Adversarial Memory for Detecting Destructive Races

Cormac Flanagan
Computer Science Department
University of California at Santa Cruz
Santa Cruz, CA 95064

Stephen N. Freund
Computer Science Department
Williams College
Williamstown, MA 01267

Abstract

Multithreaded programs are notoriously prone to race conditions, a problem exacerbated by the widespread adoption of multi-core processors with complex memory models and cache coherence protocols. Much prior work has focused on static and dynamic analyses for race detection, but these algorithms typically are unable to distinguish destructive races that cause erroneous behavior from benign races that do not. Performing this classification manually is difficult, time consuming, and error prone.

This paper presents a new dynamic analysis technique that uses *adversarial memory* to classify race conditions as destructive or benign on systems with relaxed memory models. Unlike a typical language implementation, which may only infrequently exhibit non-sequentially consistent behavior, our adversarial memory implementation exploits the full freedom of the memory model to return older, unexpected, or stale values for memory reads whenever possible, in an attempt to crash the target program (that is, to force the program to behave erroneously). A crashing execution provides concrete evidence of a destructive bug, and this bug can be strongly correlated with a specific race condition in the target program.

Experimental results with our JUMBLE prototype for Java demonstrate that adversarial memory is highly effective at identifying destructive race conditions, and in distinguishing them from race conditions that are real but benign. Adversarial memory can also reveal destructive races that would not be detected by traditional testing (even after thousands of runs) or by model checkers that assume sequential consistency.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification–*reliability*; D.2.5 [*Software Engineering*]: Testing and Debugging–*monitors, testing tools*; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Algorithms, Verification

Keywords Race conditions, concurrency, dynamic analysis, relaxed memory models

1. Introduction

Multithreaded software systems are notoriously prone to race conditions, which occur when two threads access the same memory location at the same time without synchronization, and at least one of these accesses is a write. Race conditions are particularly problematic because they often cause errors only on certain rare executions, which makes them difficult to detect, reproduce, and eliminate. The widespread adoption of multi-core processors significantly exacerbates these problems, both by increasing the degree of thread interleaving and by making memory model and cache coherence issues much more prevalent. In particular, relaxed memory models cause programs with intentional race conditions to behave in ways that even experienced programmers find subtle, complex, and counter-intuitive.

The insidious nature of race conditions has motivated much prior work on race detection analyses, both static [1, 4, 5, 8, 16, 20, 28, 38, 43] and dynamic [14, 15, 30, 33, 35, 41, 45], as well as via post-mortem analysis [3, 13, 34]. While many of these race detectors are effective at locating potential races, they tend to report a large number of warnings. A substantial first task for a programmer using these tools is to manually classify the reported potential race conditions into the following categories:

- *False alarms* that are caused by analysis imprecisions.
- *Benign races* that exist but do not cause erroneous behavior.
- *Destructive races* that can cause erroneous behavior and should be fixed.

In practice, large software applications typically include a number of intentional and hopefully benign race conditions to mitigate performance concerns. While precise race detectors (*e.g.*, [14, 17]) facilitate race classification by guaranteeing that they never produce false alarms, they still fail to distinguish between benign and destructive race conditions.

Correct classification is critical, since attempts to "fix" benign races may introduce additional real bugs (such as deadlocks) or performance bottlenecks (due to unnecessary synchronization). Moreover, warnings of destructive races could be ignored by the programmer on the mistaken assumption that the races are benign or that the memory model ensures sequentially consistent behavior. Classification by hand is also quite difficult and time consuming and requires a deep understanding of the target code base. Overall, therefore, this classification problem significantly limits the usability and utility of race detection algorithms.

Adversarial Execution. This paper presents a dynamic analysis technique that facilitates precisely identifying destructive race conditions. We first use a standard precise race detector to identify program variables that have (benign or destructive) race conditions. For each "racy" variable, we then attempt to generate an execution in which that race condition causes the program to behave incorrectly. An *erroneous execution* is one that exhibits incorrect observable behavior, such as a crash, an uncaught exception, incorrect output, divergence, etc. If this approach generates an erroneous execution, then we can guarantee that the target program is buggy

PLDI'10, June 5–10, 2010, Toronto, Ontario, Canada.

Figure 1. Racy initialization. Initially `x == null`.

Thread 1	Thread 2
`x = new Circle();`	`if (x != null) { x.draw(); }`

and we have strong evidence for classifying that race condition as destructive. This approach of *adversarial execution* provides two benefits: it only warns the programmer about real errors in the software (that is, no false alarms); and it provides a concrete execution as a witness to that error.

1.1 Memory Models

The essence of our approach to adversarial execution is to exploit the full range of possible behaviors permitted by the relaxed memory models found in most current architectures. In general, a memory model specifies what values may be returned for each read operation in a trace.

The *sequentially consistent* memory model (SCMM) [22] requires each read from an address to return the value of the most recent write by any thread to that address. Although sequential consistency is an intuitive memory model, it significantly limits the optimizations used by the compiler, virtual machine, or hardware.

Relaxed memory models [2, 19], such as the Java Memory Model (JMM) [25] or x86-TSO [31], admit additional optimizations by imposing fewer constraints on the value returned from read operations. For data-race-free programs, each read returns the same value as under SCMM. For programs with (intentional or unintentional) races, however, a read operation could return multiple values, as illustrated by the following two examples.

Racy Initialization Example. In this program, Thread 1 initializes `x` while Thread 2 checks `x!=null` and then calls `x.draw()`. Both reads of `x` by Thread 2 are in a race with the write by Thread 1. Nevertheless, under SCMM, all interleavings of this program behave correctly, since once `x` is initialized as non-null it stays non-null.

Under the Java relaxed memory model, however, each read of `x` could independently read either `null` or an initialized reference. Hence the check `x!=null` could succeed (by reading the initialized value) after which the call `x.draw()` could read `null` and fail with a `NullPointerException`. [1]

Double-Checked Locking Example. As a more interesting example, consider the Java program in Figure 2. The class `Point` contains a static field `p` referring to a singleton `Point` object. This static field is initialized lazily on the first call to `get()`, via a double-checked initialization pattern. Prior precise race detectors such as FASTTRACK [17] and DJIT$^+$ [33] can identify race conditions on three fields (`p`, `p.x`, and `p.y`), but they do not identify which of these race conditions are destructive.

We first consider the race condition on `p`. Line 8 reads `p` into a local variable `t`, so the return value of `get()` is never null. Reading stale null values at line 8 only causes extra executions of the synchronized block, so the race on `p` is not destructive.

We next consider the race between the write of `x` at line 5 and the read at line 17. These accesses never overlap because of the initialization logic in `get()`. Nevertheless, a thread calling `get()` could return the initialized value of `p` without synchronization, meaning that there is no happens-before edge between a different thread's initialization of `x` and that thread's read of `x`. Hence, the read at line 17 could return the default initial value of zero for `x`, causing an immediate `DivisionByZeroException` at line 17. Thus, the race on `x` is destructive.

Similarly, the read of `y` at line 17 could also return a stale zero value, causing incorrect printouts. Therefore, this race is also destructive.

[1] Note that specific JVM implementations may not exhibit all behaviors permissible by the Java Memory Model, and so a specific JVM on specific hardware might never reorder reads in a way that exposes this bug.

Figure 2. Double-checked locking.

```
1  class Point {
2    double x, y;
3    static Point p;
4
5    Point() { x = 1.0; y = 1.0; }
6
7    static Point get() {
8      Point t = p;
9      if (t != null) return t;
10     synchronized (Point.class) {
11       if (p==null) p = new Point();
12       return p;
13     }
14   }
15
16   static double slope() {
17     return get().y / get().x;
18   }
19
20   public static void main(String[] args) {
21     fork { System.out.println( slope() ); }
22     fork { System.out.println( slope() ); }
23   }
24 }
```

1.2 Adversarial Memory

A key difficulty in detecting destructive race conditions like those above via testing alone is that the memory system is likely to exhibit sequentially-consistent behavior *most of the time*. Unexpected values will be read from memory only in certain unlucky circumstances (such as when two conflicting accesses are scheduled closely together on cores without a shared cache, or when two object are allocated at addresses that cause cache conflicts). Thus, even though the memory system is always allowed to exhibit counter-intuitive "relaxed" behavior, the fact that it behaves nicely most of the time makes testing problematic.

To overcome this limitation, we have developed an *adversarial memory* system, JUMBLE, that continually exploits the full flexibility of the relaxed memory model to try to crash the target application.[2] Essentially, JUMBLE stress-tests racy programs by returning older (but still legal) values for read operations whenever possible. To determine which values are legal under the memory model, JUMBLE monitors memory and synchronization operations of the target program and keeps a *write buffer* recording the history of write operations to each racy shared variable. For each read operation, JUMBLE computes the set of *visible* values in that variable's write buffer that can be legally returned according to the memory model. This visible set always contains at least the value of the last write to that variable, but may also contain older values. JUMBLE attempts to heuristically pick an element likely to trigger a program crash and thus provide evidence of a destructive race condition.

To provide a formal foundation for our approach, we first develop an operational specification for a subset of the Java Memory Model. This operational specification expresses the inherent nondeterminism of the memory model in terms of familiar data struc-

[2] JUMBLE targets Java programs, but adversarial memory can be used in any system with a relaxed memory model.

tures such as vector clocks and write buffers. The JUMBLE adversarial memory implementation then reifies this non-deterministic operational specification using heuristics to choose read values that expose destructive races.

Fairness. JUMBLE uses a variety of heuristics to choose which visible value to return for each read operation of the target program. Our simplest heuristic returns the oldest or "most stale" visible value for each read. Interestingly, this heuristic violates fairness properties typically assumed by applications. For example, consider the following busy-waiting loop, which contains an intentional race on the non-volatile boolean variable `done`.

```
while (!done) { yield(); }
```

Even after a concurrent thread sets `done` to `true` via a racy write, our "oldest" heuristic continued to return the original (and still visible) `false` value for `done`, resulting in an infinite loop.

There is some tension between memory model fairness (which helps applications behave correctly) and adversarial memory (which tries to crash applications). Although JMM does not mandate a particular notion of fairness, our implementation guarantees that any unbounded sequence of reads to a particular variable will sometimes return the most recently-written value. This fairness guarantee proved sufficient on all our experiments.

Experimental Results. Experimental results on a range of multithreaded benchmarks show that adversarial memory, although a straightforward concept, is highly effective at exposing destructive races. Each destructive race typically causes incorrect behavior on between 25% and 100% of test runs, as compared to essentially 0% under normal testing. For the example program of Figure 1, JUMBLE reveals this destructive race on roughly every other run, while traditional testing failed to reveal this bug after 10,000 runs.

Much prior work (see, for example, [27, 29, 36, 40]) developed tools that explore multiple interleavings of multithreaded programs, in an attempt to identify defects, including destructive races. Interestingly, because these tools assume sequentially consistency, they cannot detect destructive race conditions, such as those in Figures 1 and 2 and in several of our benchmarks, which only appear under relaxed memory assumptions. Conversely, JUMBLE does not explore all interleavings, and so may not detect destructive races that cause problems only under some interleavings. In general, multithreaded Java programs are prone to both scheduling nondeterminism and memory-model nondeterminism, and model checkers need to exhaustively explore both sources of nondeterminism in order to detect all errors.

1.3 Contributions

In summary, this paper:

- introduces the concept of adversarial memory for detecting destructive races;
- formalizes an operational specification for a subset of the JMM, providing a foundation for our approach (Section 4);
- proves that this operational specification is sound with respect to its declarative specification (Section 4.2);
- describes our adversarial memory implementation and its heuristics for exposing destructive races (Section 5); and
- presents experimental results demonstrating that this approach is effective at identifying destructive race conditions, with modest performance overhead (Section 6).

2. Multithreaded Program Traces

To provide a sound basis for our development, we begin by formalizing multithreaded program traces. A multithreaded program

Figure 3. Multithreaded program traces.

$$\begin{array}{rcl} \alpha \in \mathit{Trace} & ::= & \mathit{Operation}^* \\ a, b \in \mathit{Operation} & ::= & rd(t,x,v) \mid wr(t,x,v) \\ & & \mid acq(t,m) \mid rel(t,m) \\ & & \mid fork(t,u) \mid join(t,u) \\ s,t,u \in \mathit{Tid} \quad x,y \in \mathit{Var} \quad m \in \mathit{Lock} \quad v \in \mathit{Value} & & \end{array}$$

consists of a number of concurrently executing threads, each with a thread identifier $t \in Tid$. These threads manipulate variables $x \in Var$ and locks $m \in Lock$. A *trace* α captures an execution of a multithreaded program by listing the sequence of operations performed by the various threads in the system. We ignore control operations (branches, looping, method calls, etc) and local computations, as they are orthogonal to memory model issues. Thus, the set of operations that a thread t can perform are:

- $rd(t,x,v)$ and $wr(t,x,v)$, which read and write a value v from a variable x;
- $acq(t,m)$ and $rel(t,m)$, which acquire and release a lock m;
- $fork(t,u)$, which forks a new thread u; and
- $join(t,u)$, which blocks until thread u terminates.

This set of operations suffices for an initial presentation of our analysis; our implementation supports a variety of additional synchronization constructs, including `wait`, `notify`, volatile variables, etc.

The *happens-before relation* $<_\alpha$ for a trace α is the smallest transitively-closed relation over the operations[3] in α such that the relation $a <_\alpha b$ holds whenever a occurs before b in α and one of the following holds:

- [PROGRAM ORDER] Both operations are by the same thread.
- [LOCKING ORDER]: a releases a lock that is later acquired by b.
- [FORK ORDER]: a is $fork(t,u)$ and b is by thread u.
- [JOIN ORDER]: a is by thread u and b is $join(t,u)$.

If a happens before b, then we also say that b *happens after* a. If two operations in a trace are not related by the happens-before relation, then they are considered *concurrent*. Two memory access *conflict* if they both access (read or write) the same variable, and at least one of the operations is a write. Using this terminology, a trace has a *race condition* if it has two concurrent conflicting accesses.

3. Memory Models

A memory model specifies what values can be returned for each read operation in a program trace. A trace α is *legal* under a memory model if the value v produced by each read operation $rd(t,x,v)$ in the trace α is permitted under that memory model. The simplest memory model is sequential consistency:

> **Sequential Consistent Memory Model** (SCMM)**:** A read operation $a = rd(t,x,v)$ in a trace α may only return the value of the most recent write to that variable in α.

Although sequential consistency is intuitive, it limits the optimizations that may be performed by the compiler, the virtual machine,

[3] In theory, a particular operation a could occur multiple times in a trace. We avoid this complication by assuming that each operation includes a unique identifier (often called an *issue index* [31]), but, to avoid clutter, we do not include this unique identifier in the concrete syntax of operations.

Figure 4. A trace with a read-then-write race on `x`. The assertion can fail under JMM and HBMM but not under PJMM. Initially `x == y == 0`.

Thread 1	**Thread 2**
`assert (x == 0);`	
`y = 1;`	
	`while (y == 0) { };`
	`x = 1;`

or by the hardware itself. The desire for additional optimization opportunities motivated the introduction of a variety of relaxed memory models, which impose weaker constraints on the values returned for read operations in the presence of race conditions [2, 19].

The happens-before memory model relaxes the requirements on which value is returned by a read operation:

> **Happens-Before Memory Model (HBMM):** A read operation $a = rd(t, x, v)$ in a trace α may return the value v written by any write operation $b = wr(u, x, v)$ provided:
>
> 1. b does not happen after a (*i.e.*, b happens before or is concurrent with a), and
> 2. there is no intervening write c to x where $b <_\alpha c <_\alpha a$.

Condition (1) is quite permissive, and it allows a read operation to "look into the future", as in the following trace where the operation $rd(t_1, x, 1)$ reads the value 1 from the write $wr(t_2, x, 1)$ that appears later in the trace (but which is considered a concurrent write by the happens-before relation):

$$wr(t_1, x, 0).rd(t_1, x, 1).wr(t_2, x, 1)$$

This permissiveness introduces the potential for *out-of-thin-air violations*, as described in the Java Memory Model Specification [25]. We briefly illustrate this problem via the following program in which each thread copies one variable into another:

$$x \texttt{ := } y \quad || \quad y \texttt{ := } x$$

Even if x and y are zero-initialized, this program can generate the following trace under HBMM for any value of v, since the first read reads the second write, etc.

$$rd(t_1, x, v).wr(t_1, y, v).rd(t_2, y, v).wr(t_2, x, v)$$

Out-of-thin-air violations are problematic for garbage-collected languages (since the "out of thin air" value v could be an invalid pointer) and they introduce potential security loopholes. To avoid out-of-thin-air violations, the Java Memory Model [25] (JMM) extends the happens-before memory model with a complex *causality requirement*, essentially precluding nonsensical traces such as the one shown above. In our setting, this causality requirement is unnecessary because our dynamic online analysis is unable to "look into the future". Thus, in the following section we formalize our analysis for a restriction of the JMM, called the *Progressive Java Memory Model*, that removes this ability.

4. The Progressive Java Memory Model

The *Progressive Java Memory Model* is a slight restriction of the Happens-Before Memory Model that removes the ability for a read operation to see a write operation that has not happened yet.

> **Progressive Java Memory Model (PJMM):** A read operation $a = rd(t, x, v)$ in trace α may return the value v written by any write operation $b = wr(u, x, v)$ provided that
>
> 1. b executes *before* a in the trace α; and

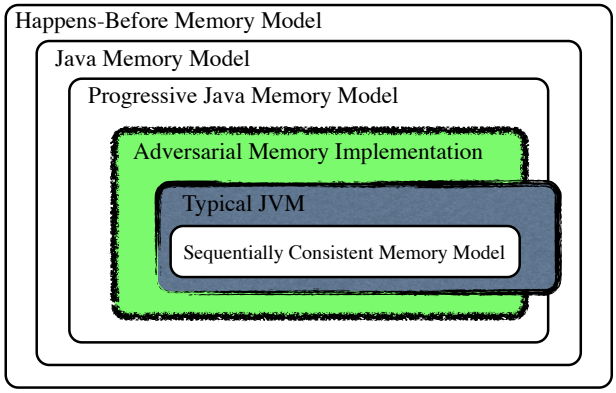


Figure 5. An illustration of four memory models and two memory implementations, including a typical JVM.

> 2. there is no intervening write c to x where $b <_\alpha c <_\alpha a$.

Here, condition 1 permits only reading past writes, and is more restrictive than HBMM for traces involving read-then-write race conditions, where a read is in a race with a later write. Under HBMM and JMM, the read can "see" the later write; under PJMM, it can not.

Figure 4 illustrates such a trace with a read-then-write race on `x`, where the assertion of Thread 1 can read a future value for `x` and thus fail under HBMM and JMM, but the assertion cannot fail under PJMM. Excluding such traces from PJMM greatly simplifies our JUMBLE implementation, without in practice significantly limiting its ability to expose destructive races.

We illustrate the relationship between these four memory models discussed so far (HBMM, JMM, PJMM, and SCMM) via the Venn diagram of Figure 5. The more flexible memory models permit a greater set of program behaviors than the more restrictive. This diagram also roughly sketches the memory model behaviors exposed by two implementations: a typical Java Virtual Machine (JVM) implementation, and by our adversarial memory implementation. A typical JVM running on commodity hardware exposes behaviors that are not sequentially consistent relatively infrequently, because, for example, those platforms do not perform all optimizations permissible under the JMM. In contrast, JUMBLE explores a larger (and more "adversarial") fraction of the behavior permitted by the JMM, thereby exposing more destructive races.

4.1 An Operational Specification of the PJMM

To guide our implementation of an adversarial memory system, we now present an operational formulation of the PJMM, called the *Operational* PJMM.[4] In particular, whereas the above PJMM definition is expressed in terms of a mathematical characterization of the happens-before relation, the Operational PJMM is expressed in terms of data structures analogous to those used in our implementation: *vector clocks* that represent the happens-before relation and *write buffers* that record the history of writes to each variable.

A vector clock $K : Tid \rightarrow Nat$ records a clock for each thread in the system [26]. Vector clocks are partially-ordered ($\sqsubseteq$) in a point-wise manner, with an associated join operation ($\sqcup$) and minimal element ($\bot$). The helper function inc_t increments the t-

[4] The Operational PJMM is motivated by similar goals as [7], but is less complex as we only provide a semantics for traces, not programs.

component of a vector clock:

$$\begin{array}{rcl} K_1 \sqsubseteq K_2 & \text{iff} & \forall t.\ K_1(t) \leq K_2(t) \\ K_1 \sqcup K_2 & \stackrel{\text{def}}{=} & \lambda t.\ max(K_1(t), K_2(t)) \\ \bot & \stackrel{\text{def}}{=} & \lambda t.\ 0 \\ inc_t(K) & \stackrel{\text{def}}{=} & \lambda u.\ \texttt{if}\ u = t\ \texttt{then}\ K(u) + 1\ \texttt{else}\ K(u) \end{array}$$

We express the Operational PJMM as an online analysis that maintains a *memory state* σ. This memory state is updated by each operation a in the observed trace via the relation:

$$\sigma \Rightarrow^{a} \sigma'$$

This relational formulation naturally supports non-deterministic reads: for a read of x by thread t from σ, it may be possible to conclude both $\sigma \Rightarrow^{rd(t,x,v_1)} \sigma$ and $\sigma \Rightarrow^{rd(t,x,v_2)} \sigma$, indicating that the read may return either v_1 or v_2. (Note that read operations do not affect the memory state σ.)

Memory State. A memory state σ is a tuple (C, L, W), where:

- $C : Tid \rightarrow K$ identifies the current vector clock of each thread t. Initially, each thread t starts with the vector clock $C(t) = inc_t(\bot)$, indicating that this thread has performed one clock tick.
- $L : Lock \rightarrow K$ is the vector clock of the last release of each lock. Each lock $L(m)$ is initially $\bot$, indicating that the lock was never acquired.
- $W : Var \rightarrow WriteBuffer$ contains a non-empty write buffer for each variable in the program. A $WriteBuffer = (Value@K)^{+}$ records the sequence of values written to that variable together with a vector clock timestamp for each write. Initially, each write buffer $W(x)$ contains a single entry $0@\bot$, reflecting that memory is zero-initialized before execution.

Thus, the initial memory state is:

$$\sigma_0 \stackrel{\text{def}}{=} (\lambda t.inc_t(\bot),\ \ \lambda m.\bot,\ \ \lambda x.(0@\bot))$$

Transition Rules. Figure 6 describes how the Operational PJMM handles each operation of the observed trace.

[ACQUIRE]: The rule for $acq(t, m)$ updates the vector clock C_t to reflect that subsequent operations of thread t happen after the last release of lock m, which happened at time L_m. Here, C is a function, C_t abbreviates the function application $C(t)$, and $C[t := b]$ denotes the function that is identical to C except that it maps t to b.

[RELEASE]: The rule for $rel(t, m)$ updates L_m with the current vector clock C_t of thread t, and then increments the t-component of C_t, so that the vector clocks can distinguish operations by thread t that happen before and after that release operation.

[FORK]: The rule for $fork(t, u)$ updates C_u to be after C_t (reflecting that the first operation of the forked thread u happens after this fork operation) and increments the t-component of C_t.

[JOIN]: The rule for $join(t, u)$ updates C_t to be after C_u (reflecting that this join operation of thread t happens after the last operation of the joined thread u). As a technical device to simplify proof invariants, this rule also increments the u-component of C_u.

[WRITE]: The rule for $wr(t, x, v)$ extends the write buffer W_x with an entry $v@C_t$ recording that the value v was written at time C_t.

[READ]: The rule for $rd(t, x, v_i)$ non-deterministically picks some value v_i from the current write buffer, which in general has the form:

$$W_x \ = \ v_1@K_1 \cdot v_2@K_2 \cdot \ldots \cdot v_n@K_n$$

Figure 6. The Operational Progressive Java Memory Model.

[ACQUIRE]
$$\frac{C' \ = \ C[t := (C_t \sqcup L_m)]}{(C, L, W) \Rightarrow^{acq(t,m)} (C', L, W)}$$

[RELEASE]
$$\frac{L' \ = \ L[m := C_t] \qquad C' = C[t := inc_t(C_t)]}{(C, L, W) \Rightarrow^{rel(t,m)} (C', L', W)}$$

[FORK]
$$\frac{C' \ = \ C[u := C_u \sqcup C_t, t := inc_t(C_t)]}{(C, L, W) \Rightarrow^{fork(t,u)} (C', L, W)}$$

[JOIN]
$$\frac{C' \ = \ C[t := C_t \sqcup C_u, u := inc_u(C_u)]}{(C, L, W) \Rightarrow^{join(t,u)} (C', L, W)}$$

[WRITE]
$$\frac{W' \ = \ W[x := W_x \cdot v@C_t]}{(C, L, W) \Rightarrow^{wr(t,x,v)} (C, L, W')}$$

[READ]
$$\frac{\begin{array}{c} W_x = (v_j@K_j)^{j \in 1..n} \qquad i \in 1..n \\ \forall j \in i+1..n.\ \neg(K_i \sqsubseteq K_j \sqsubseteq C_t) \end{array}}{(C, L, W) \Rightarrow^{rd(t,x,v_i)} (C, L, W)}$$

[GC1]
$$\frac{\begin{array}{c} W_x = (v_j@K_j)^{j \in 1..n} \qquad i \in 1..n \\ \forall t \in Tid.\ \exists j \in i+1..n.\ K_i \sqsubseteq K_j \sqsubseteq C_t \\ W' = W[x := (v_j@K_j)^{j \in 1..i-1} \cdot (v_j@K_j)^{j \in i+1..n}] \end{array}}{(C, L, W) \Rightarrow^{\epsilon} (C, L, W')}$$

[GC2]
$$\frac{\begin{array}{c} W_x = (v_j@K_j)^{j \in 1..n} \\ v_i = v_m \qquad K_i = K_m \qquad 1 \leq i < m \leq n \\ W' = W[x := (v_j@K_j)^{j \in 1..i-1} \cdot (v_j@K_j)^{j \in i+1..n}] \end{array}}{(C, L, W) \Rightarrow^{\epsilon} (C, L, W')}$$

[REMOVE OLDEST]
$$\frac{\begin{array}{c} W_x = (v_j@K_j)^{j \in 1..n} \qquad n > 1 \\ W' = W[x := (v_j@K_j)^{j \in 2..n}] \end{array}}{(C, L, W) \Rightarrow^{\epsilon} (C, L, W')}$$

To yield a legal PJMM trace, that rule ensures there is no subsequent write $v_j@K_j$ in the buffer at position $j \in i + 1..n$ that

1. happens after the write of v_i (so $K_i \sqsubseteq K_j$) and
2. that happens before this read operation (so $K_j \sqsubseteq C_t$).

If there is no such j, then this read operation can legally return the value v_i.

Note that [READ] always permits the most recent value v_n to be returned, and possibly older writes as well. If there are multiple entries in the write buffer that satisfy the requirements of the [READ] rule, then we say that the read operation has an *sequential consistency violation*, since it could return values that are not permitted under the SCMM.

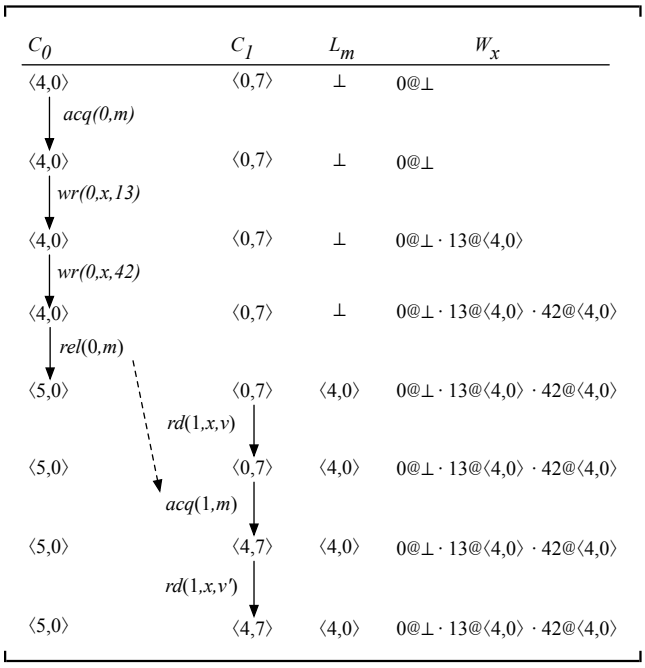
As an example, entries i and m in the write buffer

$$\ldots \cdot v_i@K_i \cdot \ldots \cdot v_m@K_m \cdot \ldots$$

could both be read provided that either (1) the second write is in a race with the first write ($K_i \not\sqsubseteq K_m$) or (2) the second write is in a race with the read ($K_m \not\sqsubseteq C_t$). Thus, even if the read operation is race-free ($K_i \sqsubseteq C_t$ and $K_m \sqsubseteq C_t$), it can still produce a sequential consistency violation if the previous write operations are racy ($K_i \not\sqsubseteq K_m$).

Conversely, if the trace has no race conditions on a particular variable, then all accesses are totally ordered, and so are the vector

Figure 7. Example execution trace.



clocks $K_1 \sqsubseteq K_2 \sqsubseteq \cdots \sqsubseteq K_n \sqsubseteq C_t$. Hence, any read will return the value v_n of the most recent write.

Example. The example trace in Figure 7 illustrates the key points of the Operational PJMM, including the need for ordered write buffers. That figure includes the relevant parts of the memory state σ: the vector clocks C_0 and C_1 of Thread 0 and Thread 1; the vector lock L_m of the lock m; and the write buffer W_x for the variable x.

- The first write $wr(0, x, 13)$ at time $C_0 = \langle 4, 0\rangle$ extends the write buffer W_x with a second entry $13@\langle 4, 0\rangle$.
- The next write adds a third entry $42@\langle 4, 0\rangle$. Note that the last two entries in the write buffer now contain identical vector clocks. Nevertheless, the order of the write buffer entries still indicates that 42 was written sometime after 13.
- The release $rel(0, m)$ updates L_m to $\langle 4, 0\rangle$.
- The first read $rd(1, x, v)$ then reads x without holding the lock. The [READ] rule enables us to determine which of the three entries in W_x are visible to that read at time $C_1 = \langle 0, 7\rangle$:
  - $42@\langle 4, 0\rangle$ is visible: it is the last entry in the write buffer.
  - $13@\langle 4, 0\rangle$ is visible: the later entry $42@\langle 4, 0\rangle$ is concurrent with the current clock C_1 since $\langle 4, 0\rangle \not\sqsubseteq \langle 0, 7\rangle = C_1$.
  - $0@\bot$ is also visible: again, the later entries at time $\langle 4, 0\rangle$ are concurrent with the current clock C_1.

  Thus, the read operation $rd(1, x, v)$ can return any value in the write buffer (0, 13, or 42) as v, resulting in a sequential consistency violation.
- The acquire $acq(1, m)$ increases C_1 to be at least L_m, reflecting the happens-before edge from the release to the acquire.
- The second read operation $rd(1, x, v')$ now reads x in a race-free manner, and the [READ] rule again enables us to determine which entries in W_x are visible to that read at time $C_1 = \langle 4, 7\rangle$:
  - $42@\langle 4, 0\rangle$ is visible: it is the last entry in the write buffer.
  - $13@\langle 4, 0\rangle$ is not visible: the later entry $42@\langle 4, 0\rangle$ now prevents Thread 1 from seeing this value, since $\langle 4, 0\rangle \sqsubseteq \langle 4, 7\rangle = C_1$. Note that the order of write buffer entries allows us to correctly distinguish between these two entries (13 and 42) that have the same vector clock.
  - $0@\bot$ is not visible: due to the later entries at time $\langle 4, 0\rangle$.

  Thus, the second read $rd(1, x, v')$ must return 42, not 13 or 0, and thus does not have a sequential consistency violation.

Write Buffer Compression. Our operational memory model includes three additional rules to prevent write buffers from becoming arbitrarily large.

[GC1]: This rule discards an entry $v_i@K_i$ from a write buffer

$$W_x = v_1@K_1 \cdot v_2@K_2 \cdot \ldots \cdot v_n@K_n$$

provided v_i is not visible to any thread. To ensure this "invisibility" property, the rule checks that, for each thread t, there is some intervening write $v_j@K_j$ such that $K_i \sqsubseteq K_j \sqsubseteq C_t$. In this situation, removing the entry $v_i@K_i$ from the write buffer does not change the set of traces accepted from the current state.

For the final memory state of Figure 7, this rule can remove the entries $0@\bot$ and $13@\langle 4, 0\rangle$, since the last write $42@\langle 4, 0\rangle$ serves an an intervening write in both cases.

[GC2]: This rule identifies situations where some thread t writes a value v to a variable x twice in a row, with no intervening synchronization operations, yielding a write buffer of the form:

$$W_x = \ldots \cdot v@C_t \cdot \ldots \cdot v@C_t \cdot \ldots$$

The first occurrence of $v@C_t$ is now redundant. If thread t reads from x, the first write is hidden due to program order. If a different thread reads from x and the first write by t is visible, then the second write by t is also visible. Thus we can remove the earlier write from the write buffer without changing the set of traces accepted from the current state.

[REMOVE OLDEST]: The previous two techniques work well in practice most of the time, but races on frequently modified variables could still cause buffers to grow quite large. The final rule allows the oldest entry to be dropped from the write buffer at any point, provided that there is still at least one write remaining. This rule is sound, in the sense that it will never accept an invalid trace, but it limits the subsequent freedom of our adversarial memory implementation to return stale values that are likely to cause crashes. Hence, we apply this rule only when necessary, that is, when write buffers would otherwise grow too large. Using a maximum buffer size of 32 did not impact the precision of JUMBLE for those programs requiring much larger buffers to hold the entire visible history.

4.2 Correctness of the Operational PJMM

We now address the correctness of the Operational PJMM. Suppose the target program P behaves incorrectly on a trace α that is legal under the Operational PJMM. Theorem 1 below implies that α is also a legal PJMM trace. The PJMM is essentially a restriction of JMM, and follows a similar specification style based on the happens-before relation, indicating that α is therefore also a legal JMM trace, and so P could also behave incorrectly on a JVM.

Before proving Theorem 1, we first introduce some additional notation. For any transition $\sigma \Rightarrow^a \sigma'$, we use C^a, L^a, and W^a to denote the components of σ, and we use C^a_a to abbreviate $C^a_{tid(a)}$, which denotes the clock vector of the thread executing a just before that operation is executed.

We restrict our attention to *feasible* traces that respect the following expected constraints on forks, joins, and locking operations.

1. There can be no instructions of thread u preceding an instruction $fork(t, u)$ or following an instruction $join(t, u)$.
2. No thread can release a lock it did not previously acquire.
3. No thread can acquire a lock previously acquired but not released by another thread.

THEOREM 1. *If $\sigma_0 \Rightarrow^{\alpha} \sigma$ then α is a legal* PJMM *trace.*

PROOF A commuting argument [23] can be used to show that the garbage collection rules ([GC1], [GC2], [REMOVE OLDEST]) all "right-commute" with the other rules. Hence, applications of the GC rules can be moved to the end of the trace, yielding a trace prefix β that does not include any GC rules. We now show that this theorem holds for β.

Consider any read $a = rd(t, x, v) \in \beta$ in a memory state where the write buffer has the form:

$$W_x^a \;=\; v_1@K_1 \cdots v_i@K_i \cdots v_n@K_n$$

The rule [READ] implies that $v = v_i$ for some i, and that:

$$\forall j \in i+1..n.\ \neg(K_i \sqsubseteq K_j \sqsubseteq C_a^a)$$

From an inspection of the rules, the write buffer entry $v_i@K_i$ must have been added to W_x by the rule [WRITE] for some operation $b = wr(u, x, v) \in \beta$, where $K_i = C_b^b$.

Now suppose there was an intervening write $c = wr(u', x, v') \in \beta$ with $b <_\beta c <_\beta a$. By Lemma 1 below, we have that $C_b^b \sqsubseteq C_c^c \sqsubseteq C_a^a$. Also, c occurs between b and a in the trace, meaning the entry $v'@C_c^c$ must have been added to the write buffer after b. Thus there exists $j \in i+1..n$ such that $K_j = C_c^c$. Hence, we have that:

$$K_i \sqsubseteq K_j \sqsubseteq C_a^a,$$

which is a contradiction. Thus, there is no intervening write c such that $b <_\beta c <_\alpha a$, and the operation a can thus read the value written by b under PJMM. □

The following lemma clarifies that vector clocks correctly represent the happens-before relation. Previous work proves a similar lemma by induction [17].

LEMMA 1 (Happens-Before implies Vector Clocks).
Suppose $\sigma_0 \Rightarrow^{\alpha} \sigma$ and $a, b \in \alpha$ are both read or write operations. If $a <_\alpha b$ then $C_a^a \sqsubseteq C_b^b$.

5. Adversarial Memory Implementation

The Operational PJMM expresses the inherent non-determinism of the memory model in an operational manner. The JUMBLE adversarial memory implementation resolves this non-determinism in a manner designed to expose destructive races.

Suppose we have previously identified data races on a specific variable x in our target program. JUMBLE executes that program in a special environment where all writes to x are recorded in a write buffer, and all reads from x are *adversarial* in that JUMBLE attempts to pick visible values that are likely to trigger erroneous behavior. If the target program behaves erroneously under such adversarial reads on x, then we characterize that race condition on x as being destructive.

5.1 Heuristics

In more detail, consider a read operation $rd(t, x, \_)$ performed in a memory state $\sigma = (C, L, W)$ where the write buffer for x is:

$$W_x \;=\; v_1@K_1 \cdot v_2@K_2 \cdot \ldots \cdot v_n@K_n$$

By the [READ] rule, the read operation can return any value v_i in the write buffer that does not have an intervening write v_j with $K_i \sqsubseteq K_j \sqsubseteq C_t$. Thus, the set Vis of *visible* values is:

$$Vis \;\stackrel{\text{def}}{=}\; \{v_i \mid i \in 1..n \text{ and } \forall j \in i+1..n.\ \neg(K_i \sqsubseteq K_j \sqsubseteq C_t)\}$$

This set always contains the most recent write v_n, but possibly additional values as well. We have implemented five heuristics for resolving this non-determinism in the case where Vis is not the singleton set $\{v_n\}$.

- *Sequentially-Consistent*. This heuristic always returns the most recently written value v_n, and provides a baseline with which to compare our more adversarial heuristics.
- *Oldest*. This heuristic picks the oldest element of V, based on the intuition that this "most stale" value is likely to induce bad behavior. Occasionally, this heuristic returns the most recently written value v_n instead, in order to satisfy fairness properties assumed by busy-waiting loops, as mentioned in the introduction.
- *Oldest-But-Different*. Consider the example of Figure 8, in which a `for` loop encloses the check-then-dereference idiom from Figure 1:

  ```
  if (x != null)  { x.draw(); }
  ```

  For this program, the *Oldest* heuristic consistently returns the same value for all reads of `x`, and thus fails to expose this destructive race. In contrast, the *Oldest-But-Different* heuristic picks the oldest element of Vis that is different from the last value read from that variable. Once the write buffer contains the write to `x` from Thread 1, *Oldest-But-Different* alternately returns null and non-null pointers and detects this destructive race on essentially every execution. (No error was detected on traces in which the initialization happened only after the loop terminated.)
- *Random*. Pick a random value from Vis.
- *Random-But-Different*. Pick a random value from Vis that is different from the last value read for that variable.

Section 6 presents an experimental comparison of these heuristics.

Figure 8. Racy initialization revisited. Initially `x == null`.

Thread 1	**Thread 2**

`x = new Circle();`	`for(int i=0; i<10; i++) {`
	`  if (x != null)  { x.draw(); }`
	`}`

5.2 JUMBLE Implementation Details

Our JUMBLE implementation is based on the ROADRUNNER framework [17, 18]. ROADRUNNER inserts instrumentation code into the target bytecode program at load time that will generate a stream of events for synchronization operations, field accesses, etc, and JUMBLE processes this event stream as it is generated.

Our implementation supports additional synchronization primitives not described in our memory model, including `wait`/`notify`, and volatile variables. Extending a happens-before analysis based on vector clocks to handle these cases is straightforward, as described in [17]. For simplicity, re-entrant lock acquires and releases (which are redundant) are filtered out by ROADRUNNER and are never passed to JUMBLE.

JUMBLE is configurable to record write buffers for the memory locations corresponding to any set of syntactic fields in the source program. While we could record write buffers for all memory locations used by a program, we have found it more useful during our initial experiments to configure it to analyze instances of only one syntactic field (identified earlier by a precise race detector) at a

time. If crashes occur only under those conditions, the underlying cause is most likely a destructive race condition on the field being "jumbled". Tracking only a single syntactic field at a time may miss some errors that involve multiple fields, as described in [24, 39], but we leave tracking multiple fields for future work.

JUMBLE ***State.*** JUMBLE associates with each thread in the target program a vector clock represented by an array of 32-bit integers.[5] JUMBLE also maintains a write buffer of value/clock-vector pairs for each monitored memory location. When a read occurs, JUMBLE uses the reading thread's vector clock to identify which writes are visible, and applies one of the previously-described heuristics to choose among them. The JUMBLE implementation limits write buffers to contain a bounded number of entries, typically 32.

Non-Atomic Longs and Doubles. JUMBLE follows the Java memory model specification in treating reads from 8-byte longs and doubles as non-atomic, and it implements them as two separate 4-byte reads. That is, when JUMBLE performs a read from a location storing a 8-byte value, it extracts two distinct, visible writes from that location's write buffer, using the top part of one and the bottom part of the other to construct the value returned to the program. In this manner, racy reads from 8-byte locations often return corrupted values that are likely to result in erroneous executions.

Arrays. JUMBLE can also jumble the values returned by array reads. To avoid high overheads on array intensive programs with huge numbers of array accesses, JUMBLE incorporates a sampling technique. We use a precise race detector to identify the array indices at which data races occur. JUMBLE then tracks values associated with a small subset of those indices for every array created by the target. This approach proved quite effective in practice — JUMBLE induced crashes for our test programs with array races when only jumbling accesses to arrays at index 0 or 1. The overhead was usually higher than when tracking a single syntactic field, but still acceptable.

6. Experimental Results

We used JUMBLE to examine all 10 race conditions detected by the FASTTRACK precise race detector [17] in a variety of multithreaded benchmarks.[6]

The programs examined include `jbb`, the SPEC JBB2000 business object simulator [37]; `montecarlo`, `sor`, `lufact`, `moldyn`, and `raytracer` from the Java Grande benchmark suite [21]; `mtrt`, a multithreaded ray-tracing program from the SPEC JVM98 benchmark suite [37]; and `tsp`, a Traveling Salesman Problem solver [42]. Their sizes, number of threads, and running times are shown in Figure 10. Experiments were performed on an Apple Mac Pro with dual 3GHz quad-core Pentium Xeon processors and 12GB of memory, running OS X 10.5.8 and Sun's Java HotSpot 64-bit Server VM version 1.6.0.

6.1 Effectiveness of Adversarial Memory

We compared the behavior of JUMBLE under six different memory implementations: *No Jumble* (in which the target program is executed directly by the HotSpot JVM) and the five JUMBLE heuristics described in Section 5. For each of the races and each of the six configurations, we ran 100 tests to detect how often that race condition caused erroneous behavior. For races on fields, we jumbled reads from all instances of that field. For races on arrays, we jumbled reads from all arrays at index 0 and 1, as described above.

Figure 9 summarizes the results for our benchmark programs, and also for the example programs in Figures 2 and 8. The last column of that figure presents the results of our manual (and time-consuming) classification of each race condition as benign or destructive. We classified a race as destructive only if we could observe deviant program behavior by doing nothing more than inserting `Thread.sleep()` operations to guide the scheduler to potentially bad interleavings.

The *No Jumble* heuristic exposed none of the destructive races, which confirms the conventional folklore that race conditions are extremely difficult to detect via traditional testing alone. The *Sequentially-Consistent* heuristic demonstrates that our instrumentation and monitoring framework, while invariably impacting thread scheduling, does not in itself expose destructive behaviors.

The remaining columns demonstrate that the other JUMBLE heuristics are highly effective at exposing destructive race conditions. Seven of the nine race conditions were destructive, and each destructive race condition is detected by at least one heuristic with high probability. We again examined program behavior manually to identify incorrect behavior.

As expected by the correctness arguments of Section 4.2, none of the benign races caused incorrect behavior under any of the configurations. We discuss each race condition in turn:

Programs in Figures 8 and 2: Under JUMBLE, the code in Figure 8 generated a "null pointer exception." Two values become visible in the write buffer for `x` after the initialization in Thread 1 occurs: the new, non-`null` value, and the original `null` value.

As previously mentioned, the *Oldest* heuristic fails to uncover the error because it always returns the same `null` value for every access. In contrast, the *Oldest-But-Different* heuristic causes the crash with high probability. The random schemes are also effective in this case. No crash occurs only on traces in which Thread 1 writes to `x` only after Thread 2 has finished.

Similarly, the destructive races previously described for the program in Figure 2 are detected with fairly high probability, but the benign race triggers no visible errors.

Program jbb, `Company.elapsed_time`: The main thread reads each company's `elapsed_time` field while computing timing data. However, due to the lack of synchronization, multiple values are visible in the write buffer, including the initial value of 0. Since this field is a `long`, JUMBLE merges 4-byte words from different writes, causing corrupted statistics to be reported. Since some values constructed in this way result in "reasonable" output for the program, the *Oldest-But-Different* and random heuristics do not uncover the errors 100% of the time.

Program jbb, `Company.mode`: This variable records the state of a company object during simulation. The transaction manager tests whether `mode` is `RAMP_DOWN` to decide whether to wake up a waiting object. If a stale value is read, the waiting object will never awaken, and the program fails to terminate.

Program montecarlo, `Universal.UNIVERSAL_DEBUG`: During the test runs, all writes to this global debugging flag wrote the same value, so no difference in behavior could be discerned by JUMBLE, and we considered this race benign.

Program mtrt, `RayTracer.threadCount`: In this program a `RayTracer` object creates a group of `Runner` worker threads that all refer to the `RayTracer` as parent. The parent increments `threadCount` each time a runner is created, and each `Runner` decrements that variable *without synchronization* upon completion. JUMBLE causes the `threadCount` variable to become corrupted,

[5] While 32-bit integers were sufficient for our tests, switching to 64-bits would enable JUMBLE to handle larger clocks, but with additional overhead.

[6] The reported races on these programs differ slightly from our earlier published results [17] due to changes in the FASTTRACK implementation, including improvements in how it creates happens-before edges for calls to `Thread.interrupt()`.

Program	Field	Erroneous Behavior Observation Rate (%)						Destructive Race?
		No Jumble	JUMBLE Configurations					
			Sequentially Consistent	*Oldest*	*Oldest-But-Different*	*Random*	*Random-But-Different*	
Figure 8	`x`	**0**	**0**	**0**	**83**	**84**	**92**	**Yes**
Figure 2	`p`	0	0	0	0	0	0	No
Figure 2	`p.x`	**0**	**0**	**60**	**52**	**32**	**30**	**Yes**
Figure 2	`p.y`	**0**	**0**	**48**	**53**	**27**	**30**	**Yes**
jbb	`Company.elapsed_time`	**0**	**0**	**100**	**0**	**15**	**5**	**Yes**
jbb	`Company.mode`	**0**	**0**	**100**	**100**	**95**	**98**	**Yes**
montecarlo	`Universal.UNIVERSAL_DEBUG`	0	0	0	0	0	0	No
mtrt	`RayTracer.threadCount`	0	0	0	0	0	0	No
raytracer	`JGFRayTracerBench.checksum1`	**0**	**0**	**100**	**100**	**100**	**100**	**Yes**
tsp	`TspSolver.MinTourLen`	0	0	100	100	100	100	QoS
sor	array index `[0]` and `[1]`	**0**	**0**	**100**	**100**	**100**	**100**	**Yes**
lufact	array index `[0]` and `[1]`	**0**	**0**	**100**	**100**	**100**	**100**	**Yes**
moldyn	array index `[0]` and `[1]`	**0**	**0**	**100**	**100**	**100**	**100**	**Yes**

Figure 9. Observation rate for erroneous behavior under various heuristics. Destructive races are marked in bold. QoS indicates that the only observed difference was significant slowdown.

but the value of that variable is not used anywhere else in the program. Thus we consider this race benign.

Program raytracer, `JGFRayTracerBench.checksum1`**:** This program creates a group of worker threads that, upon completion, add a thread-local checksum to the global checksum `checksum1`, without synchronization. Under JUMBLE, `checksum1` becomes corrupted, and the program detects and reports a failed execution. JUMBLE's treatment of `long`s helps uncover this error.

Program tsp, `TspSolver.MinTourLen`**:** This TSP solver uses worker threads to explore and evaluate routes, using a branch-and-bound algorithm in which the length of the current best route is stored in `MinTourLen` and monotonically decreases. The protecting lock `MinLock` is held for updates to `MinTourLen`, but not for reads, via the following variant of double-checked locking:

```
static void set_best(int best, int[] path) {
  if (best >= MinTourLen) return;
  synchronized(MinLock) {
    if (best < MinTourLen) {
      MinTourLen = best;
      for (int i = 0; i < Tsp.TspSize; i++)
        MinTour[i] = path[i];
    }
  }
```

Worker threads check and discard partially constructed paths longer than `MinTourLen`. This check is performed without acquiring `MinLock`, meaning that stale (*i.e.*, larger) values could be read, which would cause redundant path exploration. The program ran up to twice as slow under JUMBLE because of redundant path exploration, which we consider a "Quality of Service" (QoS) problem rather than a destructive race.

Program sor, arrays: Between each iteration of this algorithm, worker threads wait for their "neighboring" threads to finish using a barrier implemented with the array `sync`, where `sync[id][0]` counts iterations finished by the thread `id`. The following code signals that `id` has finished and waits for its neighbors.

```
public static volatile long sync[][];
  ...
  sync[id][0]++;
  if (id > 0)
     while (sync[id-1][0] < sync[id][0]) ;
  if (id < JGFSORBench.nthreads -1)
     while (sync[id+1][0] < sync[id][0]) ;
```

Unfortunately, this code does not include any synchronization — perhaps because the programmer mistakenly assumed that reads of the `volatile` variable `sync` would be sufficient. Therefore, the barrier does not introduce happens-before edges between writes before the barrier and reads following the barrier, so read operations could read stale data, causing the program to compute the incorrect final value. The program recognizes and reports this failure when validating its result 100% of the time under JUMBLE.

Programs lufact and moldyn, arrays: A `TournamentBarrier` class shared by these programs has a similar flaw. It maintains an array `IsDone` of boolean flags to indicate whether a thread has finished and is now waiting at the barrier: Since writes to the elements of `IsDone` are not ordered, a thread reading an older value can get out of sync and essentially live-lock waiting at the barrier. All of our heuristics triggered non-termination 100% of the time.

6.2 JUMBLE Performance

Figure 10 investigates JUMBLE's performance overhead and other run-time statistics. It first shows the base running time of each benchmark, when no instrumentation or monitoring is performed, and then shows the slowdown under ROADRUNNER using both the EMPTY checker and JUMBLE. The EMPTY checker performs no analysis and just measures the overhead of using the ROADRUNNER. We configured JUMBLE to use the *Sequentially Consistent* heuristic when measuring performance in order to avoid the extra path exploration performed by benchmarks such as `TspSolver` under other heuristics. The other heuristics have comparable performance to *Sequentially Consistent*, except in degenerate cases like `TspSolver`. Each measurement averages ten test runs.

Programs incur a slowdown between roughly 1.2x and 5x when run under EMPTY. Most of this overhead is due to instrumenting class files and generating events for synchronization operations. The slowdown for JUMBLE is roughly the same as EMPTY in most cases, with only minor variations due to instrumentation and event handling. This low overhead is because JUMBLE performs relatively few write-buffer operations, since it tracks a small number of racy memory locations and each one is updated only a small number of times (as shown in the "Num. Instances" and "Num. Writes" columns). More significant differences were seen for the array-based programs, since the barrier defects in those programs described above cause the write buffers to become much larger and more heavily used. In these cases, more aggressive sampling or tracking fewer arrays would help keep the overhead lower.

The last two columns of Figure 10 shows the maximum buffer size required, both with and without the use of our three compression rules. When using these rules, JUMBLE limited buffers to contain at most 32 entries, but the garbage collection rules [GC1] and [GC2] were sufficient to ensure that this bound was never reached

Program	Size (lines)	Num. Threads	Field	Base Time (s)	Slowdown Empty	Slowdown Jumble	Num. Instances	Num. Writes	Max. Buffer Size No Comp.	Max. Buffer Size With Comp.
jbb	30,491	5	`Company.elapsed_time`	74.4	1.3	1.3	1	2	2	2
jbb	30,491	5	`Company.mode`	74.4	1.3	1.4	2	10	8	4
montecarlo	3,669	4	`Universal.UNIVERSAL_DEBUG`	1.6	1.2	1.2	1	40,005	40,005	5
mtrt	11,317	5	`RayTracer.threadCount`	0.5	4.5	4.9	1	10	10	5
raytracer	1,970	4	`JGFRayTracerBench.checksum`	5.6	1.1	1.1	1	6	6	5
tsp	742	5	`TspSolver.MinTourLen`	0.7	2.3	4.0	1	26	26	23
sor	883	4	array index `[0]` and `[1]`	0.6	3.9	5.8	2,106	104,620	255	32
lufact	1,627	4	array index `[0]` and `[1]`	0.4	4.1	4.2	1,108	14,526	2,047	7
moldyn	1,407	4	array index `[0]` and `[1]`	0.9	4.1	8.9	62	53,433	16,383	32

Figure 10. Performance of JUMBLE under the *Sequentially-Consistent* configuration.

for all but two programs, and the garbage collection overhead was negligible. The `montecarlo` and `lufact` benchmarks benefited the most, and garbage collection enabled the buffers for those programs to be several orders of magnitude smaller that otherwise. For some array-intensive benchmarks, JUMBLE had to apply the [REMOVE OLDEST] rule to maintain this bound on write buffers, but in practice this rule did not limit JUMBLE's ability to detect destructive races.

6.3 Checking the Eclipse Development Environment

To validate JUMBLE in a more realistic environment, we also applied it to the Eclipse development environment, version 3.4.0. FASTTRACK reported 27 race conditions on a test configuration that involved starting-up Eclipse and rebuilding a collection of projects. Our subsequent experiments were limited by the requirement to run Eclipse interactively, since we did not have an appropriate automated test harness. Therefore, for each of these 27 racy fields, we interactively ran JUMBLE only a single time looking for incorrect behaviors.

For four of these racy fields, these JUMBLE tests produced null pointer exceptions, providing clear evidence of a destructive race. Four other fields produced non-deterministic reads, but the read value did not cause incorrect behavior (at least in this single run). For the remaining fields, JUMBLE did not detect non-deterministic reads, indicating that the races were on fields to which the same value was written, or were similar to the read-then-write race in Figure 4. An automated test infrastructure would provide the ability to perform more test runs and to identify more destructive races.

Nevertheless, by showing how to easily identify four previously-unknown destructive race conditions in a well-tested and robust software system such as Eclipse, these preliminary experiments already demonstrate the effectiveness of adversarial memory.

7. Related Work

The difficulty of manually identifying destructive races has motivated prior work on this problem. One approach uses replay analysis [29] to re-execute a racy trace after swapping the relative order of the two racy operations. Unlike JUMBLE, this approach requires a somewhat complex replay infrastructure, and is prone to "falling off the trace" during replay, resulting in false positives. Race-directed random testing [36] explores a similar approach, but avoids the need for a replay infrastructure. Both of these approaches assume sequential consistency and will not detect destructive race conditions as in Figures 1 and 2 (or in the `moldyn` benchmark) that cause incorrect behavior only under relaxed memory models. In particular, results from race-directed random testing [36] suggest that the race conditions in `moldyn` are benign (under the assumption of sequential consistency).

In concurrent work, Burnim *et al* also explore testing-based methodologies for relaxed memory models. For three hardware-level memory models (TSO, PSO, and PSLO), their work successfully detects violations of sequential consistency [9, 10], but does not identify which sequential consistency violations cause destructive behavior. An interesting area for future work is to adapt Jumble's adversarial memory approach to detect destructive race conditions for these memory models.

Much other work (including, for example, [27, 40]) identifies defects in multithreaded programs by exploring many (or possibly all) possible interleavings. Most of these tools assume sequential consistency. In contrast to this prior work based on scheduling non-determinism, this paper proposes a complementary approach of using memory-model non-determinism to expose destructive races.

Dynamic analyses to detect race conditions include Eraser's LockSet algorithm [35] and its refinements [30, 41], happens-before-based detectors [32], and detectors combining those two approaches, *e.g.*, [15, 33, 45]. Others have also combined dynamic analysis with a global static analysis to improve precision and performance [12, 42]. *Post-mortem* race identification techniques record program events for later analysis (see, for example, [3, 13, 34]), but might be difficult to use for long-running programs. The FASTTRACK algorithm preserves the precision of happens-before-based detectors, but with significantly improved performance [17], and the PACER algorithm uses sampling to provide increased performance, while still providing strong probabilistic coverage guarantees [6].

Many type-based and whole program static analysis techniques have been developed for identifying races in various languages, including C [16, 38], Java [1, 4, 8, 28, 43], and SPMD programs [5]. While static race detection provides the potential to detect all race conditions over all program paths, decidability limitations imply that, for all realistic programming languages, any sound static race detector is incomplete and may produce false alarms. A variety of other approaches have also been developed, including model checking [11, 27, 44].

Recent work [7] developed an operational semantics for programs under a relaxed memory model. Operational PJMM is similar in some ways (*e.g.*, in making write buffers explicit), but our specification only needs to define the legality of traces, not programs, and is somewhat less involved. In addition, whereas [7] develops a new relaxed memory model, the development of JUMBLE required an operational formulation of a subset of an existing memory model, namely the JMM.

8. Conclusions and Future Work

Race conditions are becoming increasingly problematic given the relaxed memory models implemented by modern multi-core processors and virtual machines. This work presents a promising dynamic analysis approach of using adversarial memory to expose destructive race conditions, which has proven highly effective in our experiments. Adversarial memory complements the traditional approach of exploring many or all possible thread interleavings under the assumption of sequential consistency (as in [27, 40]), and

suggests that future tools should exploit both scheduling and memory model non-determinism for detecting concurrency errors.

Acknowledgments This work was supported in part by NSF Grants 0341179, 0644130, 0707885, and 0905650. We thank Jaeheon Yi, Michael Bond, and Kathryn McKinley for comments on a draft of this paper. We also thank Doug Lea, Bill Pugh, and Sarita Adve for helping to clarify aspects of the Java Memory Model.

References

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *TOPLAS*, 28(2):207–255, 2006.

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

[3] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting data races on weak memory systems. In *ISCA*, pages 234–243, 1991.

[4] R. Agarwal and S. D. Stoller. Type inference for parameterized race-free Java. In *VMCAI*, pages 149–160, 2004.

[5] A. Aiken and D. Gay. Barrier inference. In *POPL*, pages 243–354, 1998.

[6] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: Proportional detection of data races. In *PLDI*, 2010.

[7] G. Boudol and G. Petri. Relaxed memory models: an operational approach. In *POPL*, pages 392–403, 2009.

[8] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *OOPSLA*, pages 56–69, 2001.

[9] J. Burnim, K. Sen, and C. Stergiou. Sound and complete monitoring of sequential consistency in relaxed memory models. Technical Report UCB/EECS-2010-31, EECS Department, University of California, Berkeley, 2010.

[10] J. Burnim, K. Sen, and C. Stergiou. Testing concurrent programs on relaxed memory models. Technical Report UCB/EECS-2010-32, EECS Department, University of California, Berkeley, 2010.

[11] A. T. Chamillard, L. A. Clarke, and G. S. Avrunin. An empirical comparison of static concurrency analysis techniques. Technical Report 96-084, Department of Computer Science, University of Massachusetts at Amherst, 1996.

[12] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridhara. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, pages 258–269, 2002.

[13] J.-D. Choi, B. P. Miller, and R. H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *TOPLAS*, 13(4):491–530, 1991.

[14] M. Christiaens and K. D. Bosschere. TRaDe: Data Race Detection for Java. In *International Conference on Computational Science*, pages 761–770, 2001.

[15] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A race and transaction-aware Java runtime. In *PLDI*, pages 245–255, 2007.

[16] D. R. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP*, pages 237–252, 2003.

[17] C. Flanagan and S. N. Freund. FastTrack: Efficient and precise dynamic race detection. In *PLDI*, pages 121–133, 2009.

[18] C. Flanagan and S. N. Freund. The RoadRunner dynamic analysis framework for concurrent programs. In *ACM Workshop on Program Analysis for Software Tools and Engineering*, 2010.

[19] K. Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Stanford University, 1995.

[20] D. Grossman. Type-safe multithreading in Cyclone. In *TLDI*, pages 13–25, 2003.

[21] Java Grande Forum. Java Grande benchmark suite. Available at `http://www.javagrande.org/`, 2008.

[22] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

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

[24] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *SOSP*, pages 103–116, 2007.

[25] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL*, pages 378–391, 2005.

[26] F. Mattern. Virtual time and global states of distributed systems. In *Workshop on Parallel and Distributed Algorithms*, 1988.

[27] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.

[28] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI*, pages 308–319, 2006.

[29] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *PLDI*, pages 22–31, 2007.

[30] H. Nishiyama. Detecting data races using dynamic escape analysis based on read barrier. In *Virtual Machine Research and Technology Symposium*, pages 127–138, 2004.

[31] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *TPHOLs*, pages 391–407, 2009.

[32] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multihreaded C++ programs. In *PPOPP*, pages 179–190, 2003.

[33] E. Pozniansky and A. Schuster. MultiRace: Efficient on-the-fly data race detection in multithreaded C++ programs. *Concurrency and Computation: Practice and Experience*, 19(3):327–340, 2007.

[34] M. Ronsse and K. D. Bosschere. RecPlay: A fully integrated practical record/replay system. *TCS*, 17(2):133–152, 1999.

[35] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *TOCS*, 15(4):391–411, 1997.

[36] K. Sen. Race directed random testing of concurrent programs. In *PLDI*, pages 11–21, 2008.

[37] Standard Performance Evaluation Corporation. SPEC benchmarks. `http://www.spec.org/`, 2003.

[38] N. Sterling. Warlock: A static data race analysis tool. In *USENIX Winter Technical Conference*, pages 97–106, 1993.

[39] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, pages 334–345, 2006.

[40] W. Visser and P. C. Mehlitz. Model checking programs with Java PathFinder. In *SPIN*, page 27, 2005.

[41] C. von Praun and T. Gross. Object race detection. In *OOPSLA*, pages 70–82, 2001.

[42] C. von Praun and T. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *PLDI*, pages 115–128, 2003.

[43] J. W. Voung, R. Jhala, and S. Lerner. Relay: Static race detection on millions of lines of code. In *FSE*, pages 205–214, 2007.

[44] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *POPL*, pages 27–40, 2001.

[45] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP*, pages 221–234, 2005.