Detecting threading-related errors in Java programs using AspectJ

ICA-3019691
Timon Bijlsma
`Graduate student of Utrecht University`
`Dept. of Information and Computing Sciences`

February 5, 2014

Contents

Chapter 1

Introduction

With the rise of multi-core processors in consumer level hardware over the past decade, writing concurrent programs has become an important skill in order to make full use of the available hardware. Non-trivial interactions between concurrently running threads greatly increase the complexity of such programs. Adding to the problem is an inherent non-determinism in the relative timing of interactions between these threads. It's possible, and in fact somewhat common, for a program to function correctly most of the time yet still contain multi-threading errors which exhibit themselves only under specific thread schedules. This category of problems can be extremely difficult to detect and fix due to the lack of repeatability as well as the complexity of the order of events sometimes necessary to trigger the bug. Furthermore, some thread interleavings may be extremely unlikely to occur in practice depending on the underlying hardware, operating system and Java Virtual Machine (JVM). Normal unit tests are often inadequate in finding these errors, as even repeated testing will miss some of the more rare problematic interleavings. In fact, because tests are mostly run on the same hardware with the same or similar software, there's a high likelihood of encountering only a very limited subset of the possible interactions.

For normal unit tests to retain their usefulness even when testing multi-threaded algorithms, several changes must be made to the underlying platform. Primarily, the thread scheduling must be made repeatable so that when a threading-related bug is discovered, an attempted fix for it can be tested with the same thread scheduling decisions. Being able to re-run failing code also allows the tester to step through the program with a debugger or other added instrumentation which can be important in finding out the cause of the problem. Before repeatability can come into play however, a problematic thread schedule has to be found first. An efficient implementation requires a controllable thread scheduler, usually guided by a heuristic designed to detect a specific type of concurrency problem. Solving the problem of repeatability requires taking full control of the thread scheduling in order to record and repeat a chosen interleaving. Implementing repeatability requires making the thread scheduler drivable, heuristics to find specific concurrency errors constitute knowing where to drive.

The required changes can be implemented within a custom JVM, replacing the standard operating system backed threads with ones scheduled and managed entirely by the virtual machine's own code. While perhaps the most straightforward approach, requiring a custom JVM implementation just for testing can be inconvenient.

Implementation at the JVM level isn't the only way to manipulate the thread scheduler. It's possible to gain the necessary level of control by injecting code into the tested program itself rather than the platform it runs on. This allows code to run in the same environment as usual with no changes other than the injected instrumentation. The framework uses internal locks in a way that effectively forces the scheduler to run a specific thread chosen by a user-overridable algorithm.

A number of different tools exist for injecting additional code into an existing Java program. AspectJ[1] is one of these tools, providing a high-level pattern-based language for determining at which code locations to insert the instrumentation. Other tools provide more low-level control over the actual bytecode generated by compiling Java classes. Considering all language semantics are defined on the Java level, working on a lower level is distracting and unnecessary.

The goal then is to create a framework to instrument existing Java code in an unobtrusive way, detecting (potential) concurrency-related problems at run-time. A number of such frameworks have been created over the years, most of them focusing on some specific subset of problems. These often don't take the Java Memory Model[2] into account and the ones that do use a simplified form, ignoring some of the subtleties which occasionally come up in the implementation of high-performance code. A complete and accurate implementation of the relaxed constraints on inter-thread semantics specified in the memory model is required to make a correct judgment on potential concurrency errors. The transparent implementation of the rules specified by the memory model provide a solid theoretical foundation for the validity of detected problems.

1.1 Related Work

1.1.1 Static Analysis

An alternative approach to run-time instrumentation, static analysis involves constructing a model from the tested program's source code and performing analysis on that model without running the actual code. Static analysis tools are capable of exhaustively exploring a programs state, but they may be hindered by practical limits of memory and time. When each possible thread interleaving has to be taken into account, the state space grows so rapidly that static testing quickly becomes impractical. Static testing is often used as a pre-processing step for dynamic algorithms[3, 4] either as a guide to find information about the program, or to exclude parts from the dynamic analysis that can be proven safe. Hybrid techniques, building and refining a limited model while partially running the program, have also been proposed[5].

1.1.2 Modified Java Virtual Machines

As mentioned earlier, modification of the JVM itself is another way in which control over the thread scheduling can be obtained. Using a custom or research VM gives the highest degree of control over the way in which code is executed, allowing for more exotic features such as rollback/snapshot support[6, 7]. Rollback in particular is a convenient tool in attempts to exhaustively test a program by allowing the code to go back in time and enter a different branch. Some of the more extensive modifications may not be compatible with all Java language features, for instance rollback has problems with native code.

1.1.3 AspectJ

The use of AspectJ for the type of invasive instrumentation required here isn't yet very widespread. Older solutions often relied on modifying the JVM[6, 7] or instrumentation through more low-level bytecode manipulation tools[4]. That's not to say no AspectJ-based dynamic analysis implementations exist. Racer[3] implements a simplified data race detection algorithm called ERASER[8] by means of AspectJ instrumentation. The elegance and straightforwardness of Racer's implementation were a major inspiration to use AspectJ as well.

Chapter 2

Concurrency in Java

Threads form the backbone of any concurrent Java program. Be it directly or through thread pools, they're the only language feature explicitly allowing multiple pieces of code to run at the same time. If threads would never interact there would be no problem, but some additional language features are required to let threads safely operate on the same data at the same time.

2.1 Synchronized Blocks

Reads and writes to most types of fields are atomic[9] in Java (double and long are the exception2.7.2). Atomicity prevents the potential problem of partial writes, where two concurrent writes to a field intertwine and the field ends up with an unexpected value. What atomic writes do not protect against however is the following:

Thread A	Thread B

`//Withdraw 50`
`if (balance >= 50) {`
`  balance -= 50;`
`}`
	`//Withdraw 70`
	`if (balance >= 70) {`
	`  balance -= 70;`
	`}`

Table 2.1: Concurrent memory access

With the above program, the following non-intuitive instruction interleaving is possible.

Possible instruction interleaving

```
   balance = 100;
A: if (balance >= 50)
B: if (balance >= 70)
```

```
A: balance -= 50;
B: balance -= 70;
```

This would leave the balance at `-20`, despite the explicit range check prior to decrementing. Another perhaps surprising thing is that the `-=` operator is *not* atomic. The assignment operator is atomic, but all the shorthand assignment operators (including `++` and `-`) are not.

The `synchronized` statement can be used to prevent multiple threads from accessing a shared resource concurrently.

Thread A

```java
synchronized (L) {
  //Withdraw 50
  if (balance >= 50) {
    balance -= 50;
  }
}
```

Thread B

```java
synchronized (L) {
  //Withdraw 70
  if (balance >= 70) {
    balance -= 70;
  }
}
```

Table 2.2: Fixing things with synchronization

In the above example, both threads synchronize on the shared object `L`. Every object in Java has an associated *monitor* which can be *obtained* by a thread. A monitor can only be obtained by one thread at a time, other threads trying to obtain that monitor will block until it becomes available. In the case of a synchronized statement, a monitor has to be obtained before entering and is automatically released when the thread leaves the synchronized block for any reason. The conflicting access to `balance` is prevented by wrapping all accesses to it with synchronization on the same monitor.

In addition to synchronized statements, entire methods can also be marked as synchronized. A synchronized method behaves as though its body were contained in a synchronized block with `this` as the monitor.

```java
//Synchronized method
synchronized void f() {
  //Do something
}
```

```java
//Equivalent
void f() {
  synchronized (this) {
    //Do something
  }
}
```

Table 2.3: Synchronized methods

Synchronized statements can be nested. The running thread acquires each monitor separately whenever it encounters a synchronized statement, blocking if the monitor is already in use by another thread. Threads can double-acquire the same monitor, for example by recursively calling a synchronized method. The monitor will then be released only once the thread leaves the topmost synchronized block.

```java
synchronized (L) {
  //Do something
  synchronized (M) {
    //Do something
  }
}
```

```java
synchronized (L) {
  //Do something

  //Legal
  synchronized (L) {
    //Do something
  }
}
```

Table 2.4: Examples of nested synchronized statements

2.2 Wait/Notify

Another way to interact with monitors is through the `wait/notify/notifyAll` methods defined in `java.lang.Object`. A call to `wait` puts the current thread to sleep. The `notify` method is used to wake up one of the threads waiting on the notified object's monitor. No guarantees are made as to which thread is woken up. `notifyAll` behaves similarly to `notify`, but wakes up all threads instead of just one. A waiting thread may also wake up for no apparent reason at all and failing to take these spurious wakeups into account can occasionally be a source of bugs.

An object's `wait` and `notify` methods may only be called by a thread currently holding that object's monitor. To allow this to work, `wait` temporarily releases the monitor it's waiting on for duration of the wait. If it didn't, no other thread would be able to obtain the monitor and therefore be incapable of using `notify`. This unique behavior of temporarily releasing a monitor will be important later in the implementation of the framework responsible for manipulating the thread scheduler.

Blocking queue example

```java
//Removes and returns the first item in the queue
//If empty, waits for an item to become available
public synchronized Object pop() {
  while (isEmpty()) {
    try {
      wait();
    } catch (InterruptedException ie) {
      //Wait interrupted
    }
  }
```

```java
    Object result = internalRemoveFirst();
    notifyAll();
    return result;
  }

  //Adds a new item to the back of the queue
  //If full, waits for a free slot to become available
  public synchronized void push(Object obj) {
    while (size() >= capacity()) {
      try {
        wait();
      } catch (InterruptedException ie) {
        //Wait interrupted
      }
    }
    ...
    notifyAll();
  }
```

The `wait`/`notify` pair is often used as an optimization when a thread needs to wait for a certain condition to become true. Even without the `wait`, the program would still function, but far less efficiently. Blocking the thread until the condition becomes true prevents it from running and wasting processor cycles.

Checking the wait condition in a loop is standard practice. It's always possible for unrelated code to call `notify` prematurely. A wait can also be interrupted2.3 at any time for no reason. Not checking the wait condition after waking up is therefore usually a bad idea and a potential bug.

2.3 Interrupts

Thread interrupts are perhaps not the first thing one would think of when listing functionality related to concurrency in Java, but they play a small role in various areas. Threads can interrupt each other by calling each other's `interrupt` methods. Calling this method sets a queryable flag on the interrupted thread and wakes that thread up from any (timed) wait with an `InterruptedException`.

2.4 Volatile Fields

The compiler has a lot of freedom to reorder or optimize reads and writes performed on regular fields. These optimizations are important to achieve acceptable performance, but when a field can be written to from an external source (like another thread), optimizing away even a seemingly redundant read could prove disastrous. Fields can be marked as `volatile` to prevent these types of

ThreadA	ThreadB

```
synchronized (L) {
  try {
    L.wait();
  } catch (
    InterruptedException ie)
    {
    //Wait interrupted
  }
}
```
	`ThreadA.interrupt();`

Table 2.5: Interrupting a wait

optimizations from occurring. Reads and writes to volatile fields may not be optimized away or reordered.

2.5 Final Fields

Final fields may not seem like they should have special meaning in concurrent programs, but Java provides some additional guarantees specific to final fields. Final fields must be given an initial value within the constructor of the object defining them. Once the constructor returns, any thread reading that field will see the initialized value. The interesting bit is that this is true *even in the presence of a data race.*2.7.2 Non-final fields don't have this guarantee and when passed through a data race may appear to have their default values rather than whatever they're set to in their object's constructor.

2.6 The Java Memory Model

Every language needs a set of rules that determine the range of allowable behaviors when multiple threads concurrently operate on the same memory locations. Shared fields are the primary way in which Java threads communicate with each other. Through only a handful of shared objects, a significant part of the entire object graph can become accessible to many threads at once. Taking into account the large segment of shared memory combined with threads potentially running on different processing cores, keeping every shared field consistently updated would seriously impact performance. To allow for optimal performance, most of the synchronization in Java requires deliberate action – either by using the synchronized statement, or by marking fields as volatile.

For single threaded programs, the behavior specified by Java's memory model is entirely intuitive. The observable behavior of a thread must be as though all instructions happen in *program order*: the same order as they appear in the source code. Behind the scenes, this gives the compiler some room to reorder or otherwise optimize things as long as the program's behavior doesn't change as seen by that one thread (intra-thread). What other threads observe (inter-thread) is limited by a much more lenient set of *happens-before* rules.

2.6.1 Happens-before order

The happens-before order forms the base of Java's memory model, setting limits to the range of acceptable behaviors for observability between different threads. The limited sequential consistency offers a balance between runtime performance and behavior which is reasonably intuitive to programmers. More specifically, happens-before enforces a partial order between the actions in multiple threads.

> When two actions are in a *happens-before* relation, `x happens-before y`, the effects of action x are observable by action y.

The term observable is important here – it's possible for an action to have been performed, but its effects not yet visible to other threads. Happens-before not only guarantees action x has been performed, but also that all of its effects are visible to the thread performing action y. The split between performing an action and having its results become visible to other threads has an important purpose: it leaves room for caching to occur. Requiring all writes to appear to every thread immediately can have a high runtime cost, especially when those threads run on separate processors with no shared cache.

The following happens-before rules are defined by the memory model.[10]

Happens-before rules

A If two actions, x and y, are in the same thread and x precedes y in program order, then x happens-before y.

B Any action in an object's constructor happens-before any action in that object's finalizer.

C An unlock action of any monitor happens-before all subsequent lock actions on that monitor by any thread.

D A write to any volatile field happens-before all subsequent reads of that field by any thread.

E An action that starts a thread happens-before the first action of the thread that was started by it.

F Initialization of default values for fields (null, 0, false) happens-before the first action in every thread.

G The final action of a thread happens-before any action detecting that thread's termination.

H An action interrupting a thread happens-before any action detecting that interrupt.

I If x happens-before y and y happens-before z, x happens-before z (happens-before is transitive).

In this context, the actions mentioned are all *inter-thread actions*. This includes all reads/writes to fields, locking/unlocking a monitor and anything which modifies a program's environment (like accessing a file stored on disk). Actions that start a thread or detect thread termination are considered inter-thread actions as well.

While seemingly innocuous, rules A and I together can create some rather complex situations. The presence of a happens-before relation between two threads gives guarantees beyond the specific actions involved in the happens-before. For example, a happens-before due to a volatile write by a thread T, read by another thread R. On the basis of rule A, there's a happens-before between earlier actions in T and the volatile write. Since happens-before is transitive (rule I), the happens-before between R and T then extends to all actions performed by T prior to the volatile write.

The difference in behavior between volatile fields and regular fields lies in the way they are handled by these happens-before rules. Rule D guarantees immediate visibility of any writes to a volatile field for all threads. Visibility of reads/writes to fields performed while in a synchronized block are similarly explained by rule C. This rule ensures changes become visible between threads locking on the same monitor. It also explains why locking on a different monitor does nothing, as there will be no happens-before relation in that case.

Some of the happens-before rules are related to thread lifetime. Fields must be initialized with some default value (rule F) prior to becoming readable in order to avoid the possibility of reading undefined data. Rules E and G are very convenient as they make spawning off some work to a background thread more convenient. In particular, rule G ensures all writes are made visible to others prior to a thread ending. This means calling `Thread.join` on a thread and waiting for it to finish is sufficient to see all writes performed by that thread without the need for further explicit synchronization. Also, while perhaps not the most obvious manner of synchronization, interrupts could potentially be used to establish a happens-before relation between threads on the basis of rule H.

Lastly, the rule concerning finalizers is more important than it initially appears to be. In Java, finalizers may be run on any thread and without any synchronization. Without the happens-before here, there would be zero guarantee as to the values of any of the object's non-final fields[1]. Because of the

[1]Final fields are an exception, guaranteeing some visibility outside the normal happens-before rules

happens-before, at least writes performed during the constructor are visible. Any changes after that may or may not be visible during finalization.

2.7 Concurrency Related Problems

2.7.1 Deadlocks

Rather than a problem related to the memory model, deadlocks are caused by erroneous nested locking. A deadlock happens when multiple threads are in a cyclic wait that will never end.

ThreadA

```
synchronized (L) {
  ...
  //Blocks here until B is available
  synchronized (M) {
    ...
  }
}
```

ThreadB

```
synchronized (M) {
  ...
  //Blocks here until A is available
  synchronized (L) {
    ...
  }
}
```

Table 2.6: Potential deadlock

It's possible here for thread A take take lock L, while thread B takes lock M. Eventually, thread A will try to take lock M, but block since B is still holding it and thread B will try to take lock L which is still in use by thread A. Since neither thread can back up out of the situation, this mutual wait will never end.

Whether or not a deadlock occurs often depends on the specific order in which threads are scheduled. Even when two threads could potentially end up in a deadlocked state, the required situation where both threads hold one of the locks may be exceedingly rare in practice. Things have to happen with a specific timing so one thread doesn't just take both locks and continues past the problematic section.

Since deadlocks can only occur when a thread blocks while holding a lock, avoiding nested locking in general can be a good idea. In some cases though, nested synchronization simply cannot be avoided. As long as the nested locks are always taken in the same order, there's no risk of a deadlock – only when different locking orders exist in the same program it becomes truly dangerous. The presence of different orders alone isn't necessarily a problem since surrounding code may make it impossible for a deadlock to occur.

Not as common, but still a very real possibility are deadlocks involving more than two threads.2.7.1 Any number of threads waiting to acquire a monitor can form a chain of dependent waits which can be represented as a directed graph. As long as there's a cycle in this dependency graph, all threads involved in the cycle will be stuck in a deadlock, endlessly waiting for each other.

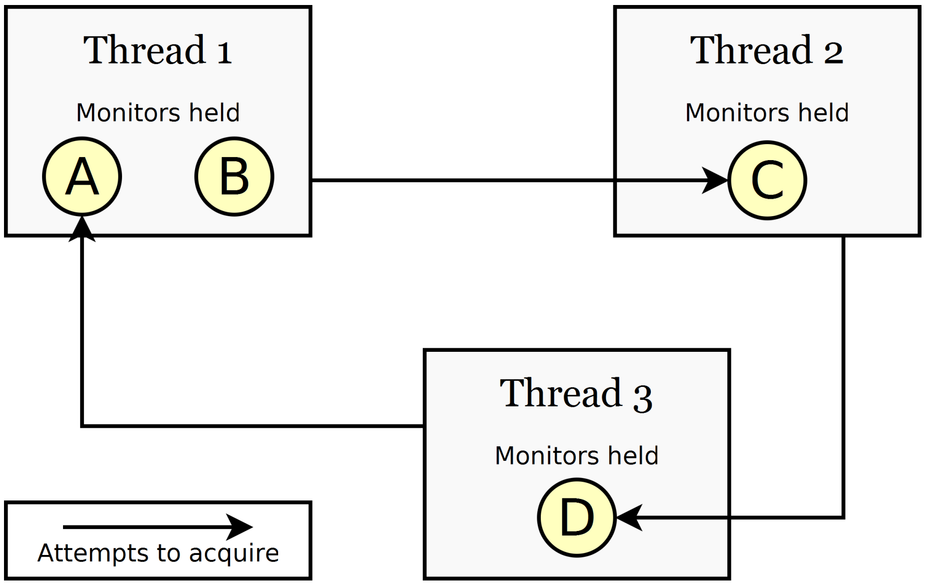


Figure 2.1: Example of a deadlock situation

2.7.2 Data Races

While the occurrence of a deadlock is often fatal to a running program, finding the cause is often relatively straightforward given the stack traces of the threads involved. Data races however can cause much more subtle problems, many of which can be dependent on the implementation of the virtual machine or even the underlying hardware. A *data race* in Java is defined by the language specification as follows.

> When a program contains two conflicting accesses that are not ordered by a happens-before relationship, it is said to contain a data race.[11]

Two accesses are *conflicting* when performed by different threads and at least one of the accesses is a write. Accessing a field through a data race creates a risk of witnessing implementation-specific optimizations allowed within the bounds of the happens-before rules. Often, this merely results in observing an outdated cached value for a field, but the consequences can be much more severe.

A data race free program on the other hand will appear sequentially consistent. Because all inter-thread behavior is ordered by happens-before rules (or order irrelevant as is the case with two concurrent reads), there's no opportunity to be influenced in any way by the types of optimizations and reorderings performed by the compiler. Perhaps most importantly, this means the behavior of the program (under the same thread scheduling) is guaranteed by the language semantics and not at the mercy of implementation details or other random factors.

The common methods of ensuring a field isn't involved in a data race are through use of the synchronized statement or by marking the field volatile. Code accessing a non-volatile shared field with no or inconsistent synchronization (using different monitors) is suspicious, but not necessarily incorrect. A data race situation may be impossible due to an innocent looking read/write of a volatile field guaranteeing a happens-before. Detecting real data races is a major implementation concern as they can be the cause for all kinds of strange behavior.

Unoptimized

```java
class T extends Thread {
  boolean stop = false;

  public void run() {
    while (!stop) {
      ...
    }
  }
}
```

Optimized

```java
class T extends Thread {
  boolean stop = false;

  public void run() {
    //Valid optimization!
    if (stop) return;
    while (true) {
      ...
    }
  }
}
```

Table 2.7: Infinite loop caused by a data race

Observing an older/cached value for a field involved in a data race may not seem a major problem. Remember though, unless there's a happens-before between the reading and writing threads later, the reading thread may never see the updated value. Probably the clearest example illustrating this principle is shown in 2.7. This example shows how, in the presence of a data race, simple compiler optimizations may result in serious problems. Here, the non-volatile field `stop` is judged to never change within the body of the while loop, allowing the check to be moved outside the loop condition. Another thread may be able to access `stop` and set it to `true` later, but because the field isn't volatile there's no happens-before and the compiler is allowed to ignore that possibility. An unintended infinite loop like this would normally be fairly easy to detect, but remember that the optimization shown here isn't consistently applied between different JVM implementations or sometimes even different runs of the same program.

Non-atomic double and long

Data races involving double/long fields are even more dangerous. Non-volatile writes to these fields aren't guaranteed to be atomic like other writes.[9] Reading such a field through a data race may not only result in an outdated/reordered value being seen, it's even possible to see a half-updated value. For example reading double with its high 32 bits matching one write and its low 32 bits matching another. This could result in a value being seen that was never written to that field anywhere in the program.

Chapter 3

Tackling the Thread Scheduler

The first step in detection of concurrency related errors is to remove non-determinism from the thread scheduler. Even ignoring for a moment the desire to guide the thread scheduler intelligently, it's important for testing to be able to repeat a run of the program with the exact same thread scheduling decisions. Otherwise it becomes exceedingly hard to properly debug errors that only occur under a specific interleaving of threads. Repeatedly running the same test with a non-deterministic scheduler can help increase coverage, but even then certain problematic code paths may just not be triggered with sufficient likelihood to make that a viable option.

The most straightforward way of overriding the thread scheduling algorithm would be to modify the JVM implementation itself, but doing so would require rather complex changes as modern JVMs typically don't perform their own thread scheduling anymore. Instead, each Java thread is backed by an operating system thread, leaving the scheduling decisions to be handled by the underlying operating system. Even ignoring for a moment the difficulties of making such a modification, building a testing framework on top of a specially customized JVM implementation is less than ideal. The customizations would need to be kept updated with changes to the base JVM and forcing users to switch to a different JVM in order to do threading-related tests is inconvenient at least. Fortunately though, it's possible to manipulate the thread scheduling without touching the JVM, merely by cleverly injecting code into a standard Java program.[4]

The core idea is to make sure only one thread is capable of running at any one time. No matter what algorithm the operating system uses to select which thread to run, if there's only one eligible thread to choose from, it must select that thread. The selected thread will then become the *active thread* and runs until it reaches one of the predetermined *switch points*; the only place where the active thread is allowed to change. These restrictions basically limit the instrumented program to behave as though it were running on a single-core processor with cooperative multitasking.

Some care must be taken with the implementation of the thread selection algorithm to ensure its results are actually repeatable. Mainly, the selection algorithm shouldn't rely on external information for its decisions. Anything that changes between runs of the program such as for example the current system time will result in a thread selection algorithm that still won't be repeatable as the it relies on a changing external state. Using a pseudo random number generator is fine as long as its initial seed value is either a constant or at least calculated in a deterministic way – the default constructor of `java.util.Random` typically uses the system time is therefore unsuited for use within the thread selection code. When scheduling decisions are based solely on the algorithm's inputs and not any changing external state, no complex behind-the-scenes bookkeeping is required to replay a specific thread schedule. When the tested program behaves in a deterministic manner (as unit tests should) and the scheduling algorithm does so too, re-running the program with the same inputs will always result in the same scheduling decisions.

3.1 Switch Points

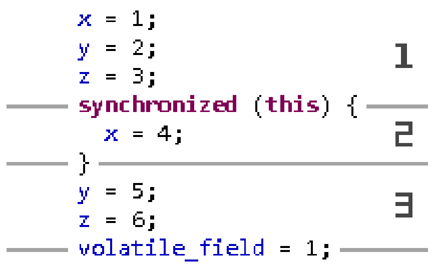
The instrumentation inserts calls to a `switchPoint` function at every point where a thread switch should be able to occur. Inside this function, the currently running thread is put into a wait state and a user-overridable thread scheduling algorithm selects which of the potentially runnable threads should become the new active thread. Once selected, the new active thread is woken up and runs until it either completes or reaches another switch point.

Compared to allowing the active thread to switch at any time, limiting thread switches to explicit switch points only has a number of benefits. The reduction in thread switching helps keep thread switching overhead under control. This is especially important when arbitrarily complex algorithms may be used to select which thread to run. Furthermore, even a small reduction in possible switch points is accompanied by a massive decrease in the number of possible thread schedules for the instrumented program.

That said, arbitrarily limiting the places where thread switches may occur risks hiding the existence of certain problematic thread schedules. There is however, a safe way of increasing the granularity of switch points without negatively affecting the ability to detect concurrency errors.

3.2 Granularity of Switch Points

The placement of switch points within the instrumented program is governed by the rules laid out in the Java memory model. In order to be able to detect data races later, switch points are inserted wherever an action mentioned in the happens-before rules is performed. Switching the active thread not at every read/write to a shared field, but only at happens-before events is safe with respect to the ability to detect data races. When switch points are inserted at each action associated with a happens-before, the sections of code in-between two adjacent switch points cannot contain any happens-before events. The definition of a data race2.7.2 requires the absence of a happens-before between two conflicting accesses. So as long as no new happens-before orderings are introduced by a section of code, nothing can 'solve' an existing data race and the number of data races in a running program can only increase. This effectively means data race detection can safely be postponed until right before a happens-before action is performed, with no other switch points necessary.



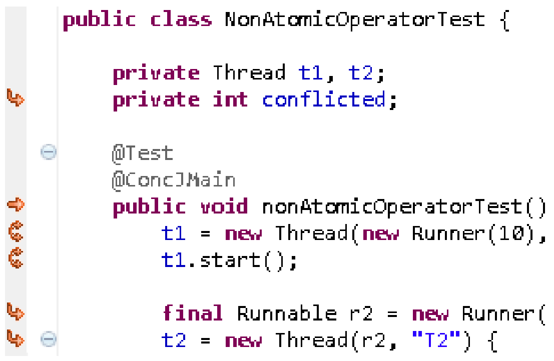
```
x = 1;
y = 2;
z = 3;
synchronized (this) {
  x = 4;
}
y = 5;
z = 6;
volatile_field = 1;
```

Figure 3.1: Segmentation of code between switch points

A few additional switch points are inserted by the framework to assist with thread management. Newly created threads must be put into an immediate wait state in order to prevent two threads from running at the same time (the new thread and its parent thread). The framework also inserts switch points before each untimed wait in the instrumented program, be it an attempt to acquire a monitor, a call to `Object.wait()` or any other. Whenever a switch point is inserted right before an attempt to acquire a monitor, this *pending monitor* is tracked by the framework and used for deadlock detection.

3.3 Implementation

Although the basic theory behind the thread scheduling code is fairly straightforward, a few practical problems do turn up during implementation. To start with, the main thread of the instrumented program is created by the JVM without any opportunity for the framework to involve itself in the process. Because the framework needs to keep track of thread lifetimes, it normally adds instrumentation before the first and after the last action of each created thread. Since the main thread is created by the JVM, it requires special bootstrapping code for the framework to start tracking its behavior while it's already running. This is done either manually through a call to `ThreadManager.registerInitialThread()`, or by marking the instrumented program's main method with a special `ConcJMain` annotation. Methods marked with that annotation automatically register the main thread when entering the annotated method, and unregister the thread upon leaving it. The framework uses an explicit annotation rather than simply pattern matching on `static void main(String[])` in order to support other types of entry points such as those used by applets (`init`) or various testing frameworks. When using JUnit with the provided test runner (`ConcJRunner`), the `@Test` annotation is assumed to automatically imply `@ConcJMain` and no explicit annotation is necessary.



```
public class NonAtomicOperatorTest {

    private Thread t1, t2;
    private int conflicted;

    @Test
    @ConcJMain
    public void nonAtomicOperatorTest()
        t1 = new Thread(new Runner(10),
        t1.start();

        final Runnable r2 = new Runner(
        t2 = new Thread(r2, "T2") {
```

Figure 3.2: JUnit test with AspectJ instrumentation markers

`ThreadManager` is responsible for tracking the metadata associated with each thread and maintains a set of living threads managed by the framework. Instrumentation of threads other than the initial thread happens at the thread creation site. The `Runnable` object passed to a thread's constructor is wrapped such that ThreadManager methods `onThreadStart()` and `onThreadStop()` are called immediately before and after calling the `run()` method of the user-supplied `Runnable`. When `Thread` itself is subclassed in the instrumented code, the instrumentation further subclasses to insert calls to the start/stop methods directly inside `Thread.run()`. The `onThreadStart()` method is responsible for putting newly created threads in a wait state upon creation to prevent a newly created thread from running immediately before becoming the active thread. `onThreadStop` mainly does some bookkeeping, removing the stopping thread from the set of living threads maintained by the thread manager. It also triggers the thread scheduling code to run, since the currently active thread is ending.

As mentioned previously, all instrumented threads get some metadata associated with them, stored by the thread manager. These associated `ThreadState` objects store for example a stack of all monitors currently acquired by that thread, as well as various other information required for analysis (data race detection in particular). The thread state also contains a kind of serial code for each thread to facilitate matching the data collected for a thread over multiple runs of the instrumented program. The serial code consists of the filename and line number where the thread object was created, combined with a monotonically increasing counter counting separately for each unique filename:line combination. The extra counter is there to handle the case where multiple threads are created from the same line of source code, either because the thread is created from a method which is called multiple times, or because threads are created in a loop.

The actual thread scheduling is handled by a dedicated background thread. When the active thread in the instrumented program goes into a wait at a switch point or terminates, it wakes up the thread scheduler. At the point in time when the thread scheduling algorithm runs, all monitored threads will be in a wait state inside a switch point. This allows the algorithm to safely access any metadata associated with those threads as the wait/notify used to implement the wakeup establishes a happens-before. The metadata is used by the thread scheduling algorithm to make a more informed decision as to which thread to run next.

3.3.1 Selection

The algorithm used to choose which thread to schedule at any given time usually relies on some sort of heuristic to be able to make an intelligent decision. No matter which algorithm is used however, not every thread can always be scheduled safely. Depending on the circumstances, trying to schedule a specific thread may only result in an immediate deadlock.

Pseudo code for checking if a thread is runnable

```
boolean isValid(Thread thread)
  //Unless the allowDisabled flag is set,
  //don't wake up 'disabled' threads.
  //Ex: In a wait() and no notify() has been issued yet.
  if (!allowDisabled && thread.isDisabled()) {
    return false;
  }

  //Don't wake up a thread requiring an monitor
  //currently in use by some other thread.
  if (isLockBlocked(thread, thread.getPendingLock())) {
    return false;
  }

  //Don't wake up threads waiting on a monitor currently
  //in use by another thread.
  if (isLockBlocked(thread, thread.getWaitLock())) {
    return false;
  }

  return true;
}
```

The above pseudo code forms the backbone of the default thread scheduler implementation. The `isLockBlocked()` method checks if a specific monitor is currently in use by another thread (remember that `Object.wait()` can temporarily free a monitor for use by other threads.) The `isValid()` method checks for three separate reasons why it might want to avoid scheduling a specific thread. These reasons are the following:

1 The first reason is not a hard blocker – waking up a sleeping thread doesn't cause any problems other than a drop in performance. This check is there to prevent the scheduler from interrupting any threads currently in a `wait()`. Under normal conditions, threads would only wake up from a `wait()` once a `notify()` or `notifyAll()` is issued, but JVM implementations are permitted to implement `wait()` in such a way that it could spontaneously wake up for whatever reason. If calls to `wait()` are properly handled in the instrumented program to account for this behavior, waking up sleeping threads in this way is pointless and a performance drain. On the other hand, if some instances where `wait()` is used do not take the possibility of spontaneous wakeups into account, intentionally causing these wake ups may trigger some latent implementation bugs. Since there's a cost/benefit analysis involved, the behavior of the thread scheduler is dependent upon a configuration parameter (`allowDisabled` in the pseudo code).

2 The second check is there to avoid scheduling a thread that requires a currently unavailable monitor. Switch points are inserted in the instrumented program right before any attempt to acquire a monitor, segmenting the code into runnable chunks that acquire at most one monitor when run. This *pending monitor* is tracked in the thread's metadata, allowing the thread scheduler to know in advance which monitor a thread will try to obtain if scheduled to run. Attempting to run a thread that will just immediately block because it requires a monitor currently in use by another thread is something that should be avoided.

3 When a thread goes to sleep as a result of a call to `Object.wait()`, it temporarily releases the monitor associated with the object on which the wait method was called for the duration of the wait. What that means for the implementation of the thread scheduler is that when a thread is woken up from a wait, the first thing it'll try to do is reacquire its monitor. This is akin to the issue with pending monitors – waking up a thread just to have it immediately block is pointless.

3.3.2 `wait`

Calls to `Object.wait()` in the instrumented program require some additional bookkeeping to track the pending monitor and such. Rather than adding instrumentation around the existing function call, the whole thing is just replaced with a drop-in substitute wait function that calls `switchPoint()` internally.

Simplified implementation of the substitute wait function

```java
void wait(Object obj, long millis)
    throws InterruptedException
{
  onEnterWait(currentThread, obj);
  try {
    SwitchPointMode waitMode = WAIT;
    if (millis > 0) waitMode = TIMED_WAIT;

    switchPoint(obj, waitMode);
  } finally {
    onExitWait(currentThread, obj);
    if (Thread.interrupted()) {
      throw new InterruptedException();
    }
  }
}
```

The replacement wait function calls `onEnterWait()` and `onExitWait()` to allow the framework to keep track of which thread is waiting on which monitor. It also handles thread interrupts by throwing an exception just like the regular `Object.wait()`. The interesting bits are inside the try block. First, note the wait time is disregarded outside of changing the switch point mode. Second, `switchPoint()` is called passing in `obj` as a parameter, resulting in an eventual call to `obj.wait()` inside the switch point function. Other switch points can just use some private monitor created by the thread manager, but when replacing a call to `obj.wait()` in the instrumented program that's not possible. Since a call to `obj.wait()` will temporarily release the monitor associated with `obj` for the duration of the wait, the only way to retain that behavior is to make the wait inside `switchPoint` affect the same monitor.

3.3.3 `switchPoint`

The implementation of the switch point function essentially boils down to calling `Object.wait()` in loop until the point in time when the waiting thread is again chosen by the thread scheduler to run. The condition must be checked in a loop in order to account for the possibility of a spurious wakeup from the wait call.

Pseudo code for the switch point function

```java
void switchPoint(Object monitor, SwitchPointMode mode) {
  Thread currentThread = Thread.currentThread();

  synchronized (monitor) {
    synchronized (this) {
      currentThread.setWaitLock(monitor, mode);
      if (mode != SwitchPointMode.REGISTER_THREAD) {
        threadRunner.poke(); //Wake up thread scheduler
      }
    }

    boolean interruptedFlag = false;
    while (currentThread.isWaiting()) {
      try {
        monitor.wait();
      } catch (InterruptedException e) {
        interruptedFlag = true;
        //Interrupts wake up the thread
        currentThread.onNotify();
      }
    }
    if (interruptedFlag) {
      //Need to restore the thread's interrupted flag
      currentThread.interrupt();
    }
  }
}
```

Some care was taken not to disturb the normal interrupt behavior of threads. The inner call to `wait()` may throw an `InterruptedException`, either from an explicit interrupt inside the instrumented program or by the instrumentation itself (if it's configured to generate spurious wakeups of sleeping threads). When that happens, `wait()` has detected that the thread has been interrupted by consuming the thread's interrupted flag. Switch points shouldn't influence program behavior and are therefore required to catch that exception and restore the interrupted flag.

The second point of attention is the presence of a `SwitchPointMode` parameter. This enum is used to keep track of the cause that led to insertion of a switch point. In the current implementation, the following switch point modes are tracked:

NORMAL The default. Corresponds to no reason specified or a switch point manually called from the instrumented code.

REG_THREAD Signifies the current switch point is the one inserted at the beginning of each new thread, the one intended to prevent two threads from running at the same time. This is the only type of switch point that does not call `threadRunner.poke()` to wake up the thread scheduler. In all other cases, the call to `switchPoint()` will be performed while that thread is the active thread, but `REG_THREAD` switch points are inserted at the beginning of newly spawned threads specifically to prevent them from running before they become the active thread. Waking up the thread scheduler in that situation could result in the thread scheduler running while another thread is still active.

TIMED_WAIT Indicates a timed wait such as `Thread.sleep()` or `Object.wait()` with a non-zero wait time. The actual wait time is normally ignored by the instrumentation as it does not influence the possible range of behaviors for the instrumented program – there are no guarantees for the accuracy of timed waits, so the actual wait time may be anywhere between zero and the end of time.

PARK Inserted due to a call to `LockSupport.park()` which is an alternative way to block a thread, introduced in Java 5. It behaves similarly to `wait()` in the sense that the blocked thread may spontaneously wake up, although the use of `LockSupport` doesn't involve any monitors.

Chapter 4

Error Detection

Controlling the thread scheduler allows for tests that are repeatable and the overridable scheduling algorithm makes it much easier to get broad coverage of all possible thread interleavings, bypassing the biases of the thread scheduler provided by the operating system. While those properties are certainly beneficial if not critical for thorough testing of a multi-threaded program, they are meaningless without the accompanying capability to detect deadlocks and particularly data races that occur under the thread interleavings produced by the custom scheduler.

4.1 Deadlocks

Deadlocks are typically not that hard to detect even without any instrumentation. Not too many programs continue to function properly when several of their threads permanently stall in a deadlock. On one hand, this makes deadlocks easy to detect when they turn up, but on the other hand it also means that even a single deadlock in production code can be a critical issue. Early detection, preferably during testing, is greatly preferential to catastrophic failure later.

The basis for deadlock detection within a running program starts with the definition of a deadlock. In the general sense, a deadlock is an unresolvable cyclical dependency between two or more threads in a mutual wait for each other's 'resources'. In our specific case, these resources are the monitors held by each thread, with the possibly indefinite wait occurring whenever a thread tries to acquire a monitor still in use by another thread. The reason such a wait may become unresolvable is the inability of threads to temporarily release a monitor they've acquired outside of calls to `Object.wait()`. This can create the situation where a chain of threads are all waiting for each other's monitor, resulting in an permanent stall: a deadlock.

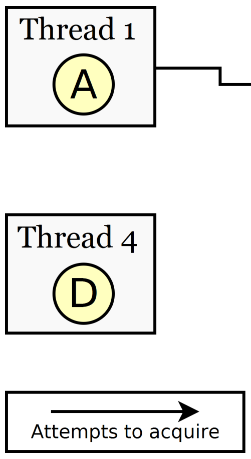


Figure 4.1: One step away from deadlock

The example situation displayed in figure 4.1 shows four threads, each holding a monitor. Threads 1 and 2 are blocked, waiting to acquire monitors currently held by other threads. In this situation, there's no deadlock (yet) as it's still possible for thread 3 to progress and maybe release monitor C.

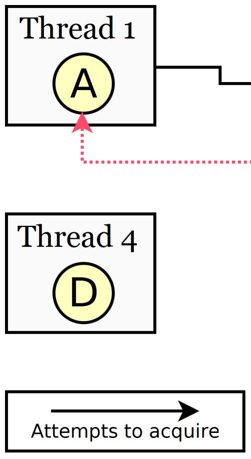


Figure 4.2: The dotted dependency creates a deadlock

It becomes problematic when thread 3 also tries to acquire another monitor (A) before releasing the one it's already holding (C). In this case, the additional connection of T3→T1 creates a circular dependency (T3→T1→T2→T3) and therefore a deadlock. Representing the threads in a directed graph like this makes any deadlocks very apparent. Any cycles that show up in the graph represent a circular wait: a deadlock. The implementation of the deadlock checker is based on that idea. It traverses the dependency graph whenever an edge is added to it and checks for cycles. Whenever any thread is capable of reaching itself through a traversal of the pending locks graph, that thread is part of a wait cycle and therefore involved in a deadlock.

4.2 Data Races

Where the effects of deadlocks are obvious and severe, the presence of data races in a program can have a much more subtle effect. For one, depending on the JVM implementation, the negative effects of a data race may never even show up during testing. A big problem with data races is just how well they hide themselves until at some point, possibly years in the future, some implementation detail changes somewhere and a formerly harmless data race suddenly triggers a major problem. Without instrumentation to specifically check for data races, it can be hard to have any sort of confidence a particular program is data race free.

In contrast to the straightforward implementation of the deadlock detection, accurately identifying data races is rather complex. Data race detection requires finding violations of the happens-before order2.7.2 as defined by the Java memory model. Inevitably, that means somehow keeping track of the effects of each and every happens-before for every thread.

It's important to find a good abstraction capable of simplifying a concrete Java program to the minimal parts necessary to do data race detection. Let's say for example we have the following program.

Thread A

```java
class A extends Thread {
  volatile int shared;

  public void run() {
    int n = 1;
    while (n <= 3) {
      shared = n;
      n++;
    }
    shared = 42;
  }
}
```

Events A

```
1 [E] Thread start
2 [D] shared = 1
3 [D] shared = 2
4 [D] shared = 3
5 [D] shared = 42;
6 [G] Thread stop
```

Table 4.1: Happens-before action numbering

Abstractly, Thread A can be viewed as performing a sequence of happens-before events. In this example, the thread performs several writes to a volatile variable (which is subject to happens-before rule D). By sequentially numbering each of these, it becomes possible to reason about which information becomes available when a happens-before relation is introduced between this thread and another one. Let's say a another thread B reads the `shared` field in-between A_2 and A_3 (thread A, events 2 and 3). This volatile read would introduce a happens-before between the two threads, guaranteeing visibility in thread B of all actions performed by thread A up to and including A_2.

In order to detect happens-before violations, each thread needs to keep track of the happens-before relations it has observed between itself and other threads. Storing each individual observed happens-before rule would quickly grow out of hand in any program with a lot of interaction between threads. Fortunately, that isn't necessary. Recall the following two happens-before rules.

A If two actions, x and y, are in the same thread and x precedes y in program order, then x happens-before y.

I If x happens-before y and y happens-before z, x happens-before z (happens-before is transitive).

Based on these, we can infer:

$$\forall x \geq 1, \forall y \geq 0\text{: happens-before}(\mathrm{A}_x,\ \mathrm{B}_y) \rightarrow \text{happens-before}(\mathrm{A}_{x-1},\ \mathrm{B}_y)$$

And because of that, threads only needs to keep track of a single happens-before for each other thread. For example, if a thread B knows `happens-before(`A_3`, `B_1`)` there's no need to separately remember a `happens-before(`A_2`, `B_1`)` since the second happens-before can be inferred from the first.

What we're left with then, is that each thread needs to keep a monotonically increasing counter to number its own inter-thread actions (the ones that appear on either side of a happens-before), combined with known event counters for each other thread that have been observed through a happens-before. Interestingly, that set up exactly matches an algorithm used to track causality within distributed systems: vector clocks.[12][13][14]

4.2.1 Vector Clocks

Vector clocks are a technique for detecting causality violations between asynchronous nodes/processes in a distributed system. As it so happens, communication between threads in Java behaves almost identically to the abstraction on which the vector clocks algorithm is based. The algorithm assumes each node runs independently, synchronizing with other nodes only by passing and receiving explicit messages. These messages act in much the same way as the happens-before rules in the Java memory model. Receiving a message guarantees visibility of all actions performed by the sending node prior to sending the message. By associating all inter-thread actions in Java with corresponding vector clock message passing, happens-before violations can be detected by comparing vector clocks.

The core idea behind vector clocks is the event numbering mentioned in the previous chapter. To put things in Java terminology, each thread maintains a

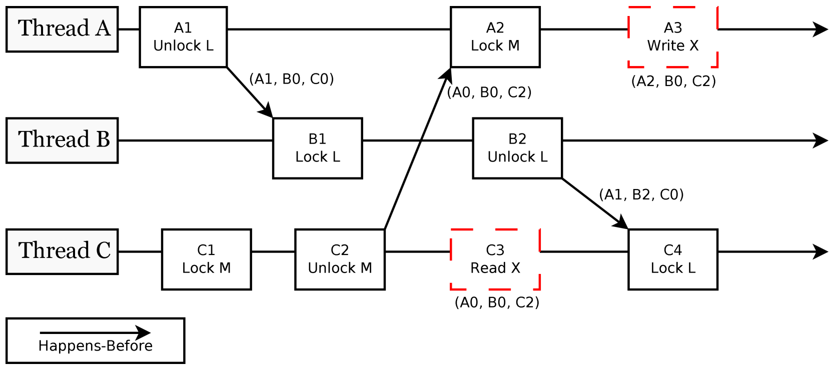


Figure 4.3: Happens-before ordering in a multi-threaded program (A3 and C3 are concurrent)

monotonically increasing timestamp to number its events. This timestamp isn't based on any real clock, but rather behaves more like an event counter providing only an abstract concept of time. Aside from its own (local) timestamp, each thread also maintains additional timestamps for each other thread it can communicate with, representing the minimum guaranteed observable events from other threads. A thread's own timestamp, together with observed timestamps for other threads form a vector clock. Messages exchanged between threads consist of a copy of the sender's vector of timestamps. When a thread receives a message, it updates its timestamps with the values in the message.

Vector Clock Operations

Each vector clock maintains a vector of timestamps, one of which is the *local timestamp* which is used as an event counter. The other timestamps are received directly or indirectly by communicating with other vector clocks. Communication happens by exchanging *messages*, which can be either regular vector clocks, or *anonymous vector clocks* as returned by `peek` or `send`. Anonymous vector clocks lack a local timestamp.

event(VC)	Increments the vector clock's own local timestamp. Undefined on anonymous vector clocks.
peek(VC)	Returns an anonymous copy of the given vector clock.
send(VC)	The basic vector clock message sending operation. Implemented as an `event` followed by a `peek`.
fork(VC)	Creates a new vector clock initialized with the timestamps of `VC` and a new local timestamp.
join(VC, MSG)	Sets each timestamp of `VC` to `max(VC.stamp[i], MSG.stamp[i])`.

receive(VC, MSG) The basic vector clock message receiving operation. Implemented as a `join` followed by a `event`.

leq(VC1, VC2) Compares two vector clocks. Returns true if each of the timestamps in `VC1` are less than or equal to the corresponding timestamps in `VC2`. If some of the timestamps are less than or equal to those in `VC2`, but not all of them, the two vector clocks are said to be *concurrent*.

Modeling happens-before with Vector Clocks

Although the concept of modeling happens-before by exchanging messages between threads is fairly straightforward, the practical implementation has some minor complications. Happens-before relations don't always happen in nice one-to-one configurations where one event in thread A connects with another event in thread B. It's equally likely for a single event (for example unlocking a monitor) to be connected to several events in a number of different threads. Likewise, although the graphical representation of vector clock message passing shows nice direct arrows between threads, when they are used to model happens-before in Java, most message passing is actually indirect.

Say for example we have a happens-before between a monitor unlock (thread A) and a subsequent monitor lock (thread B). The message that is received by thread B should be based on the vector clock of thread A *at the time thread A unlocked the monitor*. The only way to make sure such a copy is available is to always create one whenever a thread unlocks a monitor, then store the copy somewhere so it can be retrieved if the unlock event turns out to be involved in a happens-before at some later time.

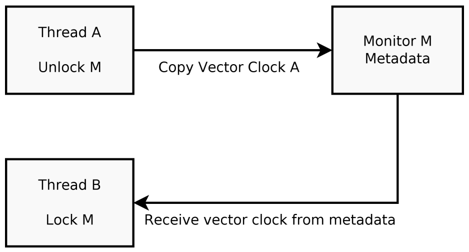


Figure 4.4: Indirect message passing between two threads

One final issue with storing vector clocks in metadata is how to handle the situation where multiple threads need to store concurrent vector clocks in the same location. If it were only a single thread or none of the clocks involved were concurrent, the stored clock could simply be overwritten. When (some of) the clocks are concurrent however, that would lead to loss of information. Separately storing each of the concurrent vector clocks is one solution, but a much better one is to merge all of them into a single message.

Say we have have two messages, A and B, that would need to be received at the same time by some thread with a vector clock C. Receiving those messages separately would result in:

$$receive(receive(C, A), B)$$

Expanding `receive` with its definition results in the equivalent:

$$event(join(event(join(C, A)), B)).$$

The inner `event` can safely be moved to the outside. All it does is increment the local timestamp of C, which we know will always be greater than or equal to the values for that timestamp found in A and B (their value is a copy). Since `join` takes the maximum of each timestamp, we know the local timestamp of C will not be changed by either instance of `join`. Therefore, the inner event can be moved to the outside of the expression, resulting in:

$$event(event(join(join(C, A), B)))$$

Since `join` is associative, that can be rewritten to:

$$event(event(join(C, join(A, B)))$$

Because C no longer has to join with A and B separately, it suffices to store only the result of $join(A, B)$, thus allowing multiple messages to be merged together.

What is left is to ensure each and every happens-before rule is correctly modeled using vector clock message passing. In most cases it's as simple as finding a good place to store the intermediate vector clock for indirect message passing, but several rules make references to vaguely defined high-level concepts such as detecting when a thread has terminated which could use some further elaboration.

A *If two actions, x and y, are in the same thread and x precedes y in program order, then x happens-before y.*
This rule is largely handled just by the inherent nature of vector clocks. Because the timestamps of the vector clock associated with each thread can only increase over the course of the program, for any two events within that thread, the vector clock at the event occurring later in program order will compare $\geq$ to the vector clock at the earlier event.

B *Any action in an object's constructor happens-before any action in that object's finalizer.*
An important aspect to consider when using custom finalizers is that there are very little guarantees as to when and in which thread they will run. Finalizers are typically run in a separate thread, concurrent to the execution of the rest of the program. The happens-before between a constructor ending and that object's finalizer running can be modeled by storing a dedicated vector clock in the metadata associated with each object. This vector clock is then joined with a copy of the current thread's vector clock whenever a constructor ends. When some other thread executes that object's finalizer later on, the thread's vector clock is joined with the object's constructor clock to model the visibility guarantee proided by this happens-before rule.

C *An unlock action of any monitor happens-before all subsequent lock actions on that monitor by any thread.*
A separate vector clock is stored in the metadata associated with each monitor to pass along the message. Whenever a monitor unlock happens, the clock in the metadata is joined with a copy of the vector clock of the unlocking thread. Then, when a thread attempts to lock that same monitor later, it joins its own vector clock with a copy of the monitor's clock stored in the metadata.

D *A write to any volatile field happens-before all subsequent reads of that field by any thread.*
Similar to rule C, the framework stores a vector clock in the metadata associated with each instance of a volatile field.

E *An action that starts a thread happens-before the first action of the thread that was started by it.*
Each thread already needs a private vector clock to number its own events. That vector clock could be created from scratch and initialized with all zeros, but it would be just as easy to fork some existing vector clock instead. Like say, the vector clock of the thread calling `Thread.start()`. Doing so initializes each thread with a vector clock corresponding to a happens-before between it and the thread responsible for starting it.

F *Initialization of default values for fields (null, 0, false) happens-before the first action in every thread.*
This basically boils down to initializing the initial vector clock associated with each field to a value less than the initial vector clock of every monitored thread. Such a vector clock is easily obtained, namely the initial vector clock of the main thread. All other threads obtain their vector clock by forking their parents's, which means the clock of the main thread is the (grand) parent of the vector clocks of all other threads and its earliest value will be chronologically before all other clocks in the system.

G *The final action of a thread happens-before any action detecting that thread's termination.*
Because after the final action of a thread executes its vector clock will no longer change, we don't actually need to store a separate copy anywhere – just read the thread's vector clock directly when needed. Specifically, `Thread.isAlive()` and `Thread.join()`.[15]

H *An action interrupting a thread happens-before any action detecting that interrupt.*
Not often the preferred way of handling synchronization, but implementation-wise quite similar to rules C and D. A secondary vector clock is associated with each thread specifically to handle happens-before messages passed through interrupts. The one Java function capable of triggering an interrupt is `Thread.interrupt()` and only two methods can directly detect interrupts: `Thread.interrupted()` and `Thread.isInterrupted()`.

I *If x happens-before y and y happens-before z, x happens-before z (happens-before is transitive).*
As happens-before is determined by comparing vector clocks, and vector clock comparison is transitive, no explicit action is required to adhere to this rule.

Chapter 5

Implementation

5.1 JUnit

Given that unit tests are the prime candidates for instrumentation, the testing framework comes with some extra classes for easier JUnit integration. A rather annoying property of JUnit is that it ignores exceptions and assertion errors in all but the main thread. That is acceptable in mostly single-threaded tests, but rather annoying in the presence of heavy multi-threading. As a workaround for this behavior, uncaught exceptions from secondary threads are collected and rethrown at the end of each test case. This causes JUnit to mark tests with uncaught exceptions in any thread as a failure instead of a success.

A special test runner (`ConcJRunner`) is also provided that allows re-running the same test(s) repeatedly with different thread scheduling decisions. It will run each test for the specified number of repetitions or until the test fails, seeding the random number generator of the thread scheduler with different values each run. The test runner is based on the standard test runner and activated with the usual JUnit `@RunWith` annotation.

5.2 Metadata

In order to be able to detect concurrency errors, some per-object metadata must be maintained (vector clocks and such). There are two general approaches to storing this data. The first is to insert extra metadata fields directly into every single class in the instrumented program. The second alternative leaves the class files alone and uses an external hashmap, mapping objects to their corresponding metadata. Unfortunately, the first option doesn't work for classes that lie outside the control of AspectJ – namely classes belonging to Java's standard class library. The external hashmap approach works for all classes, even if it's at the cost of an extra hashmap lookup every time an object's metadata needs to be accessed.

The type of map used must be an `IdentityHashMap` or equivalent, since it needs to associate information with specific object instances, ignoring the result of `Object.equals()` which is normally used to determine key equality. However, in order to still allow normal garbage collection, the objects used as keys must also be weak-referenced. Using normal, strong references, would prevent garbage collection of every object in the metadata map.

Over the course of running an instrumented program, interesting events are logged to an xml-based log file. This event log contains thread switches, locks taken, data races, etc. that may be used for off-line analysis at some later time. Objects appearing in this log are named in a way that attempts to generate the same name for equivalent objects over multiple runs. Such a naming scheme makes it possible to deduce a range of possible behaviors for objects/threads/monitors by observing many runs of the same program.

5.3 Code Organization

One advantage of using AspectJ is that it allows the instrumentation code to be separated by concern. The instrumentation is split between different Aspects for objects, field accesses, locking and thread management. These Aspects define the behavior associated with events such as field writes or thread creation and invoke the appropriate global callbacks (`onFieldWrite()` and such).

The thread manager and other metadata are accessible through static methods in the `Globals` class. This class uses static fields internally to store references to the metadata objects. Although static fields are the most straightforward way of making data available to any number of threads, as a side effect it also limits the instrumentation to allowing one active instance per JVM. If running multiple instances of the instrumentation simultaneously within the same JVM is desired, the `Globals` object could be modified to use `ThreadLocal` fields instead and populate those for any newly registered thread. The use case for this seems far-fetched, so in the interest of simplicity, the current implementation just uses static fields.

Simple thread-safe logging facilities are provided that add the originating thread-id to each logged event. Knowing which thread is responsible for which message is critical when trying to make sense of the actions of complexly intertwined threads. The same logging facility is responsible for responding to any errors detected by the instrumentation. User-provided deadlock and data race handlers allow custom handling of these events instead of the default behavior (throwing an exception).

5.4 Arrays

Arrays require a bit of special care. As far as the memory model is concerned, each array element functions as though it were a separate field.[16] The instrumentation simply treats array elements as though they were numerically named fields of their parent array object, associating field metadata with every array index. Although array elements behave as fields, the syntax of the language doesn't allow them to be marked final or volatile in any way. It's a common mistake to mark an array as volatile, not realizing that the volatile keyword only applies to the field containing the array reference and not the elements of the array itself.

Array creation can be instrumented by AspectJ if the `-Xjoinpoints: arrayconstruction` flag is used. Unfortunately, accessing array elements doesn't trigger the normal field get/set pointcuts and no separate arrayget/arrayset pointcuts exist.[1] There has been research on creating an extension to add these missing pointcuts[17] but unfortunately this work targets AspectBench, an outdated[2] alternative implementation, rather than the far more up-to-date AspectJ reference compiler, *ajc*.

This current AspectJ limitations is worked around by adding an additional step to the build process. After the class files are compiled and instrumented, they are run through a second instrumentation step which uses the ASM[3] library to insert calls to static instrumentation methods for all instances of array get/set opcodes. These static methods then forward the array access event to the same data race detection code that regular field reads and writes go through.

5.5 Performance

In general, because there are so many reads and writes to fields, even a small improvement in the performance of that part of the instrumentation results in a significant improvement in runtime. Specifically, memory allocations in parts of the code related to field accesses should be avoided whenever possible. It's very easy to completely overwhelm the garbage collector with temporary objects, such as those created by AspectJ when `thisJoinPoint` is used.

Another potential performance bottleneck is the instantiation of field metadata. Some of the methods in `java.lang.Class` can be slow due to internal security manager checks. As a countermeasure, the instrumentation maintains a cache of the necessary information in order to speed up repeated initialization of fields of the same classes.

Because array elements are treated as fields, each index of an array requires a full field metadata object. For arrays with millions of elements, the overhead of this instrumentation can seriously degrade performance. As a trade-off between perfect accuracy and performance, a configurable property allows limiting the number of instrumented elements per array object, allowing for example to only instrument the first 100 elements of each array. If an array would then contain a data race only at index 237 for example, it wouldn't be detected.

[1] `https://bugs.eclipse.org/bugs/show_bug.cgi?id=157031`

[2] AspectBench doesn't support any Java 5 features such as annotations and generics.

[3] http://asm.ow2.org/

In cases where instrumentation overhead becomes excessive, as a last resort specific methods of the instrumented program can be excluded from instrumentation with the `ConcJExclude` annotation.

5.6 Problems and Limitations

5.6.1 Finalizers and Garbage Collection

Finalizers are hard to deal with. Due to the inherent non-deterministic nature of most garbage collection algorithms there are no guarantees when, or even if, the finalizer for an unreachable object will be executed. For finalizers that have an observable effect on other living objects, this means the behavior of the program depends on an unknown (the garbage collector) and can vary from run to run even with the exact same thread scheduling decisions. Thankfully, such finalizers are rare and most just clean up native resources without influencing other Java objects.

However, even finalizers that don't access other objects can be problematic. Since finalizers are typically executed from a separate finalizer thread, trying to access any field written to after the object's constructor returned (happens-before rule B) will be a data race. Detecting the data race works just as usual, but because there are no guarantees when exactly the finalizer will run, the detection will be unreliable and dependent on the whims of the garbage collector. In some cases, detecting data races in finalizers is more important than consistency, but in other cases exact repeatability of the events of a test program is considerably more important. Since there's no one good solution, finalizer behavior can be changed with a command-line flag to either skip or perform instrumentation of finalizers.

5.6.2 Uninstrumentable Code

Not all code can be instrumented by AspectJ. Native methods are an obvious example, given that their implementations don't even consist of Java code. Another limitation comes from the way in which the instrumentation is added. The AspectJ compiler modifies classes by changing them at the bytecode level and therefore can't instrument classes it doesn't have access to.[18] This includes the classes in the Java standard library such as `java.lang.String` and `java.lang.Thread`. In most cases this limitation can be worked around by instrumenting the call site (which lies in user code) rather than the internals of a class. Be aware that the internals of standard library classes lie outside the control of AspectJ's compile-time modifications.

Method calls and field accesses performed through Java's reflection mechanism aren't picked up by AspectJ either. In the case of method calls, pointcuts using `execution` will still trigger as those modify the internals of the method itself. The `call` pointcut on the other hand will not trigger for method calls made through reflection. It's technically possible to intercept calls to Java's reflection methods with AspectJ, check the parameters used and invoke the appropriate instrumentation methods, but doing so would significantly complicate the AspectJ code for a minor edge case.

5.6.3 Static Initializers

Static initializers in classes can be instrumented just fine, but they cause problems due to the time at which they are run. Say you have a class C which contains a main method annotated with `@ConcJMain`. When entering/leaving that method, the instrumentation automatically starts and stops. However, static initializers of C run before the main method as well as any static initializers in classes (indirectly) referenced by C. Because they run outside the timing window associated with the instrumentation of the annotated method, actions performed by those static initializers are ignored by the implementation. Generally this has no negative effects beyond the loss of filename+line information in the metadata of object instantiated during static initialization.

Mutable static fields can also be problematic if they're not reset to the same initial state each time the instrumented program is run. If those fields are only initialized at class-load time and change while running the program, re-running the same test several times in a row (for example to test different thread schedules) may produce different results since those fields will have different starting values for different runs. The test runner could use a custom ClassLoader and load/unload the tested classes for each run, but writing test cases such that they can be run more than once per JVM instance would be preferable regardless.

5.6.4 Compile-time Constants

Certain Java constructs are changed/removed during compilation. This may result in unexpected method calls, such as the creation and use of a temporary StringBuilder object in place of string concatenation using the + operator. In the same way, references of the form `SomeClass.class` are replaced by calls to `Class.forName()`.[19]

The main case where these types of optimizations may become noticeable is when the value of a final field is a compile time constant, allowing the compiler to optimize the field away and replace all references to it with its value. This can be observed by the absence of an AspectJ marker for accesses of that field in your IDE of choice.

5.6.5 Java 5 Concurrency Package

Java 5 introduced an extensive package of concurrent collections and synchronization primitives: `java.util.concurrent`. Many of these require explicit support in the testing framework to function correctly, either because they start threads or block inside methods that lie outside the control of AspectJ. Their implementations are required to conform to a set of happens-before rules as described in the concurrency package's javadoc.[4] Support for the concurrency primitives is currently not yet implemented.

[4]`http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html`

Chapter 6

Results

To measure the performance impact of the instrumentation, it was tested on the multi-thread version of the Java Grande Forum (JGF) benchmark v1.0 suite[1]. Although an improved 2.0 version is available, the older 1.0 version was used as it's known to contain several data races.[20]

Time measurements were obtained by running each test 10 times in a loop in order to smooth out the effects of random performance fluctuations and JVM warm-up time. Tests were run using their default settings, but with 4 concurrent threads instead of the default 1. The instrumentation also used its default configuration (tracking only the first 100 elements of each array).

Name (**+Dataset**)	Base	Instr.	Races	R	W	L	Switches
crypt	0.073	10.539	0	105M	0.5k	13	45
lufact	0.054	74.149	104	6.4M	2.7M	10	324399
series	1.840	5.950	0	20k	0.2k	10	30
sor	0.165	11.775	100	51M	11M	10	8946
sparsemult	0.140	49.005	0	419M	0.9M	10	30
moldyn (A)	0.581	288.044	201	1.3G	15M	16	32867
montecarlo (A)	0.589	50.759	1	303M	45M	16	42
raytracer (A)	0.563	440.511	5	3.1G	0.4G	32	342

Figure 6.1: Runtime Performance

The **Base** time (in seconds) is the time spent by each benchmark when running without any instrumentation.[2] **Instr.** is the time required by the same benchmark with all instrumentation turned on. Because the test system is a

[1]`http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/threads.html`

[2]System used: Core i5 750, 4GB RAM, Oracle Java 1.7.0_21-b11

quad-core, and the instrumentation limits to running only one thread at a time, there's already an effective 4× penalty for the multi-threaded parts of each test even before any other overhead. Additionally, the results table includes the approximate number of instrumented reads/writes (**R/W**) per run as well as the number of locks obtained (**L**) and the number of thread switches (**Switches**).

An issue with counting data races is that there are multiple ways to do it. Perhaps the simplest way is to just count the total number of racing field-accesses in a program run. The number of data races produced by this method has some correlation with the quality of the tested program and the likelihood of encountering a data race in practice. That is not what we're trying to measure however. What we want to do is measure the effectiveness of the instrumentation itself. A better measure of quality for that is to count only the number of distinct fields that can be detected to be in a data race. This can be done in two ways: count each field of each object separately, or count fields only once per class. While there's certainly a case to be made for a per-object count, the per-class count has the distinct advantage of allowing comparison with the results of static analysis tools.

Data races involving array elements are a special case – it doesn't make a whole lot of sense to say field `0` of class `double[]` is involved in a data race. Instead, arrays are treated much like anonymous inner classes. Each array creation site in the source code is considered a different virtual subtype for the purposes of counting data races. This method isn't perfect, consider for example a program that uses factory methods to create all of its arrays, but in general it produces a fairly good approximation of which array objects are semantically distinct from each other.

6.1 Detected Data Races

6.1.1 Group A - Volatile Array Fields

The common cause for most of the detected data races lies in the implementation of `TournamentBarrier.java` which attempts to implement a synchronization barrier similar to `java.util.concurrent.CyclicBarrier`. Unfortunately, its implementation contains a data race and therefore the barrier doesn't actually work as a method of synchronization. Specifically, it uses a volatile array field (`IsDone`), erroneously expecting the elements of that array to behave as volatile fields. The same mistake is made with `SOR.sync`, not realizing the volatile modifier applies only to the field containing the array reference and not the contents of the array itself.

6.1.2 Group B - Harmless Data Race

This specifically occurs with the static `UNIVERSAL_DEBUG` field in the montecarlo benchmark. Although the field accesses are improperly synchronized, in this

	Field	Count	Group
lufact	TournamentBarrier.IsDone[]	4	A
lufact	LinPack.a[][]	100*	D
sor	SOR.sync[][]	1	A
sor	SOR.SORrun(G)	99*	D
moldyn (A)	TournamentBarrier.IsDone[]	4	A
moldyn (A)	md.epot[]	3	D
moldyn (A)	md.vir[]	3	D
moldyn (A)	md.interacts[]	3	D
moldyn (A)	mdRunner.sh_force[][]	100*	D
moldyn (A)	mdRunner.sh_force2[][][]	88*	D
montecarlo (A)	Universal.UNIVERSAL_DEBUG	1	B
raytracer (A)	TournamentBarrier.IsDone[]	4	A
raytracer (A)	JGFRayTracerBench.checksum1	1	C

(*) Number of detected data races limited by only monitoring the first 100 array elements.

Figure 6.2: Data Races by Type

specific case the data race can't influence the results of the program. Different threads concurrently write to the same field, but they all attempt to set its value to `true`. Each thread may then see either the value written by some other thread (which will be `true`) or otherwise the value it wrote itself (which is also `true`).

6.1.3 Group C - Incorrectly Synchronized

Accessing a shared field exclusively from within synchronized blocks is an effective way of preventing data races... but only if the same monitor is used for every access. Modifications to the `checksum1` field in the raytracer benchmark occur within a synchronized block, but this offers no protection as each thread synchronizes on a different object.

6.1.4 Group D - Consequences of Other Errors

Several other errors are a direct consequence of the failing implementation of TournamentBarrier. The barrier is supposed to assure all threads have reached a certain point in their code, establishing a happens-before relation between those threads. Due to the implementation error in the barrier, no such happens-before is created and a data race occurs between the initialization and processing phases of the benchmarks.

6.1.5 Accuracy

Because the data race detection algorithm is a direct implementation of the rules laid out in the memory model, as long as all relevant happens-before events occur in instrumented code, no false positives can occur. On the other hand, only races that provably occur during a concrete execution of the instrumented program will be detected. Certain fields can only be proven to be involved in a data race under specific conditions. Some may depend on the program's input, some may depend on a particular set of thread scheduling decisions. When for example a thread performs an unsynchronized write to a shared field, another thread has to be capable of accessing that same field in a conflicting way for it to be classified as a data race. The system of vector clocks will detect all conflicting accesses that occur during a concrete run of the instrumented program, but not necessarily all conflicting accesses that *can* occur for other program inputs or thread scheduling decisions.

Thread A

```java
synchronized (M) {
  shared = 1;
}
shared = 2;
synchronized (M) {
  shared = 3;
}
```

Thread B

```java
synchronized (M) {
  System.out.println(shared);
}
```

Table 6.1: Example of a data race which is not always detected

The likelihood of finding a conflicting access depends on the nature of the instrumented program. Take for example a program where two threads access a shared field, where one of the writes to `shared` from Thread A lacks sufficient synchronization. The only way in which a conflicting access will occur is if Thread B is scheduled immediately following the unsynchronized write in Thread A. Once Thread A enters its second synchronized block and performs another write to `shared`, the effects of the 'dangerous' write earlier will be overwritten. Since the active thread may only change at switch points and the number of switch points in a thread is usually low, there's a still decent chance of stumbling into the required sequence of events even in this worst case scenario. If we were to omit the second synchronized block in Thread A, the window for a finding a conflicting access would be much larger: a data race will be detected if Thread B is scheduled at any time after Thread A leaves its synchronized block.

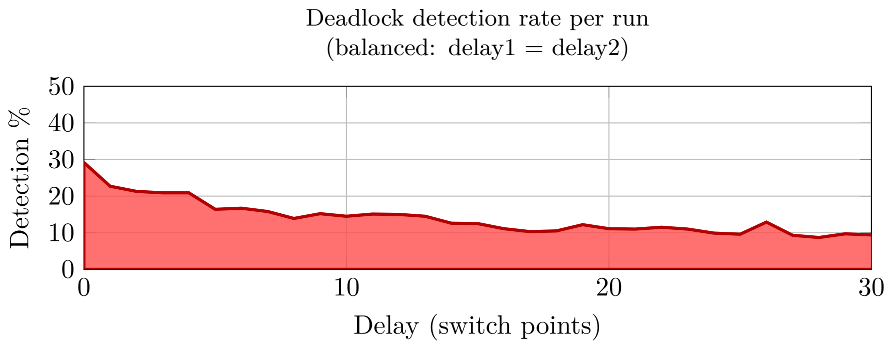
Detection of the data races found in the JGF benchmark is independent of the chosen thread schedule. Shared fields are typically accessed prior to any happens-before event within the multi-threaded parts, guaranteeing detection. The sole exception is a conflicting write to `checksum1`, which is also guaranteed to be found because no happens-before follows it.

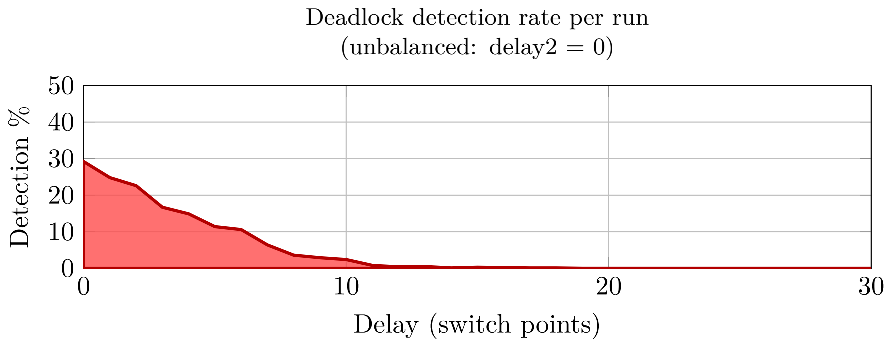
6.2 Deadlock Detection

The following test program was used to measure the likelihood of detecting a deadlock with the random thread scheduler.

Deadlock Test

```
void testAB(final int delay1, final int delay2)
    throws InterruptedException
{
  final Object a = new Object();
  final Object b = new Object();

  Thread t1 = new Thread() {
    public void run() {
      for (int n = 0; n < delay1; n++) {
        switchPoint();
      }
      synchronized(a) {
        synchronized (b) {
          //Lock a->b
        }
      }
    }
  };

  Thread t2 = new Thread() {
    public void run() {
      for (int n = 0; n < delay2; n++) {
        switchPoint();
      }
      synchronized(b) {
        synchronized (a) {
          //Lock b->a
        }
      }
    }
  };

  t1.start();
  t2.start();

  t1.join();
  t2.join();
}
```





This don't so much measure the accuracy of the deadlock detection algorithm itself (detecting a deadlock is easy), but rather chance stumbling into a deadlock situation. Compared to data races, deadlock detection is much more dependent on the thread scheduling algorithm used. For small test programs, or threads with little interaction (and thus few switch points), the random scheduler performs adequately. On the other hand, if in order to trigger a deadlock the thread scheduler has to pick the same thread 50 times in a row, the chances of that happening by chance become extremely low.

Chapter 7

Future Work

A random thread scheduler works fine for smaller test cases, but loses effectiveness when the goal is to trigger deadlocks in larger programs. For a deadlock to occur, multiple threads must be at very specific positions in the program at the same time. Data race detection seems less affected by this, as there's usually a much larger window where a race may be detected. In order to trigger deadlocks more reliably, a specially designed thread scheduling algorithm could be used. A simple strategy would be to avoid scheduling threads that are inside synchronized blocks in order to maximize the number of concurrent monitors held in order to increase the likelihood of finding a deadlock. More complex implementations could look at the event log for suspicious cases of nested locking and direct the thread scheduler towards attempting to trigger a specific deadlock.

In general, any attempt to guide execution towards triggering a deadlock would cause a bias towards which thread schedules are investigated at runtime. Such a bias could have negative consequences for the likelihood of finding data races. Specialization of the thread scheduling algorithm towards finding one class of errors necessitates separated testing for deadlocks and data races, each using a different thread scheduler.

Some practical limitations also affect the current implementation. The lack of an AspectJ pointcut for reads/writes of array elements requires what's basically an ugly hack to achieve the necessary instrumentation by running the compiled program through an additional instrumentation step using ASM. On the other hand, a lower-level method of instrumentation may be required to achieve better runtime performance, especially for field reads/writes. Retrieving field metadata and updating/comparing vector clocks dominate the runtime. Storing field metadata in a separate hashmap may be necessary for classes outside the control of AspectJ, but for classes that AspectJ *can* modify it would be possible modify the class definition to include extra fields and store references to the metadata directly inside the objects themselves. Another improvement would be the use static analysis as a pre-processing step to exclude from instrumentation those fields/objects that can be proven to be accessed by only one thread.[3]

Chapter 8

Conclusions

AspectJ-based instrumentation can reliably detect runtime concurrency errors for normal Java programs running on a stock JVM. Because the exact rules of the memory model are implemented, all detected errors are real. The exact conditions and thread scheduling decisions leading up to the detected error can be easily recreated thanks to the deterministic thread scheduler. The current implementation is accurate but prohibitively slow for more complex programs. Achieving satisfying performance for larger test cases will require further refinement of the way field reads/writes are instrumented to reduce the runtime overhead.

Bibliography

[1] "The aspectj™programming guide." `http://www.eclipse.org/aspectj/doc/next/progguide/`.

[2] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, *The Java™Language Specification, Java SE 7 Edition*, ch. 17.4, pp. 569–584. 2012.

[3] E. Bodden and K. Havelund, "Racer: Effective race detection using aspectj," Tech. Rep. abc Technical Report No. abc-2008-1, 2008.

[4] P. Joshi, M. Naik, C.-S. Park, and K. Sen, "Calfuzzer: An extensible active testing framework for concurrent programs," in *Proc. 21st International Conference on Computer Aided Verification (CAV'09)*, 2009.

[5] K. Sen and G. Agha, "Concolic testing of multithreaded programs and its application to testing security protocols," tech. rep., University of Illinois at Urbana Champaign, 2006.

[6] P. Eugster, "Java virtual machine with rollback procedure allowing systematic and exhaustive testing of multithreaded java programs," Master's thesis, ETH Zürich, 2003.

[7] D. L. Bruening, "Systematic testing of multithreaded java programs," Master's thesis, Massachusetts Institute of Technology, 1999.

[8] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," in *ACM Transactions on Computer Systems*, vol. Volume 15 Issue 4, 1997.

[9] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, *The Java™Language Specification, Java SE 7 Edition*, ch. 17.7, pp. 590–590. 2012.

[10] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, *The Java™Language Specification, Java SE 7 Edition*, ch. 17.4.5, pp. 575–578. 2012.

[11] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, *The Java™Language Specification, Java SE 7 Edition*, ch. 17.4.5, pp. 576–576. 2012.

[12] C. J. Fidge, "Timestamps in message-passing systems that preserve the partial ordering," *Australian Computer Science Communications*, vol. 10, pp. 56–66, 1988.

[13] F. Mattern, "Virtual time and global states of distributed systems," in *Parallel and Distributed Algorithms*, 1989.

[14] S. Burckhardt and M. Musuvathi, "Effective program verification for relaxed memory models," Tech. Rep. MSR-TR-2008-12, 2008.

[15] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, *The Java™Language Specification, Java SE 7 Edition*, ch. 17.4.4, pp. 574–575. 2012.

[16] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, *The Java™Language Specification, Java SE 7 Edition*, ch. 17.4.1, pp. 572–572. 2012.

[17] K. Chen and C.-H. Chien, "Extending the field access pointcuts of aspectj to arrays,"

[18] "The aspectj™programming guide, appendix c. implementation notes." `http://www.eclipse.org/aspectj/doc/released/progguide/implementation.html`.

[19] "The aspectj™programming guide, appendix c. bytecode notes." `http://eclipse.org/aspectj/doc/released/progguide/apcs02.html#the-class-expression-and-string`.

[20] P. Sanderson, “Updating the java grande forum benchmarn suite,” 2011.

[21] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, *The Java™Language Specification, Java SE 7 Edition.* 2012.

[22] J. Bloch, *Effective Java, Second Edition*, ch. 2.7, pp. 27–31. 2008.

[23] P. S. Almeida, C. Baquero, and V. Fonte, “Interval tree clocks, a logical clock for dynamic systems.” `http://gsd.di.uminho.pt/teaching/misd/2010/sm/itc.pdf`, 2010.

[24] C.-S. Park and K. Sen, “Randomized active atomicity violation detection in concurrent programs,” in *Proc. 16th International Symposium on Foundations of Software Engineering (FSE’08)*, 2008.

[25] “Java™platform, standard edition 7 api specification, java.lang.thread.” `http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#getState()`.

[26] C. Flanagan and S. N. Freund, “Fasttrack: efficient and precise dynamic race detection,” in *ACM Sigplan Notices*, vol. 44, pp. 121–133, ACM, 2009.