInt`, `SynchronizedLong`, `SynchronizedFloat`, *etc.*

We expressed our model by adding region and lock declarations to `SynchronizedVariable`.

```
1 /**
2  * @region public Value
3  * @lock Lock is lock_ protects Value
4  */
5 public class SynchronizedVariable {
6   protected final Object lock_;
7   ...
8 }
```

The new region `Value` has no state in `SynchronizedVariable`; the intent is that it apply to subregions added by the various subclasses. Each subclass declares a field `value_`, for which we add the design intent that it is part of the region `Value`. For example:

```
1 public class SynchronizedChar
2 extends SynchronizedVariable implements ... {
3   /** @mapInto Value */
4   protected char value_;
5   ...
6 }
```

In version 1.3.2 of the library, the class `SynchronizedLong` fails to assure because there is an unprotected access to `long value_` at the end of the method `swap()` (code is not shown here). In general, even simple getters may require locks to be held to ensure that inappropriate intermediate values are not returned (though there are exceptions to this rule). In the specific case of 64-bit primitive types, a simultaneous access can yield a bad value because two separate 32-bit accesses are used needed to retrieve the composite value. A minor code change, now part of the current version of the library, allowed us to assure consistency of the code and model. (The lock was correctly held in the code for the other 64-bit primitive type, `SynchronizedDouble`.)

2.3.5 Case Study: jEdit

The `jEdit` project is an open source programmer's text editor. It contains a class `BufferListSet` that has a field `files` referencing an array of Strings. Our reverse engineering suggested that the lock policy was for access to both `files` and the array it references to be protected by the `BufferListSet` object. We expressed the model as follows:

```
1 /**
2  * @region protected FilesList
3  * @lock FilesLock is this protects FilesList
4  */
5 public abstract class BufferListSet
6 implements SearchFileSet { ...
7
8   /** @return {@unique} */
9   public synchronized String[]
10      getFiles(View view) { ... }
11
12   /**
13    * @mapInto FilesList
14    * @unshared
15    * @aggregate [] into FilesList
16    */
17   private String[] files;
18
19   public void invalidateCachedList() {
20     files = null;
21   }
22 }
```

These annotations create (at line 2) a region named `FilesList` that includes the field `files` (line 13). The locking policy (line 3) states that `FilesList` is protected by locking the object instance. The array (line 14) referenced by field `files` is made part of the region `FilesList` (line 15). In our model we note that it is uniquely referenced, *i.e.*, unaliased. The `@return {@unique}` annotation (line 8) indicates that there is no retained reference to the value returned.

Because method `invalidateCachedList` is not `synchronized`, verification fails. This can cause a `NullPointerException` because it could set `files` to `null` after another method has checked for the fact that the field is non-`null`. (Details in jEdit Bug 893735.)

From the standpoint of evolution, it is interesting to note that in this example and the Log4j example described above, the problems were introduced when new methods (in this case, `invalidateCachedList`) were added to a class during evolution, when developers might not have had an accurate memory of original design intent.[4]

2.3.6 Discussion

We have found that attempts to fix seemingly simple verification failures often reveal deeper issues. For example, a single negative result, when coupled with numerous related positive verification results, can reveal inconsistencies in the design intent embodied in the code. This experience is similar to that of Hovemeyer and Pugh in [17], where they report that bug pattern detectors often serve as "confusion detectors." Simple examples of such situations are when a field is only sometimes accessed from a critical section, or when one lock is used sometimes to protect one field and other times another. (There are safe—*i.e.*, race-free—ways to use these patterns, further complicating the issue.) A more complex example comes from another production system: the object used as a lock to protect a mutable field `f` is the object referenced by `f`. Thus, the lock on the field changes with the value of the field, which was not the intent.

[4]We learned this by examining the change logs of the classes.

Trust	Modify	Scenario
Model	Model	Evolved Model
Model	Code	Assured Evolution of Code
Code	Model	Reverse-Engineered Model
Code	Code	Assured Evolution of Intent

Table 2: Model–code evolution scenarios.

3. CO-EVOLVING CODE AND MODELS

Assurance of the consistency between design intent and code can fail for several reasons. Most obviously: (1) there are errors in the source code—a bug in the program—or (2) there are errors in the models of design intent—bugs in the model or its expression as program annotation. Consistency can be restored in the first case by correcting the code and in second case by correcting the model. There are two additional reasons: (3) the analysis capability is insufficiently powerful to verify consistency, and (4) it is not possible to achieve consistency between model and code (i.e., produce a safe *i.e.*, race-free program) without modifying both.

Based on our experience with our tool, we have identified four evolution scenarios corresponding to the cross product of whether the programmer *a priori* trusts the model or the code and whether the programmer modifies the model or the code: see Table 2. We now discuss how our tool can assist a programmer to *reestablish* code–model consistency in each scenario. Underlying this discussion is the recognition that the reality of the process is an ongoing *coevolution* of code and design intent, which consists of many steps of the four kinds considered here.

3.1 Evolved Model

In this scenario, the programmer both trusts and modifies the model. Negative analysis results identify segments of code that need to be modified to establish conformance with the new model. For example, we might decide that instances of `BoundedFIFO` (above) should be protected by the object referenced by a new field `lock` instead of by `this`. We would update the model by changing its `@lock` annotation to be

```
@lock BufLock is lock protects Instance
```

and by adding a new field to the class:

```
public final Object lock = new Object();
```
[5]

In fact, our first step could be to alter the annotation alone: assurance would identify that no field named `lock` exists. We would thus be guided on how to restore consistency.

Once the new lock model with the new representation of `BufLock` is declared, our tool will identify all known call sites of the methods `put` and `get` as being inconsistent with the methods' preconditions, and will identify all uses of the class's fields within `synchronized` method `resize` as being unprotected. We already know *what* must be done to restore consistency—because we have deliberately changed the model—but we may not have known *where* in the code to implement the new model. Here, the tool results conveniently focus our attention to exactly those `synchronized` blocks that need to be updated.

3.2 Assured Evolution of Code

[5]We make the field `public` to ensure that it is visible to the clients of the object. Were we to make the field less visible, our assurance tool would force us to introduce a `public` "lock getter" method into the class.

Consider now the case where a programmer trusts the design intent and modifies the code, with intent to maintain consistency with the model. In general, this scenario describes program implementation and maintenance. It is during these activities that the programmer can inadvertently introduce bugs while introducing new functionality or, even worse, fixing existing bugs. Using our tool, however, analysis results would indicate whether the programmer successfully maintained consistency and, perhaps more usually, focus attention to segments of code where consistency is lost, *e.g.*, identifying a potential data race or other bug. Consider again the jEdit example introduced in Section 2.3.5 in which the addition of a new method introduced a race condition. Use of our techniques could have refocused the programmer's attention from extending functionality to compliance with the locking model.

3.3 Reverse-Engineered Model

When the programmer trusts the code and changes the model, he is essentially reverse engineering the code to evolve the model to more accurately describe the implementation reality. Addition of model information, *e.g.*, by adding annotations to code, amounts to hypothesizing a model with which the code might be consistent. Analysis tests this hypothesis. To illustrate, consider the case where a field `f` is accessed from many `synchronized` methods. The programmer might hypothesize that the field is protected by the object itself, and thus annotate the containing class with

```
@lock FLock is this protects f
```

Analysis might indicate that a majority of the uses of `f` are protected under this model, while several of them are not. Suppose further that the negatively assured cases all occur within private methods that are called from `synchronized` methods. The programmer can improve the model by annotating those private methods with `@requiresLock FLock`, declaring the intent that callers should acquire the lock.

3.4 Assured Evolution of Intent

In the fourth scenario, the programmer both trusts and modifies the code—deliberately changing the design intent embodied in the code. Analysis assists by identifying code segments that are inconsistent with the old model, and thus implicitly identifying segments of the old model that need to be updated. Furthermore, because the tool basically identifies the segments of code that have changed, the programmer knows where to look in the code for manifestations of the new evolved model. This scenario is the dual to the "evolved model" scenario: in the later case the model is evolved and then the code is evolved to ensure consistency; in this case the code is first evolved and then the model is evolved to restore consistency.

Consider again the case of changing the lock that protects an instance of class `BoundedFIFO`. Suppose we first changed the code by introducing the new field and modifying the `synchronized` blocks and declarations as appropriate. Negative analysis results would point to the callsites of `put` and `get` and to the implementation of `resize` and report that the `BoundedFIFO` object is not being locked per the existing annotations of design intent. Again, these assurance failures suggest which segment of the model must be modified—the lock representation—and where to look for the new representation: in the `synchronized` blocks surrounding the now-inconsistent code.

4. REFACTORING CONCURRENT PROGRAMS

Refactorings are patterns for systematic restructuring of code. In many tools, automatic support for refactorings is provided as program transformations. As we noted in Section 1, automated refactoring generally requires explicit modeling of programmer design intent in order to assure soundness. This means that refactoring is risky for both programmers and tool implementors. The transformed code may be more difficult to understand than the original code, and also broken because an unstated precondition is not met. For example, subtle changes to order of computation can be introduced when field declarations with initializers are hoisted to superclasses. Another example is the extraction of new method definitions that leave behind substantive computation for actual parameter values.

For some refactorings, conservative analyses can compensate for missing intent [19]. Perhaps for this reason, the Java refactoring literature has generally focused on sequential programs. With model information and supporting analyses, it becomes more tractable to consider more ambitious refactoring and program transformations, including manipulation of concurrent programs.

The conventional usage model for refactorings assumes that the code being refactored is initially "good," though usually with respect to an unstated model. That is, the refactoring modifies the code, implicitly trusting that at the start the code is consistent with design intent, and results in code that remains consistent with design intent—and furthermore we assume that the refactorings do not modify the (usually unstated) design intent.

When models are explicit, however, the picture can be slightly different: refactorings can manipulate both code and models. In this section, we consider some problems arising from traditional refactorings when there are explicit models. We then describe an approach to refactoring concurrent programs that supports coevolution of code and models. We provide an example of one such refactoring: split lock.

4.1 Refactorings and Models

The first step we take is to incorporate the manipulation of models into the program transformation process. Without this, there can be risks to programmers making subsequent changes to code.

Extract Method. This refactoring replaces a programmer-selected sequence of statements with a call to a new method whose body is made up of those statements. Consider the case of extracting a sequence of statements that are nested within a `synchronized` block. Because the newly introduced call to the newly extracted method will still be within the `synchronized` block, it is easy to see that the code will be in the same state of consistency with the locking model as it was prior to the application of the refactoring. If subsequently any new calls are introduced to the extracted method definition, they must adhere to the requirement to hold the lock prior to making the call.

If the locking policy is explicit, this can be handled by adding a `@requiresLock` annotation to the newly extracted method. A program transformation operating with an explicit model could do this automatically.

An extract method transformation may also interact badly with non-lock–based concurrent designs. For example, the AWT enforces thread safety by requiring that certain methods (e.g., `paint` and `update`) be executed by the "AWT Thread" only. We can capture this design intent using "thread coloring," whereby threads are abstractly identified by colors, and segments of code are colored by the threads in which they are allowed to execute [22]. Here again, extract method needs to be made aware of the design intent so that it correctly propagates the colors to the newly extracted method. Otherwise, the new method could be run from an incorrectly colored thread.

Convert Local Variable to Field. This refactoring replaces a method-local variable with a new field declared in the class containing the method. The danger here is that we miss an opportunity to record design intent as the new field is created. Unlike extract method, we cannot use our analyses to catch this problem after the fact because, in general, no prior models exist for the field. This refactoring, and indeed any refactoring that introduces new fields, will benefit from interaction with the developer to capture the design intent for the field. To which region of state should the field belong? Is the new field intended to be accessed from multiple threads, and if so, how is access to it synchronized? Is the field intended to refer to an unaliased object? By capturing this information up front, correct use of the field can be checked from the outset.

4.2 Refactoring Concurrency

The presence of explicit models of concurrency-related design intent enables implementation of refactorings that directly affect how concurrency is managed within a program. We have identified a number of transformations, listed below, that are potentially applicable in a *generative approach to concurrency management*, in which more complex concurrency is introduced in a systematic fashion. In this approach, the programmer begins with a class definition formulated as simple monitor—all state is protected by the object itself and every public method is `synchronized`. Such a class has a simple concurrency policy (see Section 2.1): no method is allowed to interleave with any other method. The programmer then applies refactorings to modify the extent of concurrency supported by the class, updating both the code and the models describing its locking and concurrency policies. In particular, when applying these refactorings, the programmer may choose to liberalize the concurrency policy.

In these cases, there is an important distinction between "preserving meaning" and "retaining consistency with a model." This is due to the fact that our models are more abstract than fully elaborated representation invariants. This means that we achieve model compliance more often than preservation of meaning.

The split lock refactoring—considered in more detail below—decreases the granularity at which state is protected by moving it "down" the region hierarchy. That is, a lock used to protect a larger single region is replaced by multiple locks used to separately protect subregions. This increases opportunity for concurrency, but it can be dangerous if the underlying representation invariants involve close relationships among the new finer-grained regions.

The merge locks refactoring decreases the granularity of protection by moving "up" the region hierarchy, replacing multiple shared regions with a single ancestor shared region.

The shrink critical section refactoring alters the scope of the code in a critical section. The purpose is to move the boundary of a critical past statements that do not access shared

state.

The split critical section refactoring creates additional opportunities for method interleaving by converting a single `synchronized` block into a sequence of `synchronized` blocks.

Its dual, merge critical sections, may be used to remove interleaving opportunities and reduce lock acquisition.

The synchronize method and synchronize callsite transformations modify locking responsibility, move the responsibility between callee and caller, respectively. These transformations affect a body of code wider than the class definition because of the necessity of identifying and updating method callsites.

4.3 The Split Lock Refactoring

Split lock is parameterized by the `@lock` annotation of the lock to be split, which provides a lock L with name M associated with a shared region R. The lock annotations within the class, and all of its subclasses,[6] are analyzed to determine all the child regions $R_1, \ldots, R_n$ of R. For each R_i, the programmer is asked to provide a new mutex name M_i, and to identify a `final` field of the class or `this` to be used as the lock representation F_i for that region.

Let us assume, first, that there are no `@requiresLock` annotations. We start by identifying all the `synchronized` blocks that use the given lock L.[7] For each identified block:

1. Determine the child regions $\{R_j\}$ of R that are affected by the body of the block. These can be located by identifying the fields that are read or written and through knowledge of the region hierarchy.
2. Replace the `synchronized` block with a set of nested[8] `synchronized` blocks that acquire the appropriate locks $\{L_j\}$ for the regions $\{R_j\}$.

The class is then modified by adjusting annotations describing models:

- The original `@lock` annotation is removed.
- Any annotations `@returnsLock` M are removed. Depending on the visibility of the regions $\{R_j\}$, it may be necessary to introduce new lock getter methods, annotated with the appropriate `@returnsLock` annotation to ensure that the locks are as visible as the regions they protect.
- For each child region R_i of R, a new lock declaration is added to the class

  `@lock` M_i `is` F_i `protects` R_i

Handling `@requiresLock` annotations. We now consider the case of existing `@requiresLock` annotations. These are most easily dealt with by replacing all uses of M in them with $M_1, \ldots, M_n$. This is always sound, and does not interfere with existing uses of the methods. Because this affects the locks that must be acquired, it would be done before modifying any `synchronized` blocks. This approach, however, can unnecessarily constrain the future use of the methods because they may now require more locks than they may actually need, *e.g.*, the method only accesses one subregion of R. The difficulty in automatically making the annotations less restrictive is discerning the programmer's intent: the programmer might want to use less specific annotations to preserve flexibility for future uses of the method. Thus, the programmer should be supplied with an additional refactoring enabling him to modify the `@requiresLock` annotation of a method, but that uses analysis to prevent the programmer from under specifying the method's locking requirements.

Policy issues. Recall that split lock is expected to maintain consistency with the programmer-specified concurrency policy that describes how methods are allowed to interleave. Splitting a lock can introduce additional opportunities for methods to interleave—consider the case of two critical regions, where each one accesses a single subregion of R—and thus render the code inconsistent with the model. Thus, split lock may have to introduce additional locks that are not associated with any region but whose purpose is to enforce consistency with the stated concurrency policy. Such *policy locks* would be acquired prior to any of the new locks associated with regions R_i.

The point of split lock, however, is to enable additional concurrency. The programmer would use it first, and then follow it by a refactoring in which he redefines the concurrency policy to take advantage of some of the newly enabled interleavings. Here the tool can further assist the programmer by identifying all possible interleavings based on the critical sections extant in the code, and by highlighting those that are newly enabled. In addition to modifying the annotations defining the concurrency policy, the refactoring would remove some of the acquisitions of policy locks so that the code could take advantage of the newly liberalized intent.

4.4 Related Work

We are unaware of any specific proposals for concurrency-related refactorings. Lea describes splitting locks as a design concept [18, p. 127], but does not consider it as a refactoring, nor in the context of explicitly expressed design intent. Most closely related in the literature are compiler optimizations that modify the scope of critical sections. Escape analysis can remove critical sections from classes not used in a multi-threaded manner [3, 4, 8, 23]. There are also optimizations that remove "excess" concurrency to reduce lock-acquisition overhead. Plevyak, Zhang, and Chien [20] expose critical sections by inlining method calls, and then expand them to enable merging of adjacent critical Diniz. sections and Rinard [10] decrease the protection granularity in automatically parallelized object-oriented programs. All objects are originally protected by their own locks, which are then "coarsened." In later work [9], they use a technique for increasing the critical section size using flow-graph reachability. These techniques do not provide source-to-source transformation, and thus cannot be said to provide automated evolution of the program.

5. CONCLUSION

By augmenting concurrent Java programs with model information representing design intent related to "mechanical" program properties, it becomes possible both to use static analyses to assure consistency of code and model, and to develop refactorings that, while possibly changing some aspects of program meaning, are sound in the sense that they maintain consistency with existing design intent. It is also

[6]We must be able to determine the *complete* set of subclasses—*i.e.*, the class may be part of a framework, and we do not have access to all the subclasses introduced by clients of the framework. This problem is not unique to concurrency and we shall say no more about it herein.

[7]We consider a `synchronized` method to be a method whose body is enclosed in `synchronized (this) { ... }`.

[8]We do not consider deadlock in this paper, though in this case, it is obvious that the tool should solicit from the programmer an ordering for the new locks.

potentially possible to define refactorings that can restructure code to accommodate particular kinds of change in design intent. In this paper we have presented some initial steps in this exploration. In our experiments in applying our modeling and analysis tools to a variety of existing production systems, we have been able to (1) reverse engineer to identify model information, (2) perform analyses to assure consistency of model with code, (3) use modeling and analysis results to identify flaws in code. This experience illustrates the pervasiveness of race conditions, and also the potential value of more systematic approaches to developing and, particularly, evolving concurrent code. These approaches range from more disciplined use of models to tool-assisted refactorings, as suggested in the previous section.

6. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their comments. We thank additional project members Edwin Chan, Elissa Newman, Dean Sutherland, David Swasey, Greg Mathis, and John Boyland for their help. Effort sponsored in part through the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298 and in part by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory (AFRL), Air Force Materiel Command, USAF, under agreement number F30602-99-2-0522. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of NASA, DARPA, AFRL, or the U.S. Government.

7. REFERENCES

[1] D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: A dialect of Java without data races. In *OOPSLA '00*, pages 382–400.

[2] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proceedings of the 29th Symposium on Principles of Programming Languages*, pages 1–3, New York, Jan. 2002. ACM Press.

[3] B. Blanchet. Escape analysis for object-oriented languages: application to Java. In *OOPSLA '99*, pages 20–34.

[4] J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *OOPSLA '99*, pages 35–46.

[5] G. Bollella, J. Gosling, B. Brosgol, J. Gosling, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[6] J. Boyland. Alias burying: Unique variables without destructive reads. *Softw. Pract. and Exper.*, 31(6):533–553, 2001.

[7] E. C. Chan, J. T. Boyland, and W. L. Scherlis. Promises: Limited specifications for analysis and manipulation. In *ICSE '98*, pages 167–176.

[8] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *OOPSLA '99*, pages 1–19.

[9] P. Diniz and M. Rinard. Synchronization transformations for parallel computing. In *POPL '97*, pages 187–200.

[10] P. Diniz and M. Rinard. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. In *Ninth International Workshop, Languages and Compilers for Parallel Computing*, pages 285–299, 1996.

[11] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *PLDI '00*, pages 219–232.

[12] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[13] A. Greenhouse. *A Programmer-Oriented Approach to Safe Concurrency*. PhD thesis, Carnegie Mellon University, 2003.

[14] A. Greenhouse and J. Boyland. An object-oriented effects system. In *ECOOP '99*, pages 205–229.

[15] A. Greenhouse, T. J. Halloran, and W. L. Scherlis. Using Eclipse to demonstrate positive static assurance of java program concurrency design intent. In *eTX Workshop 2003*, pages 101–105. `http://doi.acm.org/10.1145/965660.965681`.

[16] A. Greenhouse and W. L. Scherlis. Assuring and evolving concurrent programs: Annotations and policy. In *ICSE '02*, pages 453–463.

[17] D. Hovemeyer and W. Pugh. Finding bugs is easy. `http://www.cs.umd.edu/~pugh/java/bugs/`.

[18] D. Lea. *Concurrent Programming in Java*. The Java Series. Addison-Wesley, second edition, 2000.

[19] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, 1992.

[20] J. Plevyak, X. Zhang, and A. A. Chien. Obtaining sequential efficiency for concurrent object-oriented languages. In *POPL '95*, pages 311–321.

[21] W. L. Scherlis. Systematic change of data representation: Program manipulations and a case study. In *ESOP '98*, pages 252–266.

[22] D. F. Sutherland, A. Greenhouse, and W. L. Scherlis. The code of many colors: Relating threads to code and shared state. In *PASTE '02*, pages 77–83.

[23] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *OOPSLA '99*, pages 187–206.

Author Index

Cole, C. 63
Gopalakrishnan, G. 26
Greenhouse, A. 90
Halloran, T. 90
Harris, T. 46
Herlihy, M. 63
Hicks, M. 18
Hovemeyer, D. 80
Jagannathan, S. 54
Lea, D. 1
Lindstrom, G. 26
Manson, J. 36
Pizlo, F. 54
Potter, J. 10
Prochazka, M. 54
Pugh, W. 36, 80
Rose, J. 18
Scherer, W. 70
Scherlis, W. 90
Scott, M. 70
Shanneb, A. 10
Swamy, N. 18
Vitek, J. 54
Yang, Y. 26
Yu, E. 10