Precise Data Race Detection in a Relaxed Memory Model using Model Checking

KyungHee Kim, Tuba Yavuz-Kahveci, Beverly A Sanders
Department of Computer and Information Science and Engineering
University of Florida
Gainesville, FL 30611-6120
email: {khkim, tyavuz, sanders}@cise.ufl.edu

Abstract

Most approaches to reasoning about multithreaded programs, including model checking, make the implicit assumption that the system being considered is sequentially consistent. This is not a valid assumption for most current multiprocessor/core systems and this fact is exposed to the programmer for many concurrent programming languages in the form a relaxed memory model. For example, the Java Memory Model only promises sequentially consistent behaviors for programs that are free from data races, making the ability to detect and eliminate data races essential for soundly reasoning about Java programs. Towards this end, we introduce a new summary function that captures the information necessary for precise data race detection along with an efficient representation of the function that allows data race detection by model checking. In addition, we introduce novel search heuristics specialized for data race detection that lead to shorter counterexample paths than standard search strategies. The ideas have been implemented in Java RaceFinder (JRF), an extension to the model checker Java PathFinder (JPF). In contrast to many data race detection tools that can only deal with a restricted set of concurrent programming idioms such as lock-based synchronization, JRF correctly handles programs that contain memory model relevant features including volatile fields, final fields, compareAndSwap, and static initialization in addition to both intrinsic and extrinsic locks. As a result, it is powerful enough to be effectively used with wait-free and lock-free data structures. Once a concurrent program is data race free, standard model checking techniques can be soundly used to check other properties of interest.

1 Introduction

Virtually all approaches for reasoning about the behavior of concurrent programs, both the informal reasoning practiced by programmers writing a concurrent program and formal methods and tools such as model checkers start with an assumption of sequential consistency (SC) [14]. With SC, a concurrent program behaves as if all of its atomic actions occur in some global order that is consistent with the program order on each thread. In particular, all threads "see" values written to main memory in a consistent order. Modern programming environments do not satisfy SC. Common optimizations by the compiler and the hardware which significantly speed up programs without affecting their sequential semantics, are not necessarily benign in a concurrent environment.

As an example, consider the following program fragment.

```
result = computation();
done = true;
```

The variable `done` is initially false and not accessed by `computation()`, which updates `result` and possibly has other side effects. Since the two statements are independent, the order could be reversed without changing the sequential semantics. However, if this fragment occurs in a concurrent program and `done` is intended to be a flag to other threads that `computation()` is finished, then reversing the order could result in another thread finding `done==true`, and seeing a state reflecting an incomplete execution of `computation()`. This scenario violates SC.

Architectures provide low level instructions that can be used to prevent reordering effects and are typically inserted in the object code as a result of synchronization instructions in the program's source. Although it would make concurrent programming much easier to use compiler analysis to automatically insert the necessary instructions, this currently isn't practical and the programmer is expected to insert sufficient synchronization to ensure that SC is not violated.

Exactly how threads interact with memory and how the programmer can control this is defined by a *memory model.* Traditionally, memory models have been defined for architectures, but more recently memory models have become part of a programming language's semantics. The Java memory model (JMM) [15] is an important example. In the JMM, a situation that can lead to non-SC behavior is called a *data race.*

The term data race has often been used where the definition and consequences are subtly different from those of data races in the context of the JMM. Two memory accesses by different threads on the same location are said to *conflict* when at least on is a write and a data race has been defined to be a situation where conflicting operations are not ordered by synchronization. In a sequentially consistent system, data races may indicate some sort of concurrency related nondeterminism that may or may not affect the overall correctness of the program. For example, in a SC system, for the example given earlier, the accesses to done would be considered a benign data race[1], while it is a bug with potentially serious consequences in a program executing under the JMM. Similarly, a program may be free of data races in the sense of the JMM, and thus be SC, while still containing concurrency related errors. For example, suppose a class representing bank account contains field for the balance and offers a deposit method that executes `balance = balance + amount`. In the JMM, if `balance` is marked volatile (or if each of the accesses to this field occur inside their own critical section implemented using a common lock) then the program will not have a data race, but it will still be incorrect due to the fact that the entire deposit method is not atomic. Some authors would call this error a "race"', adding to the confusion around the term. In this paper, we will use the term "data race" as defined in the JMM and discussed in more detail in section 2.1 with the goal of detecting situations that lead to non-SC behavior.

The JMM satisfies the important fundamental property for memory models [19]:

> Programs whose SC executions have no races must have only SC executions.

As a result, we can assume SC, and thus use model checking to demonstrate data race freedom. Then, for programs without data races, standard model checking techniques can *soundly* be used to find other types of concurrency related errors.

Most of the tools that have been developed to detect data races do not precisely find race conditions as defined by a well-defined memory model. Rather they identify easy to detect situations that may indicate a data race. For example, many approaches (including the data race detection extensions in the current Java PathFinder distribution) attempt to ensure that for all shared variables, all accesses to a particular variable are protected by a common lock. While this condition is sufficient for data race freedom, it is not necessary. In contrast to JRF, tools that check lock usage cannot effectively analyze programs using important idioms that are not based on locking including wait-free and lock-free algorithms.

The paper is organized as follows: In section 2, we give a brief overview the formal definition the JMM, including a new result that shows a weaker condition than complete data race freedom is sufficient to ensure SC. In section 3, we introduce a summary function that can be used to capture the necessary information to precisely recognize data races during model checking. We also describe a space efficient representation of the summary function. Section 4 introduces new search heuristics based on a careful analysis of the properties of data races. The previously mentioned ideas were implemented in a model checker by extending JPF. How this was done is described in section 5, followed by experimental results in section 6. Related work is given in section 7.

The main contributions of this work can be summarized as

- A weaker requirement for the Java program to be SC.
- A summary function that captures the necessary happens-before relation along with a soundness proof.
- An efficient representation of the summary function.
- Data race specific search heuristics.
- Our implementation of JRF provides a memory model aware extension to JPF, thus making JPF sound for relaxed memory models.
- In contrast to most data race detection tools, JRF detects data races precisely.
- In contrast to many data race detection tools, which can only deal with lock-based synchronization, JRF correctly deals with data race avoid-

[1]SC rules out reordering and the access itself is atomic.

ance using any combination of intrinsic and extrinsic locks, volatile variables, join, barriers, compareAndSet operations, and transmitting values through concurrent data structures. In particular, this allows lock-free and wait-free protocols to be soundly analyzed.

2 Foundations

2.1 The Java Memory Model

In this section, we give an overview of the formal definition of the Java Memory Model. For the most part, our treatment follows that of [2][2], which is in turn based on the original specification of the JMM given in [15].[3]

An *action* is a memory-related operation a that belongs to a single thread $\mathit{Thread}(a)$, affects the monitor or variable v, and has a *kind*, which is one of the following: volatile read from v, volatile write to v, non-volatile read from v, non-volatile write to v, lock of lock m, unlock of lock m, starting a thread t, detecting termination of thread t, and instantiating an object with volatile fields *volatiles* and non-volatile fields *fields*. All of the action kinds, with the exception of non-volatile read and non-volatile write are *synchronization actions.*

An *execution* E is given by a tuple $\langle A, P, \overset{po}{\rightarrow}, \overset{so}{\rightarrow}, W, V\rangle$ where

- A is a finite set of actions
- P is a program
- $\overset{po}{\rightarrow}$, the program order, is a partial order on A obtained by taking the union of total orders representing each thread's sequential semantics
- $\overset{so}{\rightarrow}$, the synchronization order, is a total order over all of the synchronization actions in A
- V, the value written function, assigns a value to each write
- W, the write-seen function, assigns a write action to each read action so that the value obtained by a read action r is $V(W(r))$.

Executions are subject to certain well-formedness constraints but still allow non-determinism since normal actions on different threads need not be ordered. In addition, it is not required that the write-seen function returns the "most recent" write to the variable in question or that the write-seen functions for actions on different threads are consistent, thus allowing various sorts of sequentially inconsistent behavior.

The synchronizes-with relation on actions, denoted $\overset{sw}{\rightarrow}$, is given below.

- An unlock action on a monitor lock m synchronizes-with all subsequent lock actions on m by any thread.
- A write to a volatile variable v synchronizes-with all subsequent reads of v.
- The action of starting a thread synchronizes-with the first action of the newly started thread.
- The final action in a thread synchronizes-with an action in any other thread (e.g. join, or invoking the isAlive() method) that detects the thread's termination.
- The writing of default values of every object field synchronizes-with the first access of the field.

In the descriptions above, "subsequent" is determined by the synchronization order.

Well-formedness constraints[4] on executions include unsurprising requirements such as type correctness, correct behavior of locks, and consistency with the sequential semantics of the program, and consistency of volatile reads and writes with the synchronization order. In addition, a well-formed execution satisfies *happens-before consistency* where *happens-before* order is a transitive, irreflexive partial order on the actions in an execution obtained by taking the transitive closure of the union of $\overset{sw}{\rightarrow}$ and $\overset{po}{\rightarrow}$. Happens-before consistency means that a read r of variable v is allowed to see the results of a write $w = W(r)$ provided that

- r is not ordered before w, i.e. $\neg(r \overset{hb}{\rightarrow} w)$.
- There is no intervening write w' to v, i.e. $\neg\exists w' : w \overset{hb}{\rightarrow} w' \overset{hb}{\rightarrow} r$.

[2]Our treatment explicitly handles object instantiation.

[3]The most important differences between [15] and [2] are that the latter requires that the total order for SC executions be consistent with both the synchronization order and program order (as opposed to just the program order), formulates the semantics in terms of finite executions, and ignores external actions.

[4]In addition to the well-formedness conditions, *legal* executions according to the JMM are also required to satisfy additional causality conditions that constrain the behavior of programs with data races, thus providing certain safety guarantees for program with races. Our goal is to detect data races so that they can be eliminated rather than reason about the properties of programs with data races, thus these conditions are not relevant for the work described in this paper.

Two operations *conflict* if neither is a synchronization action, they access the same memory location and at least one is a write. A *data race* is defined to be a pair of conflicting operations *not* ordered by $\xrightarrow{hb}$.

A sequentially consistent (SC) execution is one where there is a total order, $\xrightarrow{sc}$, on the actions consistent with $\xrightarrow{po}$ and $\xrightarrow{so}$ and where a read r of variable v sees the results of the most recent preceding write w.

- $w \xrightarrow{sc} r$
- There is no intervening write w' to v, i.e. $\neg\exists w' : w \xrightarrow{sc} w' \xrightarrow{sc} r$.

A Java program is *correctly synchronized* if all sequentially consistent executions are data race free.

Theorem 1 *Any legal execution of a well-formed correctly synchronized program is sequentially consistent.*

This is a property of the JMM [9, 15] and is equivalent to Theorem 1 in [2], where a proof can be found.

Theorem 1 is crucial for justifying our approach. It means that we can use a model checker, which assumes SC, to check whether the program is correctly synchronized, and if so, soundly use the model checker to check other properties as desired.

2.2 A Weakened Condition for Sequential Consistency

Data races in SC executions can be classified into three kinds: WR, WW, and RW, meaning that the conflicting operations are a write followed by a read, a write followed by a write, and a read followed by a write, respectively. To improve the time and space efficiency of JRF, information to allow detection of RW data races is not maintained. In most cases, a program that contains a RW race in one SC execution will contain a WR race on the same variable in a different SC execution, and the latter data race will be detected. In any case, the following Theorem guarantees that the approach is sound.

A Java program is *weakly correctly synchronized* if all sequentially consistent executions are data race free or contain only RW data races.

Theorem 2 *Any legal execution E of a well-formed weakly correctly synchronized programs is sequentially consistent.*

The proof of Theorem 1 in [2] is still valid with the weaker hypothesis.

3 Capturing the happens-before relation

3.1 The summary function, h

In this section, we introduce a function h that summarizes $\xrightarrow{hb}$ at each point in a SC execution, allowing data races to be detected as they occur during model checking. Let *Addr* be the set of (abstract) memory locations representing non-volatile variables in the program, *SynchAddr* be the set of (abstract) memory locations representing variables with volatile semantics and locks, and *Threads* be the set of threads.

We summarize the happens-before relation as a function $h : \mathit{SynchAddr} \cup \mathit{Threads} \rightarrow 2^{\mathit{Addr}}$ that maps threads and synchronization variables to sets of non-volatile variables so that $x \in h(t)$ means that thread t can read or write variable x without causing a WW or WR data race.

For a finite sequentially consistent execution E of program P that has a set of static non-volatile variables $\mathit{static}(P)$, let E_n be the prefix of E of length n, i.e. the sequence of actions $a_0, a_1, \ldots, a_{n-1}$, and h_n be the value of h after performing all of the actions in E_n. We assume that thread *main* is the single thread that initiates the program. Initially,

$$h_0 = \lambda z.\text{if } z = \mathit{main} \text{ then } \mathit{static}(P) \text{ else } \bot \qquad (1)$$

The way that h_{n+1} is obtained from h_n depends on the action a_n. First, we define four auxiliary functions *release*, *acquire*, *invalidate*, and *instantiate*.

The function $\mathit{release}(t, x)$ takes a summary function h and yields a new summary function by updating $h(x)$ to include the value of $h(t)$. It is used with actions by thread t that correspond to the source of a $\xrightarrow{sw}$ edge, for example, writing a volatile variable x, releasing lock x, starting thread x, etc.

$$\mathit{release}(t, x)\ h \ \widehat{=}\ h[x \mapsto h(t) \cup h(x)] \qquad (2)$$

The function $\mathit{acquire}(t, x)$ takes a summary function h and yields a new summary function obtained from h by updating $h(t)$ to include the value of $h(x)$. It is used in actions that form the destination of a $\xrightarrow{sw}$ edge, for example, reading a volatile x, locking lock x, and joining or detecting termination of thread x.

$$\mathit{acquire}(t, x)\ h \ \widehat{=}\ h[t \mapsto h(t) \cup h(x)] \qquad (3)$$

The function *invalidate* yields a new summary function by removing x from $h(z)$ for all $z \neq t$. It is used in actions where thread t writes non-volatile x.

$$\mathit{invalidate}(t, x)\ h \ \widehat{=}\ \lambda z.\text{if } (t = z) \text{ then } h(z) \text{ else } h(z) \backslash \{x\}$$

a_n by thread t	h_{n+1}
write a volatile field v	$\mathit{release}(t, v)\ h_n$
read a volatile field v	$\mathit{acquire}(t, v)\ h_n$
lock the lock variable lck	$\mathit{acquire}(t, lck)\ h_n$
unlock the lock variable lck	$\mathit{release}(t, lck)\ h_n$
start thread t'	$\mathit{release}(t, t')\ h_n$
join thread t'	$\mathit{acquire}(t, t')\ h_n$
t'.isAlive()	if $(t'.\mathit{isAlive}())(\mathit{acquire}(t, t')\ h_n)$ else h_n
write a non-volatile field x	$\mathit{invalidate}(t, x)\ h_n$
read a non-volatile field x	h_n
instantiate an object containing non-volatile fields *fields* and volatile fields *volatiles*	$\mathit{instantiate}(t, \mathit{fields}, \mathit{volatiles})\ h_n$

Figure 1. Definition of h_{n+1}

The function *instantiate* is used to incorporate newly instantiated object into the summary function. It yields a new summary by adding the set $fields$ to the value of $h(t)$ and initializing the previously undefined values of h for the new volatile variables.

$$\begin{aligned} &\mathit{instantiate}(t, \mathit{fields}, \mathit{volatiles})\ h \ \widehat{=} \qquad (4)\\ &\lambda z.\quad \text{if } (t = z) \text{ then } h(t) \cup \mathit{fields}\\ &\qquad \text{else if } (z \in \mathit{volatiles}) \text{ then}\{\}\\ &\qquad \text{else } h(z) \end{aligned}$$

We define

$$\mathit{norace}(x, t) = x \in h(t) \qquad (5)$$

The definition of h_{n+1}, which depends on h_n and action a_n, is given in Figure 1.

To detect data races during model checking, we maintain h and check $\mathit{norace}(x, t)$ before reading or writing of non-volatile x by thread t. When this condition holds for all non-volatile reads and writes in an execution, we say the execution is *h-legal.*

We can prove several facts about *h-legal* executions. In the following, we use $\mathit{last}(E)$ to denote the last element of finite sequence E and $E_n|_{w(x)\cup inst(x)}$ to denote the subsequence of actions in E_n that write to x plus the instantiation action.

The first, rather obvious lemma confirms our intuition that non-static variables belonging to objects that have not been instantiated yet have not been written in any *h-legal* execution and will serve as the base case for inductive proofs of other results.

Lemma 3 *For an SC execution E, and non-static x, if x has not been instantiated, $E_n|_{w(x)\cup inst(x)}$ is empty.*

The following two lemmas, for non-static and static variable, respectively, say that at any point in an SC execution, the most recent thread to write a non-volatile variable can access it without causing a data race.

Lemma 4 *For an SC execution E_n and non-volatile, non-static variable x, if x has been instantiated in E_n and $t = \mathit{thread}(\mathit{last}(E_n|_{w(x)\cup inst(x)}))$, then $x \in h_n(t)$.*

The proof is by induction. Since x cannot have been instantiated in E_0, the base case holds trivially. Now, assume the lemma holds for n and show that it holds for $n + 1$. There are four cases:

- x has not been instantiated in E_n and action a_n does not instantiate it. Then the lemma continues to hold trivially.
- x has not been instantiated in E_n and action a_n is an action of thread t that instantiates x. Then, $\mathit{last}(E_{n+1}|_{w(x)\cup inst(x)}) = a_n$ and from the rule for object instantiation in figure 1, $x \in h_{n+1}(t)$.
- x has been instantiated in E_n, $x \in h_n(t)$ where $t = \mathit{thread}(\mathit{last}(E_n|_{w(x)\cup inst(x)}))$ and $\mathit{thread}(a_n) = t$. From figure 1, there are no actions performed by thread t that have the effect of removing items from $h_n(t)$ to obtain $h_{n+1}(t)$.
- x has been instantiated in E_n, $x \in h_n(t)$ where $t = \mathit{thread}(\mathit{last}(E_n|_{w(x)\cup inst(x)}))$ and $\mathit{thread}(a_n) \neq t$. From figure 1, either a_n writes to x falsifying $t = \mathit{thread}(\mathit{last}(E_n|_{w(x)}))$, or $x \in h_{n+1}(t)$.

Lemma 5 *For an SC execution E_n and non-volatile, static variable x, $x \in h_n(t)$ where $t =$ (if $E_n|_{w(x)} \neq \Theta$ then $\mathit{thread}(\mathit{last}(E_n|_{w(x)})$ else main)*

Proof: Again the proof is by induction. Since x is static, $x \in h_0(\mathit{main})$ and the base case holds. Now, assume the lemma holds for E_n and show that it holds for E_{n+1}. There are four cases.

- $x \in h_n(t)$, $\mathit{thread}(a_n) = t$.
- $x \in h_n(t)$ and $\mathit{thread}(a_n) \neq t$.

The next lemma forms the basis of our soundness proof. In this lemma, $a \xrightarrow{hb=} b$ indicates either $a \xrightarrow{hb} b$ or $a = b$.

Lemma 6 *Let E be a well-formed, h-legal SC execution. Then for all non-volatiles x, all threads t, all volatiles v, and all n*

$$\begin{array}{lcl} x \in h_n(t) & \Rightarrow & E_n|_{w(x) \cup inst(x)} \xrightarrow{hb=} last(E_n|_t) \\ \quad \wedge & & \\ x \in h_n(v) & \Rightarrow & E_n|_{w(x) \cup inst(x)} \xrightarrow{hb=} last(E_n|_{w(v)}) \end{array}$$

where for set S, $S \xrightarrow{hb=} s$ means $(\forall s' : s' \in S : s' \xrightarrow{hb=} s)$.

The proof is given in appendix A.

The lemma tells us that if $x \in h_n(t)$, then all the preceding writes to x happen-before the latest action on thread t. Since there are no writes to x between $last(E_n|_t)$ and a_n, by program order, $last(E_n|_t) \xrightarrow{hb} a_n$, and by transitivity all writes to $x \xrightarrow{hb} a_n$. As a result, if a_n is a write by thread t to x, the action can be performed without causing a WW data race.

The next theorem, which follows easily from Lemma 6 justifies our approach.

Theorem 7 *If all SC executions of a well-formed program are h-legal, then the program is weakly correctly synchronized, and all of its legal executions are SC.*

3.2 Representation of h

In this section, we describe a representation of h that is suitable for implementation in a model checking tool. This context requires space efficiency, efficient updating, and, since model checking involves backtracking, a way to efficiently save and restore previous incarnations. We take advantage of the fact that in Java, threads and locks are also objects and handle elements of *SynchAddr*, *Thread*, and *Addr* uniformly as "memory locations".

Recall that h maps *SynchAddr* $\cup$ *Thread* to 2^{Addr}. This mapping is implemented as an array of bit vectors. Each *SynchAddr* $\cup$ *Threads* is given unique index,[5] conceptually corresponding to a row, and *Addr* an index conceptually corresponding to a column. Then $norace(t, x)$ holds when $h[row_index(t), column_index(x)] == 1$.

[5] Each memory location is given a unique key constructed from the name of the class, the instance number, and the field name (or array index if the location in question is an array element). The keys are in turn mapped to the corresponding index in the bit vector for elements of *Addr*, and the array holding the bit vectors for elements of *SynchAddr* $\cup$ *Thread*.

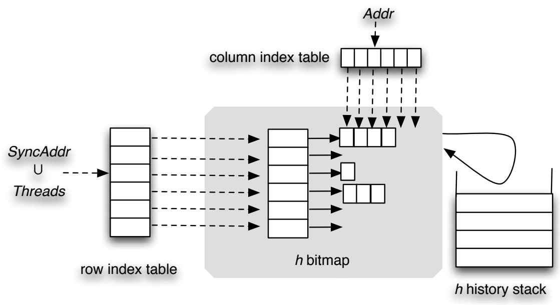


Figure 2. Internal representation of h

When an element of *SynchAddr* $\cup$ *Threads* is created, an element of the array of bit vectors is reserved. When an element of *Addr* is created, an index is assigned, but space in the corresponding bit vector is not allocated until it is actually used. This implies that the bit vectors are dynamically resized. The *acquire* and *release* operations can be implemented as a single set union step, and the *norace* check, which is used most often, can be done with bit masking. The *invalidate* operation, however, involves a number of rows of set operations. Fortunately, $|$*SynchAddr* $\cup$ *Threads*$|$ is far less than $|$*Addr*$|$. In our experiments, the former is on the order of tens, while the latter can easily exceed thousands.

The changes in h are stored in a separate stack in order to restore the value to an earlier state when the model checker backtracks. Figure 2 illustrates the bitmap for h, and a history stack used to store Δs. The Δs of h includes the changes following *acquire*, *release*, *invalidate*, and allocation including reuse of garbage collected memory and instantiation in a static initializer. In addition, the changes in *prior_release* and *written_by* information, explained in section 5, are also stored in h history stack. During state backtracking, these are undone to recover h to previous values.

4 Data race specific search heuristics

Model checking is a way to verify a program correctness by exhaustively exploring all possible states of a multithreaded program. At each state, a model checker chooses the next state from all possible candidates and checks the properties that should be satisfied at that point in the program. When there are no candidates for the next states left, it backtracks to check any remaining scheduling sequences.

The order that states are traversed during model checking influences how many states must be visited before errors are found. Since a data race requires the interaction of two threads, a search strategy with more thread interleaving is likely to find a data race earlier than a depth-first search strategy (DFS) which has minimum interleaving through sequential scheduling of threads at the beginning. Rather than simply increasing thread interleaving, we also consider the nature of data races and propose the following heuristics, which depend on the current value of h, to choose the next state. In the descriptions, *curr* refers to the current thread.

- **Writes-first(WF):** Although a data race may occur at either a read or a write, the source of a data race involving memory location m is a write of m that causes an *invalidate* operations to remove m from $h(t)$ for all $t \neq curr$. Any future read or write to m by some other thread results in a data race unless an appropriate synchronization action has occurred. Thus this heuristic prioritizes write operations.

- **Watch-written(WW):** If there has been a write on a memory location, m, it is possible that a future read or write on m by another thread will result in a data race. Thus, this heuristic prioritizes operations on a memory location that has recently been written by a different thread.

- **Avoid *release/acquire*(ARA):** A data race free program involves appropriately located matching *release* and *acquire* operations. Although programs with data races may also have *acquire* and *release* operations, existence of these on a path may indicate a lower probability of existence of a data race. This heuristic prioritizes operations on threads that do not have a recent acquire operation preceded by a matching release on the execution path.

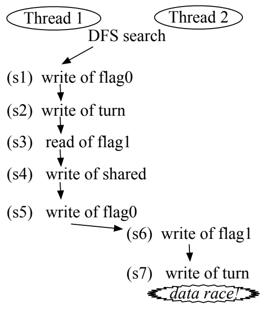
- **Acquire-first(AF):** When an *acquire* operation is executed after a matching *release*, a happens-before edge is created on the current path. However, if the *acquire* is executed before the matching *release* statement then this does not result in an happens before edge and cannot prevent a data race. This heuristic prioritizes acquire operations that do not have a matching release along the execution path. This situation often corresponds to situations of unsafe publication of an otherwise correctly synchronized object.[6]

[6]Publication of an object is the act of making its reference visible to other threads. Unsafe publication (section 3.5 of [8]) can lead to a partially constructed object becoming visible to other threads.

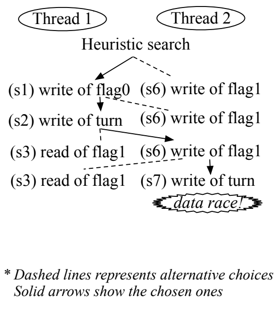
```
Initially, flag0 = flag1 = turn = shared = 0;
/* all fields are non-volatile */

       Thread 1                Thread 2
======================   =====================
s1: flag0 = 1;           s6:flag1 = 1;
s2: turn = 1;            s7:turn = 0;
s3: while (flag1==1 \&    s8:while (flag0==1 \&
     turn==1) {/*spin*/}      turn==0) {/*spin*/}
s4: shared++;            s9:shared++;
    /*critical section*/    /*critical section*/
s5: flag0 = 0;           s10:flag1 = 0;
```

Figure 3. One iteration of Peterson's Algorithm



(a) DFS algorithm



(b) HEURISTIC algorithm

Figure 4. The model checking of Peterson's algorithm using different search strategies

The fragment of well known Peterson's algorithm in Figure 3 shows the advantage of the WF heuristic over depth-first search.

The search space using DFS for this example is shown in Figure 4a, and WF heuristic search in Figure 4b. DFS-based model checking takes seven states to find the data race on *turn* while the heuristic search finds the same race after visiting only four states. The counter example path is shorter and easier to analyze.

Figure 5 shows the heuristic search algorithm used in JRF. It stores the states in a max priority queue, where a priority is the sum of the heuristic value and the depth of a given state times the max heuristic value ($MAX{=}WRITE_{written\_by\_other}$). The heuristic values only affect the choice of a next state among children of current state. Once the next state is chosen and advanced, its newly generated children always have higher priority than the states remaining in the queue. This guarantees that the highest priority children and their descendants will be explored first. The search algorithm visits the states in descending priority order until it reaches an error state or no more states are left.

The main purpose of those heuristics is to follow paths which minimize the happens-before orderings since it is the lack of happens-before orderings that causes data races. The heuristics can be used together and the heuristic search algorithm given in Figure 6 can be configured in various ways by turning on and off each of the 4 heuristics: *WF*, *WW*, *AF*, *ARA*. The example seen earlier in Figure 4b shows the search path with only *write-first* configured. For example, if the choice of heuristic is *writes-first* and *watch-written*, the only available heuristic values are $WRITE_{written\_by\_other}$, $WRITE_{written\_by\_self}$, $READ_{written\_by\_other}$, $READ_{written\_by\_self}$, and $OTHER$.

In comparison with DFS, the heuristic search algorithm requires more memory. A model checker using DFS only advances and backtracks while heuristic search generates and stores all possible child states before advancing and later restores them. Although only Δs of h related data are stored during state advance and backtrack, a complete copy of the h is saved when the child states are generated. Heuristic search also tends to take more time because it visits more states in general. As we can see from Figure 4, DFS only generates 7 states to find the race, but heuristic search generates 8 states since it takes into consideration all children of current state to decide next one. The advantage of heuristic search is that it tends to find data races with a shorter counter example path than DFS in most cases. This is important because reasoning about the cause of a data race using the counterexample path is not a straight-forward task in our experience, especially when the length of the path is fairly large. Experimental results are given in section 6.

```
AlgAlgorithm HeuristicSearch
    $Q$: max priority queue
    $s, s'$: state
    $value$: integer
    $Q \leftarrow$ empty
    put $(initial\ state, max\ integer)$ into $Q$
    while $Q$ not empty do
        $(s, value) \leftarrow$ remove from $Q$
        if $s$ is an error state then
            print("error")
            break
        for each successor $s'$ of $s$ do
            if $s'$ is not marked then
                mark $s'$
                put $(s', HeuristicValue(s') + depth * MAX)$ into $Q$
```

Figure 5. Heuristic search algorithm. States are prioritizes based on their heuristic value as computed in Figure 6 and the search depth.

```
Algorithm HeuristicValue ($s$: state)
    $v$: variable
    if $s$ reached via write to non-volatile $v$ then
        if $v$ most recently written by another thread then
            return 8 /*$WRITE_{written\_by\_other}$ (WF $\vee$ WW)*/
        else return 7 /*$WRITE_{written\_by\_self}$ (WW) */
    if $s$ reached via read from non-volatile $v$ then
        if $v$ most recently written by another thread then
            return 6 /*$READ_{written\_by\_other}$ (WF) */
        else return 5 /*$READ_{written\_by\_self}$ (WF $\wedge$ WW) */
    if $s$ reached via read from a volatile $v$
    or locking an object not released before then
        return 4 /*$ACQUIRE_{without\_prior\_release}$ (ARA) */
    if $s$ reached via read from a volatile $v$
    or locking an object released before then
        return 2 /*$ACQUIRE_{with\_prior\_release}$ (ARA $\vee$ AF) */
    if $s$ reached via volatile write
    or unlocking an object then
        return 1 /*$RELEASE$ (AF) */
    else return 3 /*$OTHER$ (ALL) */
```

Figure 6. Algorithm for deciding heuristic values for states based on their likelihood of leading to a data race. Heuristic values becomes available according to the heuristics (WF, WW, ARA, AF) presented in Section 4.

5 Extending Java Pathfinder

Java PathFinder(JPF) [13] is a model checker for Java byte code. It reads Java class files and simulates program execution using its own virtual machine with on-the-fly verification of specified properties. We have extended JPF so that we can precisely detect data races using the summary function, h, and search heuristics described earlier. This section describes implementation details.

5.1 The Listener Implementation

JPF supports a Listener interface which can be used to extend its functionality. The interface provides a set of callback functions allowing low level operations such as object creation, object locking and unlocking, the start of a new thread, and execution of instructions to be intercepted and augmented with user-supplied codes. In our case, this code maintains h as described

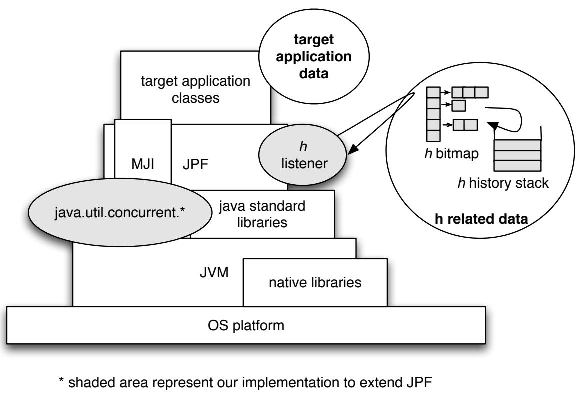


Figure 7. System Structure

in section 3.2. The operations *acquire*, *release*, *invalidate* and asserting *norace* are performed as appropriate when execution of memory model related instructions occur.

The structure of the system is shown in Figure 7. The Listener lies at the same level as other JPF code outside of the target model classes.

To implement the heuristic search algorithm described in section 4, we use the following two auxiliary functions:

- $\mathit{written\_by} : \mathit{Addr} \times \mathit{Thread} \rightarrow \mathit{Boolean}$
- $\mathit{prior\_release} : \mathit{SynchAddr} \rightarrow \mathit{Boolean}$

$\mathit{written\_by}(m, t)$ returns *true* when the most recent write to the given memory location m was done by the thread t, and $\mathit{prior\_release}(m)$ returns *true* when the given memory location m was released at least once previously.

We have discussed elsewhere [18] how our approach can be used with standard assertional reasoning techniques to (in principle) give a formal proof that a program is free from data races. In an earlier incarnation of our tool, we annotated the byte code with these assertions and then checked for assertion violations using standard JPF. The listener-based approach to extending JPF described above proved to be both more efficient and flexible.

5.2 Using the Model Java Interface

Java programs rely on a number of platform dependent functions including thread implementation and low level synchronization primitives that are implemented in native codes. The model java interface(MJI) is provided to allow JPF to handle these situations: native codes are executed by the host JVM and not model checked by JPF. Unfortunately, this means that the h data structures are no longer correct after executing native codes. Fortunately, the number of native functions that are relevant to the memory model is small enough that it is feasible to modify their model java interfaces to reflect the way the target functions update h. In particular, we modified the MJI code for *sun.misc.Unsafe*, which is heavily used by *java.util.concurrent* packages, to include the necessary calls to h manipulation code. For completeness, it was necessary to also extend MJI to include the missing classes from the *java.util.concurrent.atomic* package including all array versions of atomic data structures. Many lock-free data structures use these classes. The current implementation of JRF correctly handles all java language features related to the JMM except for finalizers.

5.3 Problems with State Backtracking

There were several issues with state backtracking that made the task of extending JPF less straightforward than one might expect.

Static initializers complicate the management of h since the state backtracking scheme built into JPF does not reload classes and re-initialize their static fields, even when the state is backtracked to a point before the class was loaded. Thus, it is necessary to identify which memory locations are allocated in a static initializer. These locations, if they have not been updated, should be accessible by all threads, including those created later without causing a data race. We maintained another set of *Addr*, $h(\mathit{static\_initializer})$, where *static_initializer* is a synthetic thread representing the class loader thread. This represents the locations corresponding to static variables that have not yet been updated outside of the static initializer for the class. In addition to the $\mathit{release}(\mathit{parent\_thread}, \mathit{child\_thread})$, the thread start operation should also performs $\mathit{release}(\mathit{static\_initializer}, \mathit{child\_thread})$.

Another complication in the listener implementation is the unpredictable garbage collection in JPF. When an object is garbage collected during state search, it is no longer in use along that path and might be reused for a new object. The problematic situation occurs when the unique key for the object is no longer unique. The original object, which is still used in other stored paths, shares its key with the new object, resulting in incorrect sharing of h. State backtracking will use the h of the new object unless it is properly restored to the value before garbage collection. The h history stack stores the necessary garbage collection and reallocation information.

5.4 Allowing trusted classes and benign races

A final field can be safely excluded from data race checking if the object containing it is properly constructed.[7]

The JMM provides a strong guarantee, namely sequential consistency for properly synchronized programs. It also constrains programs with data races in order to provide some minimal security guarantees. These guarantees include type safety and the guarantee that there are no "out-of-thin-air" values. As a result, it is possible, in principle, for programmers to write and reason about the correctness of programs that contain data races, and in some cases the presence of these races will not cause the program to violate its specification. Doing so is quite difficult and is considered to be a job for experts only. Most programmers should write race-free programs. JRF does not support reasoning about programs with races in the sense that one cannot construct a program with data races, submit it to JRF, and expect the model checker to have explored the states that only occur in non-SC executions. In order to deal with intentional races that are trusted to be benign, JRF allows specification of individual locations so that data races involving these locations will be ignored.

It is also possible to designate certain classes as trusted. JRF does not maintain necessary information to check for data races involving non-volatile variables in these classes. It does however, continue to track happens-before edges involving volatile fields and locks defined in the class in order to continue to detect data races in other classes precisely. For example, a common way of safely publishing objects is to make them available to other threads by passing them through a data structure such as a queue whose implementation is provided in the *java.util.concurrent* package where the insertion of an object happens-before removal of the object. If the package is marked as trusted, the happens-before edges related to inserting and removing objects will be preserved, but no checking will be done on the internal non-volatile variables in the class. By marking the classes in the standard java release as trusted, a significant reduction in the time and space requirements to model check an application class can be achieved.

[7]An object is "properly constructed" if the "this" reference does not escape the constructor.

6 Experimental Results

Our extension to JPF[8] make it possible to soundly analyze complex, highly concurrent data structures that do not necessarily use locking, or use a mixture of locking and other synchronization idioms. In order for JPF to be sound when applied to these programs, the program must be free of data races. Lock-free programs typically use volatile variables along with instances of classes in the *java.util.concurrent.atomic* package to create the necessary happens-before edges to prevent data races.

6.1 Performance

To evaluate the usefulness of JRF and empirically explore the behavior of the search heuristics, we used it to analyze an extensive set of concurrent data structures using a wide variety of synchronization approaches. Figure 8 shows empirical results comparing the behavior of heuristic search with DFS for a selection of examples, all containing data races identified by JRF.

Testing was performed on a 2.53 GHz Intel Core 2 Duo processor with Mac OS X/10.5.7, JPF version 4 and Java 1.6 with 2GB JVM heap memory. The states column contains the total number of states searched, the length column contains the transition length of a data race path. Each transition may be composed of multiple instructions. The *h* size column summarizes the number of memory locations kept in the *h* bitmap; this is indicative of the extra data structures maintained by listener.

The first group of test programs were examples from the acclaimed textbook by Herlihy and Shavit [10]. Java implementations were obtained from the book's web site. The table contains detailed data from a few examples with data races. The broader results from our tests on these examples were extremely encouraging about the value of JRF. A total of 65 programs were tested, with errors identified in 16. These included data races as well as other typical concurrency errors such as deadlock and atomicity errors. In addition, JRF revealed a benign data race in the *java.util.concurrent.AbstractOwnableSynchronizer* (which is used in the implementation of *java.util.concurrent.locks.ReentrantLock*), and a non-benign race in the *java.util.Math* class (a reported

[8]The standard JPF distribution includes two race detecting tools: RaceDetector and PreciseRaceDetector. Both, section 7 explains more, do not correctly deal with programs where data races are avoided by using volatile variable or classes from the *java.util.concurrent.atomic* package.

example(threads,iteration)	DFS					Heuristic				
	state	length	time	mem	h size	state	length	time	mem	h size
barrier.DisBarrier (2,2)	109	109	2s	12M	35*640	79	39	2s	21M	35*640
barrier.StaticTreeBarrier (7,1)	123	123	4s	19M	51*813	407	102	5s	123M	51*898
hash.StripedHashSet (4,12)	35	35	3s	12M	83*820	213	26	4s	60M	91*820
lists.LazyList (4,16)	87	87	3s	19M	65*603	360	44	3s	63M	56*582
lists.OptimisticList (3,6)	47	47	2s	12M	55*598	229	37	3s	39M	52*592
mutex.Bakery (2,2)	53	42	5s	12M	37*794	79	37	4s	21M	37*742
mutex.LockFreeQueue (2,2)	31	27	1s	12M	28*441	37	17	1s	12M	28*439
spin.CLHLock (4,8)	106	98	2s	12M	37*293	106	48	2s	20M	37*293
spin.CorrectedMCSLock (2,4)	289	136	3s	12M	41*537	107	33	2s	20M	41*537
IteratorTest(LockFreeQueue) (2,2)	33	33	17s	21M	213*1817	57	30	16s	36M	205*1817
IteratorTest(EBDeque) (2,2)	25	25	18s	35M	253*2030	41	22	16s	34M	245*2030
IteratorTest(LockFreeList) (2,2)	37	37	16s	21M	214*1747	65	34	15s	35M	206*1747
QueueTest (2,2)	91	81	4s	20M	72*1367	180	85	5s	59M	70*1309
moldyn (2,1)	1884	1884	48s	34M	36*629	1894	949	333s	703M	36*717
sor (2,15)	76	76	4s	12M	41*721	84	44	3s	22M	41*721
lufact (2,5)	26	26	1s	12M	34*309	32	18	1s	12M	34*309
montecarlo (2,5)	105	94	13s	30M	63*3265	29	15	4s	21M	50*1449

Figure 8. Comparison of DFS and heuristic(ARA,AF,WF,WW) search

bug in the Java bug database). The second group of examples are from the Amino Concurrent Building Blocks project (version 0.5.3) [1]. The last four examples in the table are from the Java Grande Forum (JGF) Benchmark Suite Thread Version 1.0.[12].

Figure 9 summarizes the result of JPF PreciseRaceDetector for examples in Figure 8. PreciseRaceDetector found one false race on a volatile field, and missed races on arrays declared as volatile but their element are not volatiles. In addition, it has an application exception due to the incomplete *java.util.concurrent.atomic* package. JRF heuristic search outperforms PreciseRaceDetector and gives shorter counter-example paths in most cases: all cases other than *barrier.StaticTreeBarrier* gives shorter counterexample paths.

We tested various configurations of the heuristic choices, and the results show that using all four heuristics or (WW,WF) perform best for most cases: (WW,WF) configuration gives 13 best solutions for our selected 17 test cases, which is one more than (ARA,AF,WF,WW), but gives longest path for *moldyn*. The experiment results with different heuristic configurations are given in appendix B.

6.2 Overhead

In order to show the overhead of JRF compared with standard JPF, we checked two types of examples without data races; three without any errors, and one with application program error so JPF reports property violation. The first example, Simple, is a variation of the example discussed in Section 1.

```
/* thread 1 */
result = 1;
done = true; //done is volatile

/* thread 2 */
if ( done )
  assert (result==1);
```

The second example, Peterson, is Peterson's algorithm from Figure 3 with all the variables declared to be volatile. The last two examples are from [10]. SenseBarrierTest has no error but the final example, TourBarrierTest, throws an ArrayIndexOutOfBoundsException which both JPF and JRF report as a violation of the NoUncaughtExceptionsProperty. Figure 10 summarizes the result. The default of DFS was used for both cases.

7 Related Work

Most existing tools to detect race conditions are limited in ways that make them unsuitable for the examples we discussed in this paper. Many tools for data race detection or avoidance do not start from the memory model and its definition of a data race, but instead look for conditions that imply data race freedom and are easier to check. For example, lockset-based approaches attempt to ensure that all accesses to a particular shared variable are protected by a common lock.

Eraser [20] is the seminal example of this approach and uses it to implement dynamic race detection. Eraser predates programming languages with well-defined memory models. Although based on the idea that accesses to shared variables should be consistently

example(threads,iterations)	state	length	time	mem	result
barrier.DisBarrier(2,2)	100	100	1s	12M	same race
barrier.StaticTreeBarrier(7,1)	102	102	1s	12M	same race
hash.StripedHashSetTest(4,12)	312	171	1s	13M	same race
lists.LazyList(4,16)	2491	689	3s	21M	same race
lists.OptimisticList(3,6)	1069	202	2s	12M	same race
mutex.Bakery(2,2)	1519	374	3s	22M	**different race found***
mutex.LockFreeQueue(2,2)	93	26	1s	12M	same race
spin.CLHLock(4,8)	263	94	1s	12M	same race
spin.CorrectedMCSLock(2,4)	282	129	1s	12M	false race on volatile field
IteratorTest(LockFreeQueue)(2,2)	67	28	3s	21M	same race
IteratorTest(EBDeque)(2,2)	0	0	1s	12M	*NoSuchMethodException***
IteratorTest(LockFreeList)(2,2)	68	33	2s	22M	same race
QueueTest (2,2)	165	80	2s	12M	same race
moldyn(2,1)	11386982	58035	3115s	1020M	**no race found***
sor(2,15)	2287	499	2s	20M	**no race found***
lufact(2,5)	2781	675	2s	18M	**no race found***
montecarlo(2,5)	1012	87	4s	21M	same race

*The results in **boldface** are failure to detect races on volatile array with non-volatile element accesses

***java.util.concurrent.atomic.AtomicReferenceArray* is not implemented in JPF

Figure 9. Execution result of PreciseRaceDetector

example(threads,iteration)	result	Original JPF				DFS				
		state	length	time	mem	state	length	time	mem	h size
Simple (2,)	no error	12	4	1s	7M	12	4	1s	11M	29*92
Peterson (2,2)	no error	866	46	1s	11M	866	46	3s	11M	30*82
barrier.SenseBarrierTest (2,2)	no error	3329	94	3s	12M	3329	94	19s	16M	36*629
barrier.TourBarrierTest (3,3)	other error*	182	182	1s	12M	182	182	2s	12M	37*698

**ArrayIndexOutOfBoundsException* caused by application program

Figure 10. Overhead measurement

locked, it did allow the thread that creates an object to modify it without holding a lock until some other thread accesses it. It did not check for safe publication of the reference to the object, however, and would thus be unsound if applied to the current JMM.

The standard JPF distribution includes two race detecting tools, RaceDetector and PreciseRaceDetector. The former implements the lockset algorithm based on the assumption that every shared field is protected by a common lock. This lacks the ability to check happens-before ordering by other than locking. The latter checks a data race based on a generic definition of a race rather than starting with the JMM. At each choice generator, it checks if there are more than two thread choices trying to access the same memory location and at least one of them is an update access. Read and write accesses to a volatile field are also detected as a race. Both JPF race detectors report a race for the code in Figure 3 with all fields declared as volatiles. Our tool successfully verified race freedom of this modified version of Figure 3. While the semantics of the lock primitive in Java imply that consistent locking is sufficient for data race freedom, it is not necessary, and requiring it rules out increasingly important programming idioms including wait-free and lock-free algorithms.

Static race detection tools typically sacrifice completeness, in the sense that they can only deal with a particular set of programming idioms. Some tools deliberately sacrifice soundness as well, failing to identify certain data races. For example, the Chord [17] designers made scalability an important design goal and the system can handle lexically-scoped lock-based synchronization, fork/join synchronization, and wait/notify. It starts by constructing a superset of possible conflicting operations, then filters this set using a sequence of analyses, and reports a possible data race for all remaining pairs. Although Chord is both unsound and incomplete, it seems to be extremely well-engineered and has shown significant utility in practice by identifying numerous concurrency related errors in a set of widely used, large, open source programs. Another example is the rcc checker[7] that has been recently resurrected and extended for the Mobius project[4]. This tool uses a type theory-based approach (which requires annotations by the users) to ensure that locking is done correctly. In its most recent incarnation, it also recognizes that volatile variables do not need to be protected by locks to avoid data races. Both chord and rcc would find a false data race in the example program `Simple` in section 6.2. Neither would be particularly useful for the types of algorithms in the tests in section 6.

The tool most closely related to ours is Goldilocks[6]. Goldilocks is a dynamic analysis tool using an algorithm based on a relation that is very similar to the inverse of h. In other words, the Goldilocks algorithm maintains a function for each variable that indicates which threads can access the variable. As with all tools performing dynamic analysis, the required instrumentation of the program may change its behavior and the tool is limited to analyzing paths that happen to be tested.

Recently, several studies [3, 5, 11] have incorporated memory model awareness to model checking. The approaches presented in [3, 11] can verify sequential consistency. [3] considers a hardware-level memory model and uses bounded model checking and a stateless model checker, CHESS [16]. Therefore, they use vector-clocks to capture the happen-before relation. Since JPF is a state-based model checker, we can store the happen-before information for each state. [11] considers C#'s memory model and a bytecode-level state-based model checker tailored for C#. The technique presented in [5] guides the model checker in generating a subset (i.e., under approximates the JMM) of program executions varying due to instruction reordering allowed in the JMM. However, their tool does not detect data races. JRF can verify sequential consistency of a Java program without generating all such subsets via assertional reasoning but cannot handle reasoning about programs that are not sequentially consistent.

8 Conclusion

We have described an approach, based on maintaining a function summarizing the happens-before relation that can be used in a model checker to precisely detect data races. In addition, we introduced new search heuristics based on a careful analysis of data races that leads to shorter and easier-to-understand counterexample paths. The ideas have been implemented in JRF, an extension of JPF that detects data races which is important since standard JPF is unsound for programs that contain data races. JRF has been shown to be useful on a wide range of important concurrent data structures. In contrast to most other approaches, JRF is precise and can deal with the wealth of synchronization actions in the Java programming language.

References

[1] Amino concurrent building blocks. http://amino-cbbs.sourceforge.net/.

[2] D. Aspinall and J. Sevcik. Formalising Java's data-race-free guarantee. In *TPHOLs 2007 (LNCS)*, volume 4732, pages 22–37. Springer, 2007.

[3] S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In *CAV '08: Proceedings of the 20th international conference on Computer Aided Verification*, pages 107–120, Berlin, Heidelberg, 2008. Springer-Verlag.

[4] M. Consortium. Deliverable d3.3: Preliminary report on thread-modular verification, March 2007. http://mobius.inria.fr.

[5] A. De, A. Roychoudhury, and D. D'Souza. Java memory model aware software validation. In *PASTE '08: Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 8–14, New York, NY, USA, 2008. ACM.

[6] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware java runtime. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 245–255, New York, NY, USA, 2007. ACM.

[7] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 219–232, New York, NY, USA, 2000. ACM Press.

[8] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice.* Addison Wesley Professional, 2006.

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification.* Addison Wesley, 3rd edition, 2005.

[10] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, 2008.

[11] T. Q. Huynh and A. Roychoudhury. Memory model sensitive bytecode verification. *Form. Methods Syst. Des.*, 31(3):281–305, 2007.

[12] The Java Grande Forum benchmark suite. http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html.

[13] Java Pathfinder. http://javapathfinder.sourceforge.net/.

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

[15] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 378–391, New York, NY, USA, 2005. ACM Press.

[16] M. Musuvathi, S. Qadeer, and T. Ball. Chess: A systematic testing tool for concurrent software. Technical Report MSR-TR-2007-149, Microsoft Research, 2007.

[17] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319, New York, NY, USA, 2006. ACM Press.

[18] B. A. Sanders and K. Kim. Assertional reasoning about data races in relaxed memory models. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008.

[19] V. A. Saraswat, R. Jagadeesan, M. Michael, and C. von Praun. A theory of memory models. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 161–172, New York, NY, USA, 2007. ACM Press.

[20] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

A Proof of Lemma 6

Lemma 8 *Let E be a well-formed, h-legal SC execution. Then for all non-volatiles x, all threads t, and all volatiles v, and all n*

$$\begin{array}{rcl} x \in h_n(t) & \Rightarrow & E_n|_{w(x)\cup inst(x)} \xrightarrow{hb=} last(E_n|_t) \\ & \wedge & \\ x \in h_n(v) & \Rightarrow & E_n|_{w(x)\cup inst(x)} \xrightarrow{hb=} last(E_n|_{w(v)}) \end{array}$$

where for set S, $S \xrightarrow{hb} s$ means $(\forall s' : s' \in S : s' \xrightarrow{hb=} s)$.

Proof: The proof is by induction. For the base case, we have E_0, which does not contain any actions and where $h(main) = static(P)$ and h is undefined for all other arguments. Since E_0 is empty, the right sides of both implications are trivially true. Now, we assume the property for E_n and show it holds for E_{n+1}. We must consider each action kind.

- Write a volatile field v by thread t. $h_{n+1}(t) = h_n(t)$, and due to program order, and the hypothesis, $E_n|_{w(x)\cup inst(x)} \xrightarrow{hb} last(E_n; write(v,t)|_t)$. The second requirement is similar for $x \in h_n(v)$. For the new additions, we already have $x \in h_n(t) \rightarrow E_n|_{w(x)\cup inst(x)} \xrightarrow{hb} last(E_n; write(v,t)|_t)$. Since this write will become $last(E_n|_{w(v)})$, the condition is reestablished.

- Read a volatile field v by thread t.This has the effect of adding the elements satisfying $x \in h_n(v)$ to $h_{n+1}(t)$. Since for any such x, all writes of $x \xrightarrow{hb}$ the latest write of v, which $\xrightarrow{hb}$ the read which is the current action on thread t, and thus becomes $last(E_{n+1}|_t)$, thus reestablishing the condition.

- Lock the lock lck. The situation is analogous to reading a volatile

- Unlock the lock lck. The situation is analogous to writing a volatile

- Start thread t'. This case is somewhat strange since it involves two threads, and one of them does not have any actions. To deal with this, we consider that the first action of a thread is a synthetic action where the start action by the starting thread happens-before this action. Then, by transitivity of $\xrightarrow{hb}$, the first condition is reestablished.

- Join thread t'. This case follows easily from the transitivity of $\xrightarrow{hb}$.

- Detecting termination t' with t'.isAlive(). Similar to join

- Write a non-volatile field x by thread t. In this case, both conditions are violated–it need no longer be the case, for example, that that all writes of x happen before the last statement of thread t'. The solution is to remove x from $h_{n+1}(t')$, thus reestablishing the condition. Since we require $x \in h(t)$ before performing the write, the required condition for thread t will be maintained.

- Read a non-volatile field x This does not affect the second condition, and extending E_n with a read operation preserves the first.

- Instantiate an object containing non-volatile fields by thread t non-volatile fields *fields* and volatile fields *volatiles* The instantiate adds the new non-volatiles to $h_{n+1}(t)$ while extending E_n with the instantiate action, thus preserving the first condition. The second condition is not affected by merely defining a new $h(v)$ to be empty.

B Empirical comparison of different heuristic configurations

The experimental results for the examples described in section 6 with various heuristic configurations follow. The **boldface** lengths are the shortest counter-example paths. In a few cases, *spin.CLHLock*, the three amino examples, and *moldyn* the all-heuristic configuration is not the best choice.

example(threads,iteration)	configuration	state	length	time	mem	h table
barrier.DisBarrier(2,2)	AF	77	**39**	3s	21M	35*640
	ARA	85	44	3s	21M	35*640
	WW	87	44	3s	22M	35*640
	WF	87	44	4s	22M	35*640
	ARA,AF	77	**39**	3s	21M	35*640
	WW,WF	87	44	3s	22M	35*640
	WW,ARA	87	44	3s	21M	35*640
	WW,AF	79	**39**	2s	22M	35*640
	WF,ARA	87	44	3s	22M	35*640
	WF,AF	79	**39**	3s	22M	35*640
	ARA,AF,WW	79	**39**	3s	22M	35*640
	ARA,AF,WF	79	**39**	3s	22M	35*640
	ARA,WF,WW	87	44	2s	22M	35*640
	AF,WF,WW	79	**39**	2s	22M	35*640
	ARA,AF,WF,WW	79	**39**	2s	21M	35*640
barrier.StaticTreeBarrier(7,1)	AF	379	**102**	4s	114M	51*813
	ARA	379	**102**	5s	71M	51*813
	WW	407	**102**	6s	123M	51*898
	WF	407	**102**	8s	123M	51*898
	ARA,AF	379	**102**	5s	71M	51*813
	WW,WF	407	**102**	6s	123M	51*898
	WW,ARA	407	**102**	5s	123M	51*898
	WW,AF	407	**102**	5s	123M	51*898
	WF,ARA	407	**102**	5s	123M	51*898
	WF,AF	407	**102**	5s	123M	51*898
	ARA,AF,WW	407	**102**	5s	122M	51*898
	ARA,AF,WF	407	**102**	7s	123M	51*898
	ARA,WF,WW	407	**102**	5s	123M	51*898
	AF,WF,WW	407	**102**	5s	123M	51*898
	ARA,AF,WF,WW	407	**102**	5s	123M	51*898
hash.StripedHashSet (4,12)	AF	263	35	3s	64M	83*820
	ARA	263	35	4s	64M	83*820
	WW	269	33	9s	97M	107*820
	WF	356	44	10s	97M	83*820
	ARA,AF	263	35	4s	64M	83*820
	WW,WF	213	**26**	6s	60M	91*820
	WW,ARA	269	33	8s	97M	107*820
	WW,AF	269	33	8s	97M	107*820
	WF,ARA	356	44	7s	97M	83*820
	WF,AF	356	44	7s	105M	83*820
	ARA,AF,WW	269	33	8s	97M	107*820
	ARA,AF,WF	356	44	7s	97M	83*820
	ARA,WF,WW	213	**26**	5s	60M	91*820
	AF,WF,WW	213	**26**	4s	57M	91*820
	ARA,AF,WF,WW	213	**26**	4s	60M	91*820

example(threads,iteration)	configuration	state	length	time	mem	h table
lists.LazyList (4,16)	AF	679	87	4s	127M	65*603
	ARA	679	87	4s	127M	65*603
	WW	2425	302	58s	690M	116*610
	WF	800	99	7s	219M	65*603
	ARA,AF	679	87	4s	126M	65*603
	WW,WF	360	**44**	4s	73M	56*582
	WW,ARA	2425	302	54s	1056M	116*610
	WW,AF	2425	302	53s	689M	116*610
	WF,ARA	800	99	5s	219M	65*603
	WF,AF	800	99	5s	219M	65*603
	ARA,AF,WW	2425	302	54s	690M	116*610
	ARA,AF,WF	800	99	5s	219M	65*603
	ARA,WF,WW	360	**44**	3s	67M	56*582
	AF,WF,WW	360	**44**	3s	73M	56*582
	ARA,AF,WF,WW	360	**44**	3s	63M	56*582
lists.OptimisticList (3,6)	AF	270	47	3s	69M	55*598
	ARA	270	47	3s	69M	55*598
	WW	619	102	22s	174M	92*592
	WF	331	54	5s	68M	55*598
	ARA,AF	270	47	3s	69M	55*598
	WW,WF	229	**37**	3s	38M	52*592
	WW,ARA	619	102	20s	172M	92*592
	WW,AF	619	102	20s	174M	92*592
	WF,ARA	331	54	3s	68M	55*598
	WF,AF	331	54	3s	68M	55*598
	ARA,AF,WW	619	102	20s	174M	92*592
	ARA,AF,WF	331	54	3s	68M	55*598
	ARA,WF,WW	229	**37**	3s	38M	52*592
	AF,WF,WW	229	**37**	3s	40M	52*592
	ARA,AF,WF,WW	229	**37**	3s	39M	52*592
mutex.Bakery (2,2)	AF	1071	351	29s	339M	41*901
	ARA	1071	351	30s	338M	41*901
	WW	1079	358	32s	304M	41*901
	WF	1091	358	33s	383M	41*901
	ARA,AF	1071	351	30s	337M	41*901
	WW,WF	82	**37**	7s	34M	40*767
	WW,ARA	1067	351	30s	310M	41*901
	WW,AF	1067	351	30s	310M	41*901
	WF,ARA	1079	354	29s	344M	41*901
	WF,AF	1079	351	30s	344M	41*901
	ARA,AF,WW	1067	351	30s	316M	41*901
	ARA,AF,WF	1079	351	31s	344M	41*901
	ARA,WF,WW	1062	351	30s	311M	41*901
	AF,WF,WW	1062	351	30s	312M	41*901
	ARA,AF,WF,WW	79	**37**	4s	21M	37*742

example(threads,iteration)	configuration	state	length	time	mem	h table
mutex.LockFreeQueue (2,2)	AF	56	27	1s	12M	28*439
	ARA	56	27	1s	12M	28*439
	WW	37	18	1s	12M	28*439
	WF	37	**17**	1s	12M	28*439
	ARA,AF	56	27	1s	12M	28*439
	WW,WF	37	**17**	1s	12M	28*439
	WW,ARA	37	18	1s	12M	28*439
	WW,AF	37	18	1s	12M	28*439
	WF,ARA	37	**17**	1s	12M	28*439
	WF,AF	37	**17**	1s	12M	28*439
	ARA,AF,WW	37	18	1s	12M	28*439
	ARA,AF,WF	37	**17**	1s	12M	28*439
	ARA,WF,WW	37	**17**	1s	12M	28*439
	AF,WF,WW	37	**17**	1s	12M	28*439
	ARA,AF,WF,WW	37	**17**	1s	12M	28*439
spin.CLHLock (4,8)	AF	293	98	3s	38M	37*293
	ARA	293	98	3s	38M	37*293
	WW	171	39	3s	35M	37*291
	WF	163	**35**	2s	35M	37*289
	ARA,AF	293	98	2s	38M	37*293
	WW,WF	106	48	2s	20M	37*293
	WW,ARA	171	39	2s	35M	37*291
	WW,AF	171	39	3s	35M	37*291
	WF,ARA	163	**35**	1s	35M	37*289
	WF,AF	163	**35**	1s	35M	37*289
	ARA,AF,WW	171	39	2s	35M	37*291
	ARA,AF,WF	163	**35**	2s	35M	37*289
	ARA,WF,WW	106	48	2s	20M	37*293
	AF,WF,WW	106	48	2s	20M	37*293
	ARA,AF,WF,WW	106	48	2s	20M	37*293
spin.CorrectedMCSLock (2,4)	AF	121	**33**	2s	22M	41*537
	ARA	106	35	1s	22M	41*537
	WW	119	**33**	2s	36M	41*537
	WF	140	**33**	2s	34M	41*537
	ARA,AF	121	**33**	1s	22M	41*537
	WW,WF	119	44	2s	20M	41*537
	WW,ARA	116	35	2s	35M	41*537
	WW,AF	119	**33**	2s	36M	41*537
	WF,ARA	137	35	2s	34M	41*537
	WF,AF	140	**33**	2s	34M	41*537
	ARA,AF,WW	119	**33**	2s	36M	41*537
	ARA,AF,WF	140	**33**	2s	34M	41*537
	ARA,WF,WW	106	35	2s	20M	41*537
	AF,WF,WW	107	**33**	2s	20M	41*537
	ARA,AF,WF,WW	107	**33**	2s	20M	41*537

example(threads,iteration)	configuration	state	length	time	mem	h table
IteratorTest(LockFreeQueue)(2,2)	AF	57	30	15s	36M	205*1817
	ARA	57	30	15s	36M	205*1817
	WW	32	**16**	11s	36M	194*1756
	WF	32	**16**	12s	36M	194*1756
	ARA,AF	57	30	15s	36M	205*1817
	WW,WF	32	**16**	12s	36M	194*1756
	WW,ARA	57	30	15s	36M	205*1817
	WW,AF	57	30	15s	36M	205*1817
	WF,ARA	57	30	15s	36M	205*1817
	WF,AF	57	30	15s	36M	205*1817
	ARA,AF,WW	57	30	15s	36M	205*1817
	ARA,AF,WF	57	30	16s	36M	205*1817
	ARA,WF,WW	57	30	15s	36M	205*1817
	AF,WF,WW	57	30	15s	36M	205*1817
	ARA,AF,WF,WW	57	30	16s	36M	205*1817
IteratorTest(EBDeque)(2,2)	AF	41	22	16s	35M	245*2030
	ARA	41	22	17s	35M	245*2030
	WW	24	**12**	13s	35M	234*1969
	WF	24	**12**	13s	35M	234*1969
	ARA,AF	41	22	16s	35M	245*2030
	WW,WF	24	**12**	13s	35M	234*1969
	WW,ARA	41	22	16s	35M	245*2030
	WW,AF	41	22	17s	35M	245*2030
	WF,ARA	41	22	16s	35M	245*2030
	WF,AF	41	22	17s	35M	245*2030
	ARA,AF,WW	41	22	17s	35M	245*2030
	ARA,AF,WF	41	22	17s	35M	245*2030
	ARA,WF,WW	41	22	16s	35M	245*2030
	AF,WF,WW	41	22	17s	35M	245*2030
	ARA,AF,WF,WW	41	22	16s	34M	245*2030

example(threads,iteration)	configuration	state	length	time	mem	h table
IteratorTest(LockFreeList)(2,2)	AF	65	34	15s	33M	206*1747
	ARA	65	34	15s	33M	206*1747
	WW	42	21	11s	34M	195*1686
	WF	42	21	11s	33M	195*1686
	ARA,AF	65	34	15s	33M	206*1747
	WW,WF	42	**18**	12s	33M	195*1684
	WW,ARA	65	34	15s	34M	206*1747
	WW,AF	65	34	15s	33M	206*1747
	WF,ARA	65	34	15s	33M	206*1747
	WF,AF	65	34	15s	33M	206*1747
	ARA,AF,WW	65	34	17s	34M	206*1747
	ARA,AF,WF	65	34	15s	34M	206*1747
	ARA,WF,WW	65	34	14s	33M	206*1747
	AF,WF,WW	65	34	15s	34M	206*1747
	ARA,AF,WF,WW	65	34	15s	35M	206*1747
QueueTest(2,2)	AF	169	**85**	5s	59M	70*1309
	ARA	169	**85**	5s	58M	70*1309
	WW	180	**85**	5s	56M	70*1309
	WF	180	**85**	5s	59M	70*1309
	ARA,AF	169	**85**	5s	59M	70*1309
	WW,WF	180	**85**	5s	59M	70*1309
	WW,ARA	180	**85**	5s	59M	70*1309
	WW,AF	180	**85**	5s	59M	70*1309
	WF,ARA	180	**85**	5s	57M	70*1309
	WF,AF	180	**85**	5s	56M	70*1309
	ARA,AF,WW	180	**85**	5s	56M	70*1309
	ARA,AF,WF	180	**85**	5s	56M	70*1309
	ARA,WF,WW	180	**85**	5s	56M	70*1309
	AF,WF,WW	180	**85**	5s	59M	70*1309
	ARA,AF,WF,WW	180	**85**	5s	59M	70*1309

example(threads,iteration)	configuration	state	length	time	mem	h table
moldyn (2,1)	AF	1894	949	334s	684M	36*717
	ARA	1894	949	331s	684M	36*717
	WW	1892	**948**	31s	358M	36*653
	WF	1892	**948**	31s	356M	36*653
	ARA,AF	1894	949	344s	684M	36*717
	WW,WF	1890	1352	41s	319M	36*629
	WW,ARA	1894	949	332s	713M	36*741
	WW,AF	1894	949	332s	713M	36*741
	WF,ARA	1894	949	330s	713M	36*741
	WF,AF	1894	949	331s	713M	36*741
	ARA,AF,WW	1894	949	350s	713M	36*741
	ARA,AF,WF	1894	949	341s	713M	36*741
	ARA,WF,WW	1894	949	1559s	708M	36*717
	AF,WF,WW	1894	949	339s	708M	36*717
	ARA,AF,WF,WW	1894	949	333s	703M	36*717
sor (2,15)	AF	84	**44**	2s	22M	41*721
	ARA	84	**44**	2s	22M	41*721
	WW	84	**44**	3s	22M	41*721
	WF	84	**44**	3s	22M	41*721
	ARA,AF	84	**44**	2s	22M	41*721
	WW,WF	84	**44**	3s	22M	41*721
	WW,ARA	84	**44**	2s	22M	41*721
	WW,AF	84	**44**	2s	22M	41*721
	WF,ARA	84	**44**	3s	22M	41*721
	WF,AF	84	**44**	2s	22M	41*721
	ARA,AF,WW	84	**44**	3s	22M	41*721
	ARA,AF,WF	84	**44**	3s	22M	41*721
	ARA,WF,WW	84	**44**	2s	22M	41*721
	AF,WF,WW	84	**44**	2s	22M	41*721
	ARA,AF,WF,WW	84	**44**	3s	22M	41*721

example(threads,iteration)	configuration	state	length	time	mem	h table
lufact (2,5)	AF	32	**18**	1s	12M	34*309
	ARA	32	**18**	1s	12M	34*309
	WW	32	**18**	1s	12M	34*309
	WF	32	**18**	2s	12M	34*309
	ARA,AF	32	**18**	1s	12M	34*309
	WW,WF	32	**18**	2s	12M	34*309
	WW,ARA	32	**18**	1s	12M	34*309
	WW,AF	32	**18**	1s	12M	34*309
	WF,ARA	32	**18**	1s	12M	34*309
	WF,AF	32	**18**	1s	12M	34*309
	ARA,AF,WW	32	**18**	1s	12M	34*309
	ARA,AF,WF	32	**18**	1s	12M	34*309
	ARA,WF,WW	32	**18**	1s	12M	34*309
	AF,WF,WW	32	**18**	1s	12M	34*309
	ARA,AF,WF,WW	32	**18**	1s	12M	34*309
montecarlo (2,5)	AF	189	94	13s	99M	60*1993
	ARA	189	94	13s	99M	60*1993
	WW	29	**15**	5s	21M	50*1449
	WF	29	**15**	5s	21M	50*1449
	ARA,AF	189	94	13s	98M	60*1993
	WW,WF	29	**15**	4s	20M	50*1449
	WW,ARA	29	**15**	4s	21M	50*1449
	WW,AF	29	**15**	5s	20M	50*1449
	WF,ARA	29	**15**	4s	21M	50*1449
	WF,AF	29	**15**	4s	20M	50*1449
	ARA,AF,WW	29	**15**	5s	21M	50*1449
	ARA,AF,WF	29	**15**	5s	21M	50*1449
	ARA,WF,WW	29	**15**	4s	21M	50*1449
	AF,WF,WW	29	**15**	4s	21M	50*1449
	ARA,AF,WF,WW	29	**15**	4s	21M	50*1449