Hybrid Dynamic Data Race Detection

Robert O'Callahan
IBM T. J. Watson Research Center
roca@us.ibm.com

Jong-Deok Choi
IBM T. J. Watson Research Center
jdchoi@us.ibm.com

ABSTRACT

We present a new method for dynamically detecting potential data races in multithreaded programs. Our method improves on the state of the art in accuracy, in usability, and in overhead. We improve accuracy by combining two previously known race detection techniques — *lockset-based detection* and *happens-before-based detection* — to obtain fewer false positives than lockset-based detection alone. We enhance usability by reporting more information about detected races than any previous dynamic detector. We reduce overhead compared to previous detectors — particularly for large applications such as Web application servers — by not relying on happens-before detection alone, by introducing a new optimization to discard redundant information, and by using a "two phase" approach to identify error-prone program points and then focus instrumentation on those points. We justify our claims by presenting the results of applying our tool to a range of Java programs, including the widely-used Web application servers Resin and Apache Tomcat. Our paper also presents a formalization of lockset-based and happens-before-based approaches in a common framework, allowing us to prove a "folk theorem" that happens-before detection reports fewer false positives than lockset-based detection (but can report more false negatives), and to prove that two key optimizations are correct.

Categories and Subject Descriptors

D.2.4 [**Software**]: Software/Program Verification

General Terms

Verification

Keywords

Java dynamic race detection happens-before lockset hybrid

1. INTRODUCTION

A *data race* occurs in a multithreaded program when two threads access the same memory location with no ordering constraints enforced between the accesses, such that at least one of the accesses is a write [20]. In many cases, a data race is a programming error. Furthermore, programs with data races are notoriously difficult to debug because they can exhibit different behaviors when executed repeatedly with the same set of inputs. Because of the detrimental effects of data races on the reliability and comprehensibility of multithreaded software, it is widely recognized that tools for automatic detection of potential data races can be valuable. As a result, there has been a substantial amount of past work in building tools for analysis and detection of potential data races [1, 10, 13, 16, 17, 21, 22, 23].

Previous work on dynamic data race detectors focused on two approaches. One approach is *lockset-based detection*, where a potential race is deemed to have occurred if two threads access a shared memory location without holding a common lock [22, 21]. (The race is "potential" because the two threads may not, in fact, have interfered with each other.) This approach can be implemented with very low overhead, at least when whole program static analysis is available [9]. Unfortunately perfectly valid and race-free code can violate the locking hypothesis, leading to false positives in the output of a tool. Another approach is *happens-before* detection, where a potential race is deemed to have occurred if two threads access a shared memory location and the accesses are *causally unordered* in a precise sense as defined by Lamport [18]. This approach produces fewer false positives than lockset-based detection – none, in fact – in the sense that for every potential race it reports, there is an alternative thread schedule where the two accesses happen "simultaneously". Unfortunately happens-before race detection has proven difficult to implement efficiently. The best implementation to date slows down benchmark Java programs by a factor of five compared to running the programs in an interpreter [10]. Theoretical results suggest happens-before detection will resist great improvements in efficiency [7]. We also show below that happens-before detection produces more false negatives than lockset-based detection. (Of course, being dynamic techniques, both approaches produce false negatives because they only consider a subset of possible program executions.)

Our goal is to combine these approaches into a hybrid algorithm which uses happens-before techniques to reduce the false positives reported by a lockset-based race detector, but otherwise preserves the performance advantages of the lockset-based detector. The idea of combining happens-before and lockset approaches surfaced briefly 12 years ago [11] but, to our knowledge, it has never been implemented and hence the issues in making it practical have never been explored, nor have the benefits been measured.

Figure 1 shows a Java program with a potential race, adapted from a situation we discovered in a real program. `Main.execute` starts a `ChildThread` and then tries to terminate the child thread just before returning, if the child thread has not already terminated.

PPoPP'03, June 11–13, 2003, San Diego, California, USA.

```
// MAIN THREAD
class Main {
  int globalFlag;
  ChildThread childThread;
  void execute() {
    globalFlag = 1;
    childThread = new ChildThread(this);
    childThread.start();
    ...
    synchronized (this) {
      if (childThread != null) {

L:      childThread.interrupt();
      }
    }
  }
}
```

```
// CHILD THREAD
class ChildThread extends Thread {
  Main main;



  ChildThread(Main main) { this.main = main; }
  void run() {
    if (main.globalFlag == 1) ...;
    ...

    main.childThread = null;

  }
}
```

Figure 1: A Program With A Potential Race

A race arises when the `ChildThread` sets `main.childThread` to null while `execute` is at label `L`; in this case the program will crash. This race may occur very rarely and could be very difficult to detect during normal testing. A lockset-based detector would observe that even during a non-crashing test run, the `childThread` field is accessed by `ChildThread.run()` and `Main.execute` with no common lock held, and conclude that those accesses constitute a potential race. Unfortunately, a lockset-based detector would also report a potential race on accesses to `main.globalFlag` for similar reasons. On the other hand, a happens-before detector would recognize that "`globalFlag = 1;`" in `Main.execute` must "happen before" the child thread is started, and therefore before the test of `globalFlag` in method `childThread.run`, and therefore there is no potential race there. This kind of synchronization constraint can be represented using happens-before race detection but not using lockset-based detection, and thus our hybrid detector eliminates this false positive and reports only the real race.

In this paper we describe the following advances over the state of the art:

- We demonstrate a significant reduction in the number of false positives produced by lockset-based detection, at little additional computational expense, by adding a limited amount of happens-before detection.
- We collect more information to help diagnose suspected races, including stack traces for both events in a pair of racing events.
- We improve the scalability of race detection by eliminating the need for whole-program static analysis used in prior work [9, 21]. Instead we obtain low overhead with dynamic optimizations, including a novel dynamic optimization ("oversized-lockset", see Section 5.4) and by running the program twice with different instrumentation in each run.

These features require new optimizations beyond those previously used to obtain efficient lockset-based detection [9, 21]. In particular we extend and generalize the notion of *event redundancy* [9]. We introduce a new optimization which exploits the fact that the maximum number of locks held by a Java thread is very small for most programs.

We present the results of applying our race detection tool to a wide range of programs, including the Apache Tomcat Web application server. Our results confirm our claims: our detector has much lower overhead than previous happens-before detectors, reports significantly fewer false positives than previous lockset-based detectors, and is more scalable than any detector of comparable efficiency, largely because it does not rely on any static analysis.

The rest of the paper is organized as follows. Section 2 describes the basics of lockset-based race detection and formalizes them in framework which allows us to compare with happens-before detection and prove that certain optimizations are correct. Section 3 describes "happens-before" data race detection in the same framework and proves that it reports a subset of the races reported by lockset-based detection. Unfortunately this algorithm is extremely expensive. In section 4 we present the hybrid algorithm which addresses the performance problems. Section 5 presents the high-level optimizations we use to make the algorithm practical for real Java programs. We discuss implementation details in section 6. Experimental results for the overhead and accuracy of our tool appear in section 7. Finally, Section 8 describes related work, and Section 9 contains our conclusions.

2. LOCKSET-BASED RACE DETECTION

In this section we explain a previous approach to lockset-based race detection [9] in a formal framework which allows us to compare it to happens-before detection and also to verify that certain optimizations are correct.

2.1 Program Execution Model

The analysis techniques in this paper are fully dynamic; race detection does not access the program code, but only observes a stream of events generated by instrumentation inserted into the program. Thus we treat the program as an abstract machine which outputs a sequence of events $\langle e_i \rangle$ to our detector.

Each event contains components which we abstract to the following types:

- $\mathcal{M}$: a set of *memory locations*. In Java, a memory location is a non-static field of a particular object, a static field, or an element of an array; these are the only mutable locations which can be accessed by more than one thread.
- $\mathcal{L}$: a set of *locks*. In Java, a lock corresponds to an object.

- $\mathcal{T}$: a set of *threads*. In Java, a thread corresponds to an object of class `Thread`.

- $\mathcal{G}$: a set of *message IDs*. In Java, an example of a message is an object that is synchronized on using `wait()` and `notify()`.

- $\mathcal{A} = \{\text{READ}, \text{WRITE}\}$: the two possible *access types* for a memory access.

Program execution generates the following kinds of events:

- *Memory access* events of the form $\text{MEM}(m, a, t)$ where $m \in \mathcal{M}$, $a \in \mathcal{A}$ and $t \in \mathcal{T}$. These indicate that thread t performed an access of type a to location m. In Java these correspond to reading and assigning the values of static and non-static fields, and reading and assigning array elements.

- *Lock acquisition* events of the form $\text{ACQ}(l, t)$ where $l \in \mathcal{L}$ and $t \in \mathcal{T}$. These indicate that thread t acquired lock l. (Java locks are reentrant; we only generate ACQ when t did not already hold the lock.) In Java these correspond to entering a `synchronized` method or block where t did not already hold the lock.

- *Lock release* events of the form $\text{REL}(l, t)$ where $l \in \mathcal{L}$ and $t \in \mathcal{T}$. These indicate that thread t released lock l and no longer holds the lock. In Java these correspond to leaving a `synchronized` method or block where t usage count on the lock decreases to zero.

- *Thread message send* events of the form $\text{SND}(g, t)$ where $t \in \mathcal{T}$ and $g \in \mathcal{G}$. These indicate that thread t is sending a message g to some receiving thread.

- *Thread message receive* events of the form $\text{RCV}(g, t)$ where $t \in \mathcal{T}$ and $g \in \mathcal{G}$. These indicate that a thread t has received a message g from some sending thread and may now be unblocked if it was blocked before.

Thread message events are only observed by the happens-before detector, discussed in Section 3.

To simplify the presentation, we assume the abstract machine is sequential. At each step, it chooses a single thread to run, and executes that thread for some quantum, possibly generating one or more events. Thus events are observed by our detector in a sequence which depends on the thread schedule. Our implementation uses locks inside the detector to map Java thread execution into this sequential abstraction.

2.2 Accumulating Locksets

Before performing lockset-based detection, we must compute the set of locks held by a thread at any given time.

Given an access sequence $\langle e_i \rangle$, we compute the locks before step i by a thread t, $L_i(t)$, as

$$\begin{aligned} L_i(t) \quad = \quad & \{\, l \mid \exists a.a < i \wedge e_a = \text{ACQ}(l, t) \\ & \wedge (\, \nexists r.a < r < i \wedge e_r = \text{REL}(l, t))\} \end{aligned}$$

The "current lockset" for each thread, $L_i(t)$ for each live thread t, can be efficiently maintained online as acquisition and release events are received.

2.3 The Lockset Hypothesis

Lockset-based detection relies on the following hypothesis: *Whenever two different threads access a shared memory location, and one of the accesses is a write, the two accesses are performed holding some common lock.* The postulated lock ensures mutual exclusion for the two accesses to the shared location. A potential race is deemed to have occurred whenever this hypothesis is violated. Formally, given an input sequence $\langle e_i \rangle$,

$$\begin{aligned} & \text{IsPotentialLocksetRace}(i, j) = \\ & \quad e_i = \text{MEM}(m_i, a_i, t_i) \wedge e_j = \text{MEM}(m_j, a_j, t_j) \\ & \quad \wedge t_i \neq t_j \wedge\ m_i = m_j \wedge (a_i = \text{WRITE} \vee a_j = \text{WRITE}) \\ & \quad \wedge L_i(t_i) \cap L_j(t_j) = \emptyset \end{aligned}$$

For example, in Figure 1, the statement "`childThread.interrupt()`" generates a memory access with location `main.childThread`, type READ, thread MAIN, and lockset $\{$`main`$\}$. The statement "`main.childThread`" generates a memory access on `main.childThread` with type WRITE, thread CHILD and lockset $\emptyset$. Therefore IsPotentialLocksetRace will be true for these two events.

2.4 Lockset-Based Detection

Because the number of races is potentially quadratic in the number of memory accesses, we cannot report all races, nor would that be useful in practice. Instead our tool reports one race for each memory location m on which at least one potential race is detected. This simplification creates opportunities for many optimizations.

To check IsPotentialLocksetRace for all access to a given memory location m, it suffices to store a set of (a, t, L) tuples with the access type, thread, and current lockset for each access to m. Since we only need to detect one race, multiple accesses with identical (a, t, L) tuples are redundant and only one tuple need be recorded. Therefore our basic detection algorithm processes a $\text{MEM}(m, a, t)$ event by first checking to see whether the (a, t, L) tuple is already present for m. If it is already present, the new access is ignored. Otherwise if (a, t, L) forms a potential race with any prior tuple (a_p, t_p, L_p) according to IsPotentialLocksetRace, a race is reported and we stop detecting races on m. Otherwise we add (a, t, L) to the tuple set for m.

In practice we perform many optimizations to improve this algorithm. The space requirements of the tuple set, and the cost of detecting duplicate and racing tuples, can be significantly reduced by carefully choosing the representation of the tuple set, but for the sake of brevity this paper does not describe those choices. Other optimizations are described below.

3. HAPPENS-BEFORE RACE DETECTION

Unfortunately violations of the lockset hypothesis are not always programming errors. Programmers can and do write safe multi-threaded code which mutates shared data without specific locks protecting the data. One common example is programs which use *channels* to pass objects between threads in the style of CSP [15]. In such programs thread synchronization and mutual exclusion are accomplished by explicit signaling between threads.

Figure 2 shows an example of object recycling, a common technique for reducing object allocation and initialization costs in large programs. Object recycling often leads false positives in a lockset-based detector. The problem is that thread A and thread B both access `myBig` without holding locks, and thus the accesses are reported as races by the lockset-based detector. However races are in fact prevented because exclusive ownership of the BigObject is

```
// THREAD A
BigObject big = new BigObject();
big.init(...);
...
bigPool.recycle(big);

// THREAD B
BigObject myBig = bigPool.removeOne();
myBig.init(...);
...
bigPool.recycle(myBig);
```

Figure 2: Sharing Objects Between Threads Using Pools

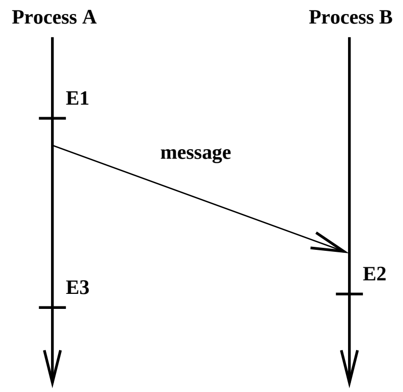

Figure 3: Happens-Before Example

transferred when the object reference itself is transferred between threads via the object pool.

Happens-before based race detection can take account of inter-thread signaling both explicit (e.g., using the Java primitives `notify` and `wait`) and implicit (e.g., via data structures such as the pool above). In this section we define Lamport's happens-before relation [18] and the (previously known) technique of happens-before race detection [10] in our framework. We name this approach *pure* happens-before detection, to contrast it with the *limited* happens-before detection that our hybrid algorithm actually implements. The difference between the pure and limited algorithms is described in Section 4.1.

3.1 The Happens-Before Relation

The *happens-before relation* was defined by Lamport as a partial order on events occuring in a distributed system [18]. We transfer this relation to our setting by treating threads as processes and the events observed by our race detector as the events occurring in the system.

The basic idea of the happens-before relation is that a pair of events (e_i, e_j) are related if communication between processes allows information to be transmitted from e_i to e_j. In Figure 3, event E1 happens-before event E2, but E2 and E3 are unrelated.

Given an event sequence $\langle e_i \rangle$, the happens-before relation $\rightarrow$ is the smallest relation satisfying the following conditions:

- If e_i and e_j are events in the same thread, and e_i comes before e_j, then $i \rightarrow j$. Formally,

$$\text{Thread}(e_i) = \text{Thread}(e_j) \wedge i < j \implies i \rightarrow j$$

- If e_i is the sending of message g and e_j is the reception of g, then $i \rightarrow j$.

$$e_i = \text{SND}(g, t_1) \wedge e_j = \text{RCV}(g, t_2) \implies i \rightarrow j$$

- Happens-before is transitively closed.

$$i \rightarrow j \wedge j \rightarrow k \implies i \rightarrow k$$

The happens-before relation is defined over the event indices i because two occurrences of the same event may have different happens-before relationships.

Following Lamport we can also say that $a \rightarrow b$ means it is possible for a to causally affect b.

3.2 Thread Messages

The definition above depends on the definition of the set of messages that can be transmitted between threads. We consider a message to be sent from thread t_1 to thread t_2 whenever t_1 takes an action that later affects t_2. One action can affect multiple threads, and therefore each send may have multiple receivers (or none).

Thread message events capture explicit synchronization between Java threads, such as that performed by Java's `start()`, `join()`, `wait()`, `notify()` and `notifyAll()` mechanisms. When a thread t_1 starts thread t_2, a unique message g is generated and two events occur, a SND(g, t_1) and a RCV(g, t_2). Similarly when thread t_1 calls t_2`.join()` and t_2 terminates, events SND(g, t_2) and RCV(g, t_1) are generated. When t_1 calls obj`.notify()` on an object, we generate an event SND(g, t_1), and for all threads t_2 waiting on obj (having called obj`.wait()`), RCV(g, t_2) is generated.

For full happens-before detection, we need some additional thread messages to capture thread interactions arising from shared memory access and locking. When a thread t_1 writes to a shared memory location, we generate a fresh message g and follow the MEM(m, WRITE, t_1) with a SND(g, t_1). Each time a thread t_2 subsequently reads or writes m, we generate an event RCV(g, t_2) after the read or write, using the g sent by the most recent writer to the location. This models the fact that information transmitted by MEM(m, WRITE, t_1) can influence the timing of events following MEM(m, READ, t_2).

Similarly, after a thread t_1 releases a lock, we generate SND(g, t_1), and the next thread t_2 to acquire the lock first generates RCV(g, t_2). (The first acquisition of a lock does not receive any message.)

Tracking all memory and locking messages is very expensive [10] and in practice we do not track them; this is discussed further in section 4.

3.3 Happens-Before Race Detection

In principle happens-before race detection is very simple: we say that a potential race has occurred if we observe two distinct events e_i and e_j that access the same memory location, where at least one event is a write, and neither i happens-before j nor j happens-before i, i.e., there is no possible causal relationship ordering i and j. Formally:

$$\begin{aligned}
&\text{IsPotentialHBRace}(i, j) = \\
&\quad \wedge\, e_i = \text{MEM}(m_i, a_i, t_i) \wedge e_j = \text{MEM}(m_j, a_j, t_j) \\
&\quad \wedge\, t_i \neq t_j \wedge\ m_i = m_j \wedge (a_i = \text{WRITE} \vee a_j = \text{WRITE}) \\
&\quad \wedge\, \neg(i \rightarrow j) \wedge \neg(j \rightarrow i)
\end{aligned}$$

The happens-before relation can be computed on-line using standard *vector clocks* [12, 19], based on Lamport clocks [18]. Each thread t_1 maintains a *vector clock* indexed by thread IDs; t_1's vector clock's entry for thread t_2 holds a logical timestamp indicating the last event in t_2 that could have influenced t_1. We also assign a vector clock to each message, which captures the vector clock state of the sending thread at the time the message was sent. We define the vector clock $V_i(t)$ of thread t at the completion of event i, and the vector clock $V(g)$ of message g, as follows:

$$\begin{aligned}
V_0(t)(t) &= 1 \\
V_0(t_1)(t_2) &= 0, t_1 \neq t_2 \\
V_i(t)(t) &= V_{i-1}(t)(t) + 1, \quad e_{i-1} = \text{SND}(g,t) \\
V_i(t_1)(t_2) &= \max(V_{i-1}(t_1)(t_2), V(g)(t_2)), \\
& e_i = \text{RCV}(g,t_1) \wedge t_1 \neq t_2 \\
V_i(t_1)(t_2) &= V_{i-1}(t_1)(t_2), \quad \text{otherwise} \\
V(g) &= V_i(t), \quad e_i = \text{SND}(g,t)
\end{aligned}$$

It can be shown that $i \to j$ if and only if $V_j(t_j)(t_i) \geq V_i(t_i)(t_i) \wedge i < j$, where $t_i = \text{Thread}(e_i)$ and $t_j = \text{Thread}(e_j)$.

Maintaining the vector clocks costs $O(|T|)$ work per message send and receive, where $|T|$ is the number of threads in the system that have ever lived. Vector clocks also require $O(|T|)$ space for each active thread and for each sent message which could be received in the future (*e.g.,* per live shared memory location). However, the cost of actually checking the happens-before relation is $O(1)$.

Some of the costs can be reduced using clever techniques [10], but the overhead remains high. Charron-Bost [7] shows that vector clocks are asymptotically space-optimal for completely characterizing the happens-before relation in the worst case.

3.4 Example

Consider Figure 1. The only inter-thread happens-before relationships that will be produced are that "`globalFlag = 1`" happens-before "`if (globalFlag)`" and the following statements, and that "`childThread.start()`" happens-before "`ChildThread.run()`" and the following statements. Therefore a happens-before detector will never report a race on `globalFlag`, but it will report a race between "`main.childThread = null`" and "`if (childThread != null)`".

3.5 Relationship Between Happens-Before Detection and Lockset Detection

If a happens-before detector detects a race between two events, then it is possible to determine feasible alternative thread schedules where the events happen consecutively in time, in either order (unless there are hidden inter-thread dependencies due to the actions of non-Java code or quirks in the virtual machine). Therefore we can say that a happens-before detector reports only "real races". This is a desirable property for a bug-finding tool[1].

In fact, we can show that the races reported by a full happens-before detector are a subset of the races reported by lockset-based detection.

THEOREM 1. *Let $\langle e_i \rangle$ be an event sequence. Suppose happens-before detection reports a race between events e_i and e_j. We show that lockset-based detection would also report a race between e_i and e_j.*

The proof is omitted for the sake of brevity.

However, a full happens-before detector also fails to detect some real bugs that would be detected by a lockset-based detector. For

[1]However, not every real race corresponds to a bug; some people can write correct code that does not depend on the order of accesses to shared memory. Examples are discussed in section 7.

```
// THREAD A
globalInt = 7;
synchronized (clockLock) { clock++; }

// THREAD B
synchronized (clockLock) { clock++; }
int x = globalInt;
```

Figure 4: Race Hidden By Happens-Before Detection

example, consider a program that frequently updates some global counter, such as in Figure 4. The reads and writes to this counter induce happens-before relations between events in the threads that are not truly synchronized, such as the accesses to `globalInt`, causing this bug to not be revealed by a full happens-before detector. The core problem is that not all happens-before relationships correspond to true synchronization.

4. HYBRID RACE DETECTION

Maintaining vector clocks for every shared memory location and every lock is too expensive in practice. Pure happens-before detection can also result in a small number of bugs being found, because too many spurious inter-thread messages are generated. Therefore we have implemented a *hybrid race detector* which combines lockset-based detection with a limited form of happens-before detection.

4.1 Approach

Our approach is to start with a lockset-based detector as previously described and to add limited happens-before checking. We record thread messages for the Java synchronization primitives `start()`, `join()`, `wait()` and `notify()`. We also provide the ability for the user to mark arbitrary Java methods, *e.g.,*`enqueue` and `dequeue`, as corresponding to thread message sends and receives. However we do not create thread messages corresponding to shared memory write/read or write/write pairs, nor do we create thread messages for lock release/acquire pairs. In practice this means we deal with a very small number of thread messages and the overhead of maintaining the vector clocks is negligible. This limited happens-before relation, denoted by $\dot{\to}$, is only a subset of the true happens-before relation, but we have found that it is nevertheless very useful for weeding out false positives.

4.2 Hybrid Detection Check

The check we perform is simply the conjunction of the lockset detection check and a limited happens-before detection check:

$$\begin{aligned}
&\text{IsPotentialHybridRace}(i,j) = \\
&\quad \wedge\, e_i = \text{MEM}(m_i,a_i,t_i) \wedge e_j = \text{MEM}(m_j,a_j,t_j) \\
&\quad \wedge\, t_i \neq t_j \wedge m_i = m_j \wedge (a_i = \text{WRITE} \vee a_j = \text{WRITE}) \\
&\quad \wedge\, L_i(t_i) \cap L_j(t_j) = \emptyset \wedge \neg(i \dot{\to} j) \wedge \neg(j \dot{\to} i)
\end{aligned}$$

This can be implemented by starting with a regular lockset detector implementation and adding thread message event tracking using vector clocks. We maintain a full vector clock for each active thread and for each sent message that could yet be recieved by another thread. We record for each memory location m a set of (a,t,L,v) tuples corresponding to the accesses e_i to m, where $e_i = \text{MEM}(m,a,t)$, $L = L_i(t)$, and v is the thread's timestamp for itself, $v = V_i(t)(t)$. We can then compute IsPotentialHybridRace between any old event e_i and the current event e_j (using the vector clock for the current thread, $V_j(t_j)$).

Limiting the number of thread messages means that we perform few vector clock updates, which are relatively expensive. We perform many happens-before checks, but they are cheap. Furthermore we do not need to record an entire vector clock for each stored event; a single element suffices to check the happens-before relation. We only need full vector clocks for each active thread and for each pending message.

4.3 Example

Consider figure 1. The lockset detector reports races on both `globalFlag` and `childThread`, but the race on `globalThread` is suppressed because even our limited happens-before detector can prove that `globalFlag = 1` happens-before `if (globalFlag)`. (The ordering is induced by the call to `childThread.start()`).

5. EFFICIENT HYBRID DETECTION

It is impractical to store a record of every access ever performed to a shared memory location, let alone compare each access to every past access to the same location. The hybrid race detector needs optimizations to make it practical. In this section we describe some refinements to the race detection algorithm that dramatically reduce the time and space requirements. These refinements generalize and improve on previous work [9]. We describe the *oversized-lockset* optimization which exploits the fact that locksets are usually all small. We also extend the previously described *lockset-subset* condition to take account of the fact that we are now using happens-before detection as well as lockset detection.

5.1 Redundant Events

Suppose we have recorded a set of past accesses e_i to a particular memory location m. Suppose a new event e_n arrives. Suppose further that we can prove that for every possible event e', if e' races with e_n then e' must also race with one of the events $e_i, 0 < i < n$. Then e_n is *redundant* and it can be ignored without affecting the possibility of detecting a race on location m.

We employ two heuristics to efficiently detect this condition.

5.2 The Lockset-Subset Condition

If the thread, access type, and timestamp of a previous event e_i match those of a new event e_n, and e_i's lockset is a subset of the locks for e_n, then the new event is redundant according to the theorem below.

$$
\begin{aligned}
&\text{IsRedundantLocksetSubset}(i, n) = \\
&\quad e_i = \text{MEM}(m, a_i, t_i) \wedge e_n = \text{MEM}(m, a_n, t_n) \\
&\quad \wedge (a_i = \text{WRITE} \vee a_i = a_n) \wedge t_i = t_n \\
&\quad \wedge L_i(t_i) \subset L_n(t_n) \wedge V_i(t_i)(t_i) = V_n(t_n)(t_n)
\end{aligned}
$$

This condition corresponds to the previously described *weaker-than relation* [9], extended to account for the presence of happens-before detection by checking to make sure that the timestamps for the thread performing the old and new events are the same. As a special case, the condition captures repeated accesses to m by the same thread with the same type, lockset, and timestamp.

Here we exploit the fact that our vector clocks only increment a thread's timestamp after it has sent a message; because we have limited the thread messages, we can obtain many events with the same timestamp. (Some other vector clock formalisms increment the thread's timestamp at every event.)

This check is implemented efficiently using lock-labelled tries to represent the per-memory-location data structures, similar to the tries used in previous work [9].

5.3 Redundancy of Lockset-Subset

THEOREM 2. *Suppose IsRedundantLocksetSubset*(i, n), $i < n$, *and IsPotentialHybridRace*(k, n). *Then IsPotentialHybridRace*(k, i).

The proof is omitted for brevity.

5.4 The Oversized Lockset Condition

Results below show that the lockset-subset heuristic is very effective, but it is not effective enough. For example, we observe many unsynchronized read accesses to "read only" data, *i.e.,* data initialized before the starting of child threads and not modified thereafter. These accesses are performed while holding a variety of locks which happen to be unrelated to the data. The locksets are often disjoint so the lockset-subset heuristic does not work. We end up having to accumulate information about very many accesses, just so that if a thread ever writes to the data, we know to report a race.

Suppose we have observed reads e_1, e_2, e_3 of a datum by a thread with locksets $\{a\}$, $\{b\}$, and $\{c\}$. Another such event e_4 arrives with lockset $\{d\}$. Now suppose that some future event races with e_4 but does not race with e_1, e_2 or e_3. That event's lockset must intersect the locksets of e_1, e_2, e_3 (and not intersect the lockset of e_4). Therefore the future event's lockset must include at least $\{a, b, c\}$. In general, as we collect more non-racing, non-redundant accesses, for a new event to not be redundant the size of the lockset of future racing events must be very large. This is unlikely because in practice we observe that the number of locks held by a thread at any one time is very small.

Therefore suppose we know N, an *a priori* bound on the number of locks a thread can hold at one time: $\forall i, t. |L_i(t)| \leq N$. We have the following *oversized-lockset* redundancy check, this time over a set of event indices instead of a single event index:

$$
\begin{aligned}
&\text{IsRedundantLocksetOversized}(I, n) = \\
&\quad e_n = \text{MEM}(m, a_n, t) \\
&\quad \wedge (\forall i \in I.\ e_i = \text{MEM}(m, a_i, t) \wedge (a_i = \text{WRITE} \vee a_i = a_n)) \\
&\quad \wedge (\forall i \in I.\ V_i(t)(t) = V_n(t)(t)) \\
&\quad \wedge |\text{MinHittingSet}(\{\ L_i(t) - L_n(t) \mid i \in I\ \})| > N
\end{aligned}
$$

Here MinHittingSet(C) computes a smallest set H such that H intersects every element of C. (We assume C does not contain the empty set. An empty set in C would correspond to $L_i(t) - L_n(t) = \emptyset$, *i.e.,* $L_i(t) \subset L_n(t)$, in which case redundancy would have already been detected by IsRedundantLocksetSubset.) The idea is that for nonredundancy, a future event must have a lockset which does not intersect the lockset at e_n but does intersect the locksets of the prior events. MinHittingSet computes a minimum-sized such lockset.

Here we may not be able to prove that e_n is redundant with respect to any single e_i, but we can prove it is redundant with respect to a set of e_is. This is a significant generalization of previous forms of redundancy checking [9, 11].

Unfortunately computing |MinHittingSet| is NP-complete [2]. Therefore we approximate it with a heuristic. Suppose a class C of sets $c_1, \ldots, c_n$ is given. Define

$$
\begin{aligned}
S(0) &= \emptyset \\
S(i) &= S(i-1) \cup \{c_i\}, \forall c \in S(i-1).\ c \cap c_i = \emptyset \\
S(i) &= S(i-1), \text{otherwise}
\end{aligned}
$$

Then $|\text{MinHittingSet}(C)| \geq |S(n)|$. The proof is straightforward:

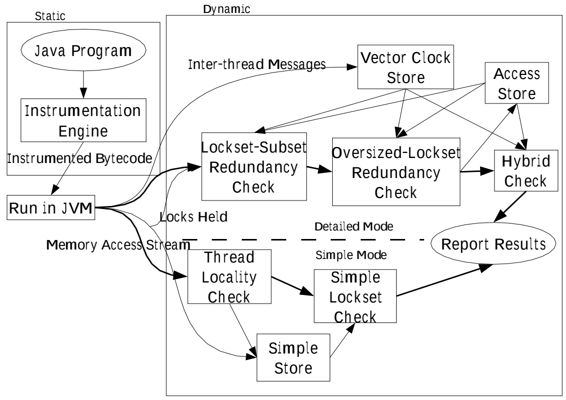

Figure 5: Detector Architecture

the sets in $S(n)$ are mutually disjoint, and therefore any hitting set must include an element from each set in $S(n)$.

Section 6 describes how we use $|S|$ to bound $|\text{MinHittingSet}(C)|$, and, thereby, efficiently compute IsRedundantLocksetOversized(I, n).

Note that for any set L and set of sets C,

$$|\text{MinHittingSet}(C)| \leq |\text{MinHittingSet}(\{\, c - L \mid c \in C\})|$$

Any hitting set for the right hand side is also a hitting set for the left hand side.

5.5 Oversized-lockset Implementation

We determine a value for the maximum lockset size, N, by running the program and observing the size of the largest lockset obtained. (This is done during the "simple mode" run described in section 6.) Of course this is not guaranteed to be correct for future runs, but if N is exceeded in a future run then we detect the violation, increase N, and rerun the program. In pratice this is uncommon, especially if one adds a small "safety margin" to the observed value of N. Alternatively one could use static analysis to find a bound for N (although static analysis may not be able to provide a bound in the presence of recursive synchronized methods).

We check IsRedundantLocksetOversized using the same lock-labelled tries as above. The details are ommitted for brevity.

5.6 Redundancy of Oversized-Lockset

THEOREM 3. *Suppose IsRedundantLocksetOversized*(I, n), $\forall i \in I.\ i < n$, *and IsPotentialHybridRace*(k, n). *Then* $\exists i \in I$. *IsPotentialHybridRace*(k, i).

The proof is omitted for brevity.

6. IMPLEMENTATION

Figure 5 shows the overall architecture of our approach.

6.1 Two-Phase Mode Selection

The overhead of detailed detection as described by IsPotentialHybridRace is generally too high to apply it to every memory location in a program. Therefore we first run the race detector in a low-overhead "simple" mode which is much less accurate but much more efficent than the detailed mode. We identify all Java fields which incur races in this mode. (Our detector does not attempt to detect races on elements of arrays.) Then we rerun the detector in detailed mode, instrumenting accesses to only these "race-prone" fields. The two program runs are not guaranteed to behave identically, but this does not seem to be a problem in practice, perhaps because in both phases we detect potential races rather than races which actually occurred.

Simple mode is not necessary to our approach. We could have the programmer select fields for detailed analysis, or to run the program several times, selecting a different set of fields for detailed analysis each time, until all fields have been checked. However, it is an effective way to make the analysis tractable.

Our detector is the first race detection tool to apply two dynamic phases in this way. The developers of Eraser investigated a similar technique but were unable to use it effectively, because at the binary level it was too difficult to reliably identify the field being accessed by an instruction [6].

6.2 Simple Mode

The simple mode behaves almost identically to the lockset-based detector Eraser [22]. Given a set of accesses $e_i, i \in I$ to location m, we report a race if

$$\begin{aligned}&\text{IsPotentialSimpleRace}(I) = \\ &\quad (\forall i.\exists a, t.\ e_i = (m, a, t)) \wedge (\exists i, t.\ e_i = (m, \text{WRITE}, t)) \\ &\quad \wedge \left|\bigcup_{i \in I}\{\text{Thread}(e_i)\}\right| > 1 \wedge \bigcap_{i \in I} L_i(\text{Thread}(e_i)) = \emptyset\end{aligned}$$

Thus we report a race if we see at least one write, not all accesses are performed by the same thread, and there is no lock that is held by all accesses. This is easy to compute efficiently online as elements are added to I.

6.3 Thread Locality Check

We speed up simple mode even more by applying a *thread locality* check to objects: we only start adding accesses to I once we see that the object has been accessed by more than one thread. Most objects in Java programs are *thread-local*, only ever accessed by a single thread. This can cause us to miss races in simple mode, for example when an object is accessed first by one thread, then by another thread, and never again, and the two accesses race. However, this check is very important for space and time efficiency and most race detectors use it [22, 21, 9]. Results below show that the thread locality check introduces few additional false negatives.

We add a field to each object in which we store the object's *owning thread*, the thread that has performed all accesses to the object so far. The field is initialized to "null" to indicate that the object has not been accessed. We store a special value "shared" if there is no owning thread. At every field access we check to see if the owning thread is the current thread, and if it is, we ignore the event. If the field is "null", we set the owning thread to the current thread, otherwise we continue with simple race detection. No synchronization is performed around the loading and storing of the owner thread field, so it is subject to races and could lead to real races being missed. We did not observe this happening in repeated runs of our experiments.

We need per-thread data, such as the thread's current lockset (and in detailed mode, its vector clock). Therefore instead of storing a reference to the owning thread itself, we store a reference to our per-thread data for the owning thread (which includes a reference to the actual thread object).

6.4 Instrumentation

We insert probes into Java programs by modifying their bytecodes. This allows us to analyze programs for which source is not available. Probes call methods in our race detector, which is also written in Java and runs alongside the user program in the same

```
/*BEFORE*/ class Example {
  Object f;
  synchronized void setF(Object p) {
    this.f = p;
  } }
/*AFTER*/ class Example {
  Object f, _detector_state;
  final static int FIELD_ID_F = 1;
  synchronized void setF(Object p) {
    ThreadInfo current =
      ThreadInfo.getCurrentThreadInfo();
    Detector.acquiredLock(this, current);
    try {
      this.f = p;
      // thread locality check
      if (this._detector_state != current) {
        this._detector_state =
          Detector.performedWrite(this,
            FIELD_ID_F, this._detector_state,
            current);
      }
      Detector.releasedLock(this, current);
    } catch (Throwable t) {
      Detector.releasedLock(this, current);
      throw t;
    } } }
```

Figure 6: Instrumented Code Example

virtual machine.

Figure 6 shows Java code equivalent to the code produced by our instrumentation for simple mode.

Instrumentation of memory accesses and lock acquisitions or releases is inserted after the corresponding bytecode instructions, to ensure that if the instruction throws an exception, then we do not record an event that did not happen. To record the locking behavior of synchronized methods, we insert code at the start of the method to record lock acquisition, and we insert code before every return instruction to record lock release. We also add a "catch-all" exception handler over the whole method to record lock release in that case.

We instrument calls to `Thread.start`, `join`, `notify` and `wait` at each call site, because these are native methods whose code we cannot modify.

We also attempt to automatically identify some "shared channel" data structures. Most such structures sometimes block, and in Java, this is usually implemented by using `notify` and `wait`. When we see a class with methods calling `notify` or `wait`, then we instrument all lock acquisitions and releases in that class to send a thread message when a lock is released, and to receive thread messages when a lock is acquired (if a message is pending for that lock).

In simple mode, our instrumentation engine inlines each thread locality check into the user code performing the access. Other than that, each inserted code snippet simply calls a method in our race detector, passing in all relevant parameters.

The race detection state passed in is an object containing all the per-object state maintained by the race detector. This is normally an array indexed by field identifier since the race detector actually keeps state for each memory location. The instrumentation stores a reference to this state in an added field in the accessed object itself.

No synchronization is performed around the loading and storing of the race detection state. Our instrumentation is carefully engineered so that races in updates to race detection state can only lead to missed races (false negatives). We did not observe this happening over multiple runs of our experiments.

6.5 Handling Class Initializers

Java class initialization performs rather complex synchronization [14]. In particular, while a thread is initializing a class, any other thread attempting to access the class will block until initialization is complete. Thus, while a thread is initializing a class C, accesses it performs to static fields of C cannot race with any other accesses to those fields. Many programs rely on this behavior. We simply do not instrument accesses to static fields of C by C's initializer; this eliminates most false race reports involving class initializers.

6.6 Debugging Support

In large programs it can be difficult to understand the behavior that leads to a potential data race. Our detailed mode assists by reporting, for each of the two accesses deemed to be a potential race, the name of the thread performing the access, the type of the access, the set of locks held, and a full stack trace for the thread.

Collecting and storing a full stack trace is very expensive, especially for events that might not even contribute to races in the future, and therefore previous race detection tools have not reported stack traces for both events in a race. However the information is invaluable, particularly because in Java lock acquisition is lexically scoped, and therefore walking the stack shows where every held lock was acquired. The redundancy optimizations described above are critical to making stack trace collection practical, because we do not need to collect stack traces (or any other information) for redundant events.

Our race detector reports a potential race as soon as the second racing event occurs. Therefore it is easy to interrupt the program and activate a debugger to inspect the full program state at the second event.

6.7 Limitations

Our tool misses some events. It does not capture field accesses performed by native code or by reflection, nor does it capture lock acquisition or release by native code. Furthermore, we do not currently instrument Java library code.

Lost lock acquisitions can lead to false positives. Lost field accesses can lead to false negatives. In practice we have not noticed any problems.

7. EXPERIMENTAL RESULTS

Here we present evidence showing that our tool achieves very precise results with reasonable slowdown. We also show the effects of our different optimizations.

7.1 Benchmarks

We applied the tool to the programs shown in table 1, drawn from a variety of sources. We do not know the exact number of source lines for `tomcat` and `resin`, because they include components for which source is not available. The "Threads" column shows the number of application threads created. The "MaxLocks" column shows the maximum number of locks held at once by an application thread.

We modified `elevator` slightly to force it to terminate when the simulation finishes (normally it just hangs).

The original `specjbb` runs for a fixed length of time and reports the number of transactions processed per second. We modified `specjbb` to process a fixed number of transactions (100,000)

Example	Lines	Threads	MaxLocks	Description
elevator	523	3	1	A real-time discrete event simulator from ETH Zurich [21]
hedc	29948	7	3	A Web-crawler application kernel developed at ETH [21],
tsp	706	3	1	Traveling Salesman Problem solver from ETH [21]
sor2	17742	5	1	Modified Successive Over-Relaxation benchmark from ETH [21]
mtrt	3751	3	2	MultiThreaded Ray Tracer from SPECJVM98 [24]
moldyn	1291	4	1	Java Grande Forum molecular dynamics benchmark
montecarlo	3557	4	1	Java Grande Forum Monte Carlo simulation benchmark
raytracer	1859	4	1	Java Grande Forum raytracer benchmark
specjbb	30078	9	5	SPEC Java Business Benchmark 2000, based on TPC-C [25]
resin	>67536	13	7	Web application server (v2.1.2) from Caucho Technology
tomcat	>54144	21	10	Web application server (v4.0.4) from Apache Foundation

Table 1: Benchmark programs and their characteristics

Example	Unmodified	Simple	Detailed	Detailed-NoHB	Detailed-NoOversized	Detailed-NoOpts
tsp	3.03s	16.87	18.56 (2.50)	13.74 (2.41)	22.39 (2.57)	— (2.59)
sor2	0.62s	5.68	5.15 (1.17)	5.13 (1.11)	5.13 (1.16)	54.66 (1.16)
mtrt	5.56s	4.23	86.38 (1.08)	85.63 (1.07)	84.97 (1.10)	— (1.13)
moldyn	26.42s	2.54	43.94 (1.54)	39.72 (1.50)	44.41 (1.50)	— (1.50)
montecarlo	9.74s	1.40	5.39 (1.04)	12.46 (1.03)	5.32 (1.03)	— (1.03)
raytracer	7.96s	26.89	189.75 (1.04)	163.80 (1.02)	187.11 (1.02)	— (1.05)
specjbb	31.81s	2.94	— (2.21)	— (1.31)	— (—)	— (6.23)
resin	78.46s	1.91	— (2.06)	— (3.74)	— (2.02)	— (9.42)
tomcat	24.08s	2.68	5.37 (1.26)	11.47 (1.30)	5.65 (1.22)	42.65 (1.15)

Table 2: Runtime Performance

and report the time taken. We configured `specjbb` to use 8 warehouses only.

To benchmark `resin` and `tomcat`, we configured them with their respective default Web sites and Web application examples, then used the `wget` tool to crawl the default site and retrieve a list of accessible URLs. Then we modified the Web servers to add a "harness" thread which scans through the list ten times, loading each URL in turn. Each server's persistent caches were cleared at the end of each benchmark run. All the other benchmarks come with their own input data sets; we used the "small" data sets for the Java Grande Forum benchmarks.

7.2 Performance

Table 2 shows the basic runtime overhead of our tool. `elevator` and `hedc` are not CPU-bound and are therefore omitted. We report wall-clock times for the best run out of three consecutive runs, restarting the Java virtual machine for each run. The tests were run on a 2GHz Pentium 4 machine with 1.5GB of memory, using the IBM JDK version 1.3.1, with initial and maximum heap sizes set to 1GB. The JIT compiler was enabled for all tests. Runs taking longer than 1800s were terminated and are represented by dashes.

"Unmodified" reports the running time of the program without any instrumentation, in seconds. The other times are all reported as a ratio of the unmodified time. "Simple" reports the running time with "simple mode" detection enabled. "Detailed" reports the running time with "detailed mode" detection enabled for all fields; the number in parentheses is the time when detection is enabled only for the fields which the Simple run identifies as incurring races. "Detailed-NoHB" reports the running times for detailed mode with happens-before checking disabled. "Detailed-NoOversized" reports the running times for detailed mode with the oversized-lockset optimization disabled. "Detailed-NoOpts" reports the running time with both lockset-subset and oversized-lockset disabled.

Simple mode applied to all fields usually has tolerable overhead. `tsp` and `raytracer` incur high overhead, probably because our instrumentation is causing the JIT to not properly optimize these numeric codes. The overhead of Simple could be reduced considerably using a variety of static optimization techniques [9], such as escape analysis.

Detailed mode applied to all fields usually incurs unacceptable overhead. However, applied only to the fields selected by Simple, the overhead is quite low. The overhead for all-fields detailed mode is, however, comparable to overheads reported for pure happens-before detectors (especially considering our detector records stack traces and other information for easy diagnosis) [10].

The cost of performing the happens-before checks is usually very low. Often, removing the happens-before checks even slows down the program, because many more races must be reported. (Race report files were as large as 25MB for a single run of `tomcat`.)

The oversized-lockset optimization makes no difference in many programs but it is essential for `specjbb`.

The lockset-subset optimization is essential for analyzing all fields and even for analyzing selected fields in `specjbb` and `resin`.

7.3 Accuracy

Table 3 records the number of distinct fields on which we detect races in a variety of scenarios.

"Fields" gives the total number of fields present in the benchmark code. (The number of fields for `tomcat` and `hedc` is misleading because much of code of these applications is not exercised by our benchmark harness.) "Simple" reports the number of racing fields found in simple mode. "Detailed-All" reports the number of racing fields found in detailed mode applied to all fields. "Detailed" reports the number of racing fields found in detailed mode applied only to the fields selected by "Simple".

Example	Fields	Simple	Detailed -All	Detailed	Detailed -NoHB
`elevator`	19	0	0	0	0
`hedc`	546	5	4–0–0	4–0–0	3–0–1
`tsp`	29	3	1–0–2	1–0–2	0–0–2
`sor2`	220	0	0	0	0
`mtrt`	413	13	2–3–2	2–3–0	2–2–7
`moldyn`	254	0	0	0	0
`montecarlo`	254	1	0–1–0	0–1–0	0–1–0
`raytracer`	254	1	1–0–0	1–0–0	0
`specjbb`	703	5	—	0–1–1	0–1–1
`resin`	9289	193	—	1–18–57	1–25–152
`tomcat`	8908	22	5–7–39	3–6–13	2–6–17

Table 3: Number of Fields With Dataraces Reported (classified as Bugs–Benign–False)

"Detailed-NoHB" is as for "Detailed", except that happens-before checking is turned off and a thread-locality check is used, as in previous work [9, 21]. In Figure1, the thread locality check would prevent the detector from recording the write `globalFlag = 1;` because at this point `globalFlag` is local to the main thread. The thread locality check prevents reporting false races for many common cases where a memory location is initialized by one thread and then read by another newly started thread without locking. Thus "Detailed-NoHB" represents the results obtained by a state-of-the-art purely lockset-based detector.

Each field count is broken down into three categories:

- *Bugs*: the races reported on the field lead to observable violations of correctness.
- *Benign*: the races reported on the field can occur in practice but cannot affect correctness, either because the code is designed to be correct in the presence of such races, or the values affected by the races do not lead to observable violations of correctness.
- *False*: the races reported on the field cannot occur in practice because the program has implicit synchronization not observed by the race detector.

These classifications were performed by inspection of the code and are subjective in some cases, given that complete correctness specifications are not available for these programs.

When multiple races are reported on a field (belonging to different objects), we classify the field with the most serious classification of the races (*i.e.,* the field is a bug if any reported race on it is a bug, etc).

The "Detailed-All" and "Detailed" results show that our detector reports few false races for most programs, and even for the large programs the number of false races does not overwhelm the true races. Comparing the two columns shows that using "Simple" mode to select fields for detailed analysis is effective: not only does it improve performance, but it also improves the ratio of real races to false races reported.

The "NoHB" results show that happens-before checking reduces the false positive rate and increases the number of bugs reported. This is partly because using the thread locality check in a lockset-based detector can mask bugs because accesses performed while an object is thread-local will be completely ignored. Also, sometimes the first race reported for a particular field by the lockset-based detector is a false race, but the hybrid detector eliminates that false positive and goes on to report a true race for the same field.

7.4 Bugs Found

We found bugs in many programs, even well-tested, much-used applications such as `tomcat`. The bugs are diverse and it seems hard to identify the "cause" of each bug, but in most cases it seems the programmer simply did not recognize the possibility of concurrent access (rather than recognizing the possibility of concurrency but failing to deal with it correctly.)

The severity of the bugs found ranges from potential aborts (*e.g.,* `hedc`) and hangs (*e.g.,* `resin`) to data corruption (*e.g.,* `tomcat`, `mtrt`), wrong answers (*e.g.,* `tsp`), and minor violations of resource constraints (*e.g.,* `tomcat`).

7.5 Benign Races

Our benchmarks contain many pieces of code which incur data races without necessarily compromising correctness.

- *Constant writes*: Some programs write to a shared memory location multiple times without synchronization, where the value written is constant. Such writes do not affect the behavior of the program. Future detectors should ignore write events when the value written is the same as the current value.
- *Caching*: Caches occur frequently in programs. Some caches have weak semantics: adding two copies of the same object to a cache, or failing to find an object in a cache, need not be treated as incorrect. Some programmers exploit this and improve performance by carefully removing some synchronization in cache operations.
- *Lazy initialization*: Some programs test to see if a memory location has been initialized, and if it has not, initialize the location with some predictable value. This pattern can be implemented without synchronization.
- *Asynchronous notification*: Many programs update a shared memory location to asynchronously signal other threads, which periodically poll the location.
- *Timestamps*: Some programs maintain global counters or timestamps which are accessed without synchronization. Threads which read the timestamps rely on observing a consistently non-decreasing value and observing updates at some minimum rate.
- *Statistics*: Many programs keep internal statistics such as pool sizes, bytes transferred, estimated queue delay, etc, to aid performance decisions. Often, incorrect values for these statistics will not affect correctness, and small deviations from the true value have no noticeable effect. Such statistics are often read and updated without synchronization.
- *Unused*: Some values that are incorrectly updated are simply never used by the program.
- *Double-checked locking*: This pattern is especially popular in `tomcat`: a boolean test is copied out of a synchronized block to try to avoid unnecessary synchronization. For example:

```
if (e) { synchronized (...) { if (e) ...; } }
```

Many of these patterns are actually *not* necessarily safe according to the Java memory model. In principle, most of the double-checked locking, lazy initialization, constant writes and asynchronous notification in these benchmarks could cause incorrect behavior in the presence of aggressive multiprocessor memory systems or aggressive optimizing compilers [3], and are arguably bugs. We have classified these races as benign because they are safe in typical Java environments.

7.6 False Races

Certain kinds of race-free code are not yet identified as such by our detector.

- *Shared channels*: Some programs use shared data structures as channels to pass data between threads. Access to the channels is synchronized but access to the transmitted data need not be. We can handle these programs by manually identifying channel structures and associating inter-thread messages with operations on those structures. These are very common in `tomcat` and `resin`.

- *Data ordering*: Some programs use data flags to signal when a shared location may be safely read from or written to. Accesses to the flags are synchronized but accesses to the shared location need not be. We can handle these cases by associating inter-thread messages with accesses to the data flags.

- *Finalizers*: When an object is no longer reachable from heap roots, the Java virtual machine invokes its `finalize` method in a special "finalizer" thread. Some finalizer code guarantees race-freedom by relying on the fact that there are no outstanding references to the finalized object. In general the race detector would have to perform some analysis of heap connectivity to capture such constraints.

- *Library and native code*: Synchronization performed by library and native code is not observed by our race detector. This contributes to a few false races where such synchronization prevents races from occurring in the user's program.

The vast majority of the false races reported are caused by shared channels. In particular `resin` and `tomcat` frequently "recycle" objects by putting unused objects into free lists; when a new object is needed it is removed from an appropriate free list, if available, rather than being freshly allocated. These free lists act as channels passing objects from one thread to another. Manually adding a few annotations to indicate that the use of object pools should induce happens-before edges eliminates almost all the false races. Automatic, low-overhead detection of shared channels and data ordering remains an open problem.

8. RELATED WORK

Past research on race detection can be classified as *ahead-of-time*, *on-the-fly*, or *post-mortem*. These approaches offer different trade-offs along *ease-of-use*, *precision*, *efficiency*, and *coverage* dimensions.

Ahead-of-time race detection is usually performed in *static data race analysis* tools which yield high coverage by considering the space of all possible program executions and identifying data races that might occur in any one of them. Flanagan and Freund's race detection tool is a static tool for Java [13] which tracks synchronization using extended type inference and checking. Guava is a dialect of Java that statically disallows races by preventing concurrent accesses to shared data [4]. Only instances of classes belonging to the *class category* called *monitor* can be shared by multiple threads. By serializing all accesses to fields or methods of the same shared data, Guava can prevent races. Boyapati and Rinard propose a system of type annotations for Java that ensures a well-typed program is race-free and allows the programmer to write a generic class and subclass it with different protection mechanisms [5].

Warlock is an annotation-based static race detection tool for ANSI C programs [23], which also supports lock-based synchronization. Aiken and Gay's work statically detects data races in SPMD programs [1]. Since SPMD programs employ barrier-style synchronizations, they do not track locks held at each statement.

The key advantage of dynamic analysis approaches such as on-the-fly and post-mortem race detection is the precision of the results, but this often comes with a high efficiency cost. A dynamic approach also has more limited coverage than a static approach because it only examines a single dynamic execution. (However, most dynamic race detectors improve coverage by considering alternate orderings of synchronization operations that are consistent with the actual events observed in the original program execution.)

Eraser [22] is a well-known lockset-based race detector. Eraser enforces the constraint that each shared memory location is protected by a unique lock throughout an execution. Our lockset check is more flexible than Eraser's and, in conjunction with our happens-before checking, our approach is more accurate than Eraser's. Our thread locality check can be traced back to Eraser's ownership model. Eraser works independently of the input source language by instrumenting binary code, but its runtime overhead is in the range of $10\times$ to $30\times$.

Praun and Gross's *object race detection* [21] for Java improved on Eraser's performance by applying escape analysis to filter out non-racing statements and by detecting races at the object level instead of at the level of each memory location (their overhead ranges from 16% to 129% on their three benchmarks). However, their coarser granularity of race detection leads to the reporting of many false races.

TRaDe [10] is a state-of-the-art pure happens-before detector for Java. TRaDe adds a runtime overhead ranging from $4\times$ to $15\times$ [10] compared to an interpreter. Since JIT compilers usually speed up programs by an order of magnitude, TRaDe's overhead for a compiled program would be considerable. Unfortunately many of the techniques we use to improve performance, particularly the use of Simple mode to select fields for detailed analysis, would not help much with TRaDe because it would still need to track an enormous number of inter-thread messages.

Dinning and Schonberg introduced the idea of detecting races based both on locksets and the happens-before relation [11]. They also describe an optimization similar to our lockset-subset optimization. They do not appear to have implemented their algorithm.

Cheng et al. [8] combined lockset-based detection with ordering information derived from the "dag" execution model of the Cilk parallel programming language. Their ordering information can be thought of as a limited happens-before relation. They only present results for small Cilk programs and their algorithm does not handle the less structured parallelism of Java programs.

Our previous work was based on locksets, but also employed the happens-before relation in a limited fashion by using locks and a thread locality check to simulate synchronization performed by `start()` and `join()` [9]. This work improves on that work by eliminating the need for whole-program static analysis and obtaining efficiency by adding "Simple" mode and performing two passes. We also improve accuracy by adding first-class happens-before checking, and improve usability by reporting stack traces for both events in a race. However, our new tool could still benefit from some of the static optimization techniques used in the previous system.

9. CONCLUSIONS AND FUTURE WORK

We have shown that by combining the two techniques of lockset-based and happens-before-based detection, and by using lockset-based detection alone to select fields for analysis, we obtain a detector with better accuracy than previous lockset-based detectors and with significantly better performance than previously studied happens-before detectors. We have also formalized race detection and proved the safety both of previously known optimizations and new optimizations in the combined happens-before and lockset framework. Our results also represent a major improvement in scalability over previously reported efficient race detectors [9, 21]; `tomcat` and `resin` are an order of magnitude larger than the benchmarks used with those detectors.

The analysis of our results also reveals interesting properties of our benchmark programs. Many of the races found in real programs are benign, probably even intentional, and do not affect correctness — if one assumes a somewhat stricter memory model than Java actually promises.

In this work we have not taken advantage of any static analysis or optimizations to reduce overhead (a strategy which contributed to our scalability improvement). A clear direction for future work is to apply previously known or new scalable static analyses to reduce amount of instrumentation required, especially in the first "Simple mode" pass.

Perhaps the most useful feature of the happens-before checking we have introduced is to allow easy modelling of arbitrary synchronization constraints between threads. In the future we need to exploit this by automatically identifying and modelling the use of shared channels. It would also be helpful to automatically recognize benign data races.

Acknowledgements

We would like to thank Dan Pelleg and the CMU zephyr community for identifying the minimum hitting set problem, Darko Marinov for helpful comments on the paper, and Mark Christiaens and Christof von Praun for helping us find benchmark programs and for sharing their results and insights with us.

10. REFERENCES

[1] A. Aiken and D. Gay. Barrier inference. In *Proceedings of the 25th Symposium on Principles of Programming Languages (POPL)*, pages 342–354, January 1998.

[2] G. Ausiello, A. D'Atri, and M. Protasi. Structure preserving reductions among convex optimization problems. *Journal of Computer System Sciences*, 21(1):136–153, Aug. 1980.

[3] D. Bacon, J. Bloch, J. Bogda, C. Click, P. Haahr, D. Lea, T. May, J.-W. Maessen, J. D. Mitchell, K. Nilsen, B. Pugh, and E. G. Sirer. The "double-checked locking is broken" declaration. http://www.cs.umd.edu/ pugh/java/memoryModel/ DoubleCheckedLocking.html.

[4] D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: A dialect of java without data races. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2000.

[5] C. Boyapati and M. Rinard. A parameterized type system for race-free java programs. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2001.

[6] M. Burrows. Personal communication.

[7] B. Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39(1), July 1991.

[8] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting data races in Cilk programs that use locks. *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, 1998.

[9] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise data race detection for object oriented programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 285–297, June 2002.

[10] M. Christiaens and K. De Bosschere. TRaDe, a topological approach to on-the-fly race detection in java programs. *Proceedings of the Java Virtual Machine Rsearch and Technology Symposium (JVM'01)*, April 2001.

[11] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 26(12):85–96, 1991.

[12] C. J. Fidge. Timestamp in message passing systems that preserves partial ordering. In *Proceedings of the 11th Australian Computing Conference*, pages 56–66, 1988.

[13] C. Flanagan and S. N. Freund. Type-based race detection for java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 219–232, June 2000.

[14] J. Gosling, B. Joy, and G. Steele. *The Java(TM) Language Specification*. Addison-Wesley, 1996.

[15] C. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science. Prentice Hall, 1985.

[16] KL Group, 260 King Street East, Toronto, Ontario, Canada. Getting Started with JProbe.

[17] Kuck & Associates, Inc., 1906 Fox Drive, Champaign, IL 61820-7345, USA. AssureJ User's Manual, 2.0 Edition, March 1999.

[18] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[19] F. Mattern. Virtual time and global states of distributed systems. In *Proceedings of the Parallel and Distributed Algorithms Conference*, pages 215–226. Elsevier Science, 1988.

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

[21] C. v. Praun and T. Gross. Object race detection. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2001.

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

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

[24] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98/, 1998.

[25] The Standard Performance Evaluation Corporation. SPEC JBB 2000. http://www.spec.org/osg/jbb2000/, 2000.