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 sophisticated 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.

[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.

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.

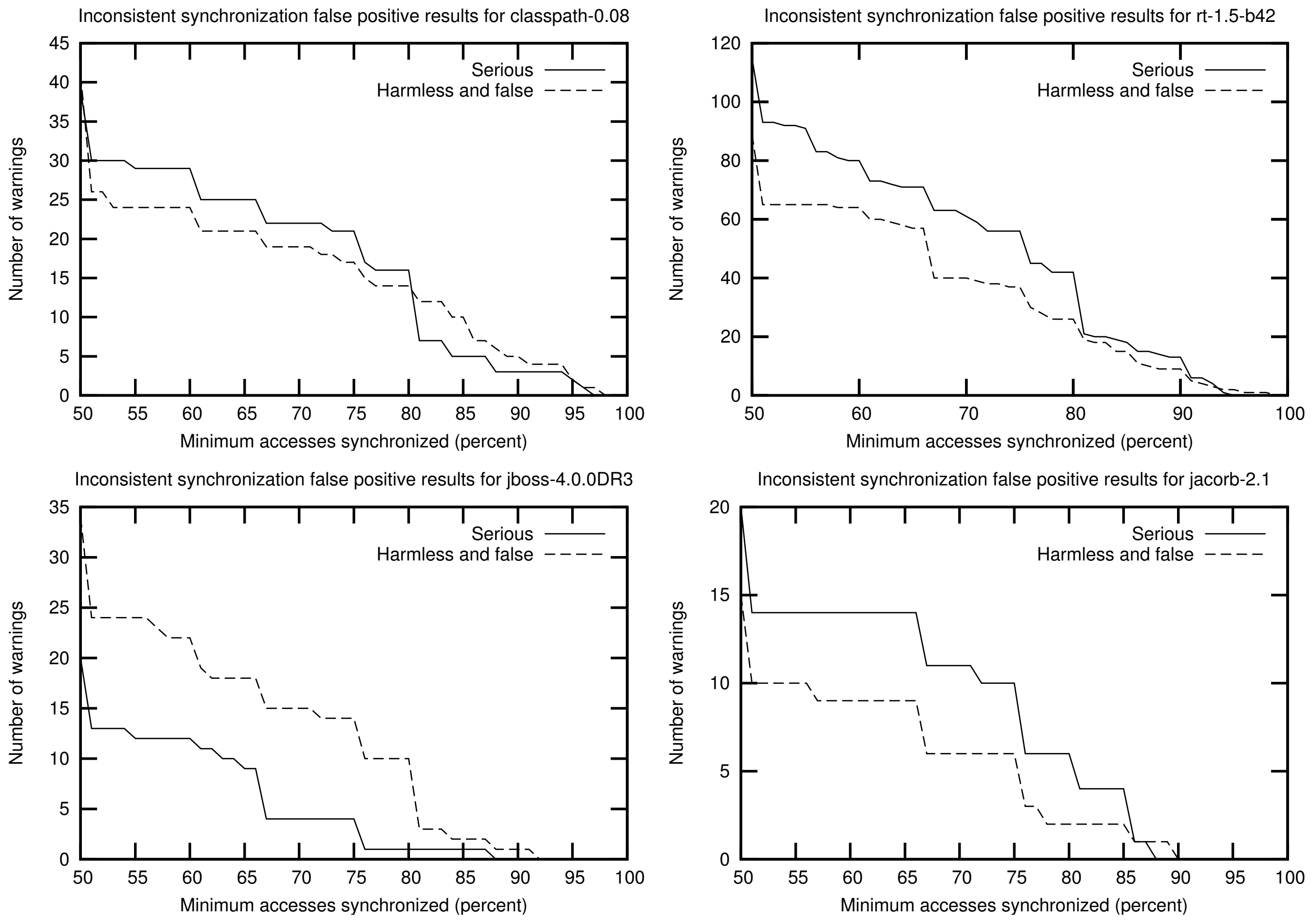

Figure 1: Serious bugs and false and harmless warnings for varying values for minimum percentage of synchronized accesses. The applications are two implementations of the J2SE core libraries (rt-1.5-b42 and classpath-0.08), a open source Java application server (jboss-4.0.0DR3), and an open source CORBA 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 program 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.