BIGFOOT: Static Check Placement for Dynamic Race Detection

Dustin Rhodes    Cormac Flanagan    Stephen N. Freund
University of California, Santa Cruz, USA    Williams College, USA

Abstract
Precise dynamic data race detectors provide strong correctness guarantees but have high overheads because they generally keep analysis state in a separate shadow location for each heap memory location, and they check (and potentially update) the corresponding shadow location on each heap access. The BIGFOOT dynamic data race detector uses a combination of static and dynamic analysis techniques to coalesce checks and compress shadow locations. With BIGFOOT, multiple accesses to an object or array often induce a single coalesced check that manipulates a single compressed shadow location, resulting in a performance improvement over FASTTRACK of 61%.

CCS Concepts • Theory of computation → Program analysis: • Software and its engineering → Concurrent programming languages; Software defect analysis

Keywords Data race conditions, concurrency, static analysis, dynamic analysis

1. Introduction
Data race conditions are a notorious problem in multithreaded software, often resulting in erroneous outputs and violations of expected correctness properties such as sequential consistency and atomicity. Much prior work has focused on static [1, 2, 4, 10, 18, 21, 31, 27, 53] and dynamic [15, 38, 40, 44, 51, 59, 59] data race detection.

Static analyses are able to reason about all executions of a program, but they generate false alarms or miss actual data races due to their necessarily conservative approximations of program behavior. In contrast, precise dynamic analyses offer a stronger guarantee of reporting a race condition if and only if a race occurs in the observed trace. The main limitation of precise dynamic detection is performance. The most efficient precise detectors, such as DJIT+ [40] and FASTTRACK [23], have overheads close to an order of magnitude or more, which is too high for many applications.

In general, precise dynamic race detectors work by keeping, for each shared memory location in the target program, a corresponding shadow location that records information about the access history for that memory location. For example, in DJIT+ each shadow location records the time of the last read and write to that location by each thread [40]. FASTTRACK refines this representation to store only the most recent read and write among all threads when possible [23].

The primary sources of overhead in dynamic race detectors are: the space overhead of maintaining a shadow location for each memory location in the target, and the time overhead of updating shadow locations for each memory access of the target. Dynamic analyses may sacrifice precision for reduced overhead, but only at the cost of introducing undesirable false alarms or missed races. In this paper, we present an optimized precise dynamic data race detection algorithm, BIGFOOT, that mitigates these overheads as follows:

1. Rather than keeping a distinct shadow location for each field in an object, or each entry in an array, BIGFOOT employs compressed representations using fewer shadow locations per object/array.

2. Rather than checking and updating shadow location metadata at each memory access of the target program, BIGFOOT uses a sophisticated static analysis to optimize check placement in the target code. In particular, it statically eliminates redundant checks where possible and statically combines multiple checks into a single coalesced check covering multiple fields or array indices.

Figure 1 compares BIGFOOT’s static check placement algorithm to the standard approach of performing a check at each access. In the move method, a typical race detector would instrument each of the six accesses with a check verifying that the access is race free. In contrast, BIGFOOT determines that the read check in each read-modify-write sequence is redundant with the check on the subsequent write, in the sense that the read will be involved in a data-race only if the write is also in a race. Thus, the read checks are not necessary to validate whether a trace is race free.

Furthermore, BIGFOOT combines the three write checks into a single coalesced check CheckWrite(this.x/y/z)
covering all three fields. Coalescing field checks in this manner is particularly helpful because it enables static shadow location compression for objects. In particular, suppose that all checks on Point objects are coalesced checks of the form CheckWrite(p.x/y/z) or CheckRead(p.x/y/z). BigFoot can then safely combine the shadow locations for the three fields into a single shadow location, and the coalesced checks then perform a single check-and-update operation on that shadow location, in contrast to the six checks on three shadow locations required by the traditional approach.

BigFoot optimizes array checks similarly, as shown in the method movePts in Figure 1. That code iterates over all array indices in a from lo to hi and moves each corresponding Point. In contrast to a standard dynamic race detector, which separately checks each array read, BigFoot coalesces these checks into the single check CheckRead(a[lo..hi]) after the loop. Here, lo..hi denotes the closed-open interval lo,lo + 1,...,hi − 2,hi − 1.

To efficiently handle such coalesced checks, BigFoot again employs a compressed representation for array shadow locations. In contrast to objects however, this compressed representation is chosen and adaptively refined at run time. Specifically, an array like a is initially represented as a "coarse-grained" single shadow location covering all array elements. A call such as movePts(a,0,a.length) generates a coalesced check CheckRead(a[0..a.length]) covering all array elements, which is processed at run time by checking and updating that array’s single shadow location. If a subsequent call movePts(a, 0, a.length/2) generates a check CheckRead(a[0..a.length/2]) covering just half the array elements, the BigFoot run time would refine the shadow state for a to be two shadow locations, each covering half of a. That check is then handled by appropriately updating the first of these two shadow locations.

BigFoot’s adaptive mechanism for arrays, modeled after SlimState [55], enables compressed array representations under a variety of common access patterns including block-based and strided accesses. If those patterns are not followed, BigFoot reverts to the “fine-grained” representation of a shadow location for each array element.

Imprecisions in BigFoot’s static analysis may lead to sub-optimal check placement, as in the following example:

```java
for(int i = 0; i < a.length; i++) {
    if(predicate()) {
        a[i].move(1, 1, 1);
        CheckRead(a[i]);
    }
}
```

BigFoot’s method-local analysis will not statically coalesce the array checks because it cannot statically determine which elements are accessed. At run time, suppose a has a single shadow location when this code runs. If predicate() always returns true, then all indices in a are accessed, and we’d like to preserve the coarse-grained representation to save both space and time. To do so, BigFoot’s run time defers checks on arrays, and instead dynamically records a per-thread footprint of which indices have “pending” checks. BigFoot

---

**Figure 1.** Check placement for precise data race detection.

<table>
<thead>
<tr>
<th>Standard Race Checks</th>
<th>BigFoot Race Checks</th>
</tr>
</thead>
<tbody>
<tr>
<td>class Point {</td>
<td>class Point {</td>
</tr>
<tr>
<td>int x, y, z;</td>
<td>int x, y, z;</td>
</tr>
<tr>
<td>void move(int dx, int dy, int dz) {</td>
<td>void move(int dx, int dy, int dz) {</td>
</tr>
<tr>
<td>int tmp;</td>
<td>int tmp;</td>
</tr>
<tr>
<td>CheckRead(this.x); tmp = this.x;</td>
<td>tmp = this.x;</td>
</tr>
<tr>
<td>CheckWrite(this.x); this.x = tmp + dx;</td>
<td>this.x = tmp + dx;</td>
</tr>
<tr>
<td>CheckRead(this.y); tmp = this.y;</td>
<td>tmp = this.y;</td>
</tr>
<tr>
<td>CheckWrite(this.y); this.y = tmp + dy;</td>
<td>this.y = tmp + dy;</td>
</tr>
<tr>
<td>CheckRead(this.z); tmp = this.z;</td>
<td>tmp = this.z;</td>
</tr>
<tr>
<td>CheckWrite(this.z); this.z = tmp + dz;</td>
<td>this.z = tmp + dz;</td>
</tr>
<tr>
<td>}</td>
<td>CheckWrite(this.x/y/z);</td>
</tr>
<tr>
<td>void movePts(Point[] a, int lo, int hi) {</td>
<td>}</td>
</tr>
<tr>
<td>for(int i = lo; i &lt; hi; i++) {</td>
<td>void movePts(Point[] a, int lo, int hi) {</td>
</tr>
<tr>
<td>CheckRead(a[i]);</td>
<td>for(int i = lo; i &lt; hi; i++) {</td>
</tr>
<tr>
<td>a[i].move(1, 1, 1);</td>
<td>a[i].move(1, 1, 1);</td>
</tr>
<tr>
<td>}</td>
<td>CheckRead(a[lo..hi]);</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>
“commits” the footprint for a thread and checks the corresponding shadow locations for races when the thread next performs a synchronization operation. This dynamic footprinting technique allows BigFoot to keep a single shadow location for the array a, even in the presence of a scenario like the above that is not amenable to static coalescing.

Figure 2 compares BigFoot to several prior precise race detection algorithms: FastTrack, RedCard (which statically eliminates some redundant checks and compresses shadow state), SlimState (which dynamically compresses array shadow state), and SlimCard (which combines the RedCard and SlimState analyses, as described in Section 5). All were implemented in the RoadRunner framework for Java [24]. The key innovations of BigFoot, namely static check motion and coalescing, provide substantial performance improvements, particularly when combined with existing static and dynamic shadow compression techniques.

Detection Precision A data race detector is trace-precise if it correctly determines whether a given trace has a race condition or not. A trace-precise race detector is additionally address-precise if it can also determine all addresses that have race conditions. Using this terminology, FastTrack and SlimState are address-precise. Our BigFoot core algorithm is also address-precise, as we discuss in Section 3. Our BigFoot implementation, however, uses additional check placement optimizations for which one data race may prevent the detection of a subsequent race. Consequently, our implementation is trace-precise but not address-precise, as described in Section 5. In practice, the BigFoot implementation was address-precise in all our experiments.

We also note that since BigFoot defers checking until after accesses occur, a data race may be detected only after it has happened. This introduces several subtleties related to precision. First, we currently assume for simplicity that all loops terminate and consider all unchecked exceptions to be programming errors. Thus, a race preventing a loop from terminating or causing an unchecked exception may be missed since the deferred check is never reached. However, we did not see this occur in practice, and we discuss analysis extensions to cover these items in Sections 3 and 5. In addition, if a data race can corrupt the race detector’s analysis state, it may similarly go undetected [5], but for type-safe languages like Java, this cannot happen.

Contributions The primary contributions of this paper are:

- We define a theory of precise check placement for dynamic race detection and describe a core static analysis to optimize check placement (Sections 2 and 3).
- We integrate static field proxy compression and dynamic array shadow compression techniques to further reduce run-time overhead (Section 4).
- We present our BigFoot prototype for Java (Section 5).
- We show that BigFoot’s static analysis scales well (requiring on average less than 0.2s per method processed) and reduces run-time overhead from 7.3x (for FastTrack) to 2.5x, an improvement of 61% (Section 6).

2. Theory of Check Placement

A key design goal of the core BigFoot algorithm is that the checks inserted into a target program enable address-precise data race detection. That is, BigFoot must insert checks that are sufficient to detect all data races but that never report false alarms. Reasoning about this requirement can be subtle. For example, the code in Figure 3 contains a single check that enables precise data race detection for all three accesses, but it may not be immediately apparent why this is the case.

---

\[ 1: \text{acq(lock);} \]
\[ 2: x = b.f; \]
\[ 3: \text{rel(lock);} \]
\[ 4: y = b.f; \]
\[ 5: \text{check(b.f);} \]
\[ 6: \text{acq(lock);} \]
\[ 7: z = b.f; \]
\[ 8: \text{rel(lock);} \]

Figure 3. A code fragment with precise checks, and the corresponding BigFoot analysis contexts from Section 3 (All variables are thread-local, and objects thread-shared.)

---

<table>
<thead>
<tr>
<th>Race Detector</th>
<th>Check Motion and Coalescing</th>
<th>Red. Check Elimination</th>
<th>Metadata Compression</th>
<th>Run-Time Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>FastTrack</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>7.3x</td>
</tr>
<tr>
<td>RedCard</td>
<td>no</td>
<td>no</td>
<td>static proxy</td>
<td>6.0x</td>
</tr>
<tr>
<td>SlimState</td>
<td>no</td>
<td>dynamic</td>
<td>no</td>
<td>6.0x</td>
</tr>
<tr>
<td>SlimCard</td>
<td>no</td>
<td>dynamic</td>
<td>static</td>
<td>5.1x</td>
</tr>
<tr>
<td>BigFoot</td>
<td>static</td>
<td>static&lt;dynamic&gt;</td>
<td>static proxy</td>
<td>2.5x</td>
</tr>
</tbody>
</table>

Figure 2. Comparison to prior precise dynamic race detectors, and SlimCard (which combines RedCard and SlimState).
In this section, we develop a theory of check placement to characterize exactly where checks must be performed to avoid false negatives and false positives. To simplify our exposition, we initially do not distinguish between read and write accesses (although our implementation extends these ideas to do so, as described in Section 5).

Given an execution trace of a program, we say the trace has a data race if it has two accesses to the same memory location that are not ordered by the happens-before relation, which is defined in the usual fashion [33][40].

Similarly, a trace has a check race if it has two checks to the same memory location that are not ordered by happens-before. A precise check placement algorithm must ensure that any execution trace of the target program has a data race if and only if it has a check race.

Figure 4(a) illustrates where checks must be performed in a trace to guarantee all data races. The trace shown has a data race because the happens-before edges (shown as solid arrows) generated by synchronization operations do not order the two accesses to y.f, as indicated by the dashed edge. Any check performed by Thread 1 in the Covering Check Range will trigger a check race corresponding to that data race. However, checks outside that range will not, resulting in a false negative because the access race would have no corresponding check race.

With this intuition, we say that a check covers an access to the same location by the same thread if the check either:

- precedes the access with no intervening release, or
- succeeds the access with no intervening acquire.

Note that we treat acquire and released differently, as they serve as sources and sinks for synchronization edges in the happens-before graph, respectively. Returning to Figure 4(a), the single check thus covers all three accesses in any trace generated by this code. We show in the supplementary appendix that if each access in a program has a covering check, then any trace with a data race also has a check race. That is, access coverage guarantees no missed races.

Figure 4(b) illustrates where checks may be performed in a trace to guarantee all check races indicate data races. This trace has no data race because the three accesses to y.f are ordered by happens-before edges. Similarly, the checks inside the critical section of Thread 1 (marked Legitimate Check Range) produce no check races. However, a check outside this range produces a check race, which would be a false alarm since there is no corresponding data race.

We say that a check is legitimate for an access to the same location by the same thread if the check either:

- precedes the access with no intervening acquire, or
- succeeds the access with no intervening release.

For example, in Figure 3 the check is legitimate for the second access, but not the first or third.

With these notions of legitimacy and coverage, we say a trace has precise checks if each access is covered by some check (no missed races) and each check is legitimate for some access (no false alarms). A program has precise checks if all possible execution traces have precise checks.

3. Optimizing Check Placement

We next describe our static analysis for optimizing the placement of precise checks.

3.1 BFJ Language and Semantics

We formalize our ideas in terms of the idealized language BFJ (BigFoot Java) shown in Figure 5. A program P contains a sequence of class definitions D and a collection of concurrent threads s1||...||sn. Each class definition D contains field and method declarations. Each field declaration is simply a field name f. Each method declaration m(x){s; return z} includes a unique method m, formal parameters x, and a body s followed by a return of the local variable z. We omit static
The BFG algorithm defers checks as long as possible and only inserts them into the program code when they cannot be further deferred without risking false alarms or missed data races; thus checks are only placed before synchronization operations and control flow merge points, and at the ends of methods and threads.

To illustrate how BFG uses context information to place checks, we examine the analysis contexts in Figure 5. As in all BFJ code, the variables in this snippet are local and cannot be changed by other threads, although they may point to shared objects.

BFG adds a past access $p^\triangledown$ to the history whenever the code accesses $p$, and before an acquire it inserts a check for any past access $p^\triangledown$ with no past covering check $p^\triangledown$, as at line 5. Since the acquire signifies the end of that past access’s covering check range, placing the check any later would introduce the potential for missed data races.

At each release, BFG removes each past access $p^\triangledown$ from the history. The release signifies the end of the legitimate check range for those accesses, and placing checks for them any later would introduce the potential for false alarms. “Forgetting” a past access $p^\triangledown$ like this typically requires BFG to place a covering check before the release, but there are two situations when no check is needed: (1) a covering check has already occurred ($p^\triangledown$ is in the history), as at line 8; or (2) we anticipate a later access to the same location, as at line 3. The anticipated later access (and hence its covering check) will occur before leaving the original access’s covering check range at the next acquire. Each check $p^\triangledown$ must also be forgotten at a release because that check does not cover any subsequent access to $p$.

Anticipated access information flows backwards, and anticipated accesses in an acquire’s post-history must be removed from its pre-history because checks covering those future accesses will not cover accesses prior to the acquire.

We now examine the if statement in Figure 5(a). The merged context $H \bullet \{ b.f^\triangledown \}$ after the if describes properties holding after both branches, and it omits past accesses occurring only on one branch. BFG must ensure a covering check exists for any such “forgotten” past access. That necessitates checking $b.g$ in the “then” branch, after which it is permissible to simultaneously forget both the past access and past check on $b.g$ when leaving the if. In contrast, $x.f$ is anticipated at the end of the “else” branch, and we skip checking it at that point because the later access will have a check covering both accesses.

### 3.3 Check Placement Algorithm Overview

The BFG algorithm defers checks as long as possible and only inserts them into the program code when they cannot be further deferred without risking false alarms or missed data races; thus checks are only placed before synchronization operations and control flow merge points, and at the ends of methods and threads.

To illustrate how BFG uses context information to place checks, we examine the analysis contexts in Figure 5. As in all BFJ code, the variables in this snippet are local and cannot be changed by other threads, although they may point to shared objects.

BFG adds a past access $p^\triangledown$ to the history whenever the code accesses $p$, and before an acquire it inserts a check for any past access $p^\triangledown$ with no past covering check $p^\triangledown$, as at line 5. Since the acquire signifies the end of that past access’s covering check range, placing the check any later would introduce the potential for missed data races.

At each release, BFG removes each past access $p^\triangledown$ from the history. The release signifies the end of the legitimate check range for those accesses, and placing checks for them any later would introduce the potential for false alarms. “Forgetting” a past access $p^\triangledown$ like this typically requires BFG to place a covering check before the release, but there are two situations when no check is needed: (1) a covering check has already occurred ($p^\triangledown$ is in the history), as at line 8; or (2) we anticipate a later access to the same location, as at line 3. The anticipated later access (and hence its covering check) will occur before leaving the original access’s covering check range at the next acquire. Each check $p^\triangledown$ must also be forgotten at a release because that check does not cover any subsequent access to $p$.

Anticipated access information flows backwards, and anticipated accesses in an acquire’s post-history must be removed from its pre-history because checks covering those future accesses will not cover accesses prior to the acquire.

We now examine the if statement in Figure 5(a). The merged context $H \bullet \{ b.f^\triangledown \}$ after the if describes properties holding after both branches, and it omits past accesses occurring only on one branch. BFG must ensure a covering check exists for any such “forgotten” past access. That necessitates checking $b.g$ in the “then” branch, after which it is permissible to simultaneously forget both the past access and past check on $b.g$ when leaving the if. In contrast, $x.f$ is anticipated at the end of the “else” branch, and we skip checking it at that point because the later access will have a check covering both accesses.
Figure 6. Analysis contexts and check placements for BFJ method bodies containing (a) an if statement and (b) a loop.

Checks deferred until after a loop may never be executed if the loop diverges. We currently assume all loops terminate but could alternatively include a termination analysis and treat potentially non-terminating loops specially by, for example, periodically committing deferred checks inside the loop.

3.4 Check Placement Rules

We formalize BigFOOT’s check placement algorithm as the judgment \( \vdash s : H \cdot A \rightarrow H' \cdot A' \) defined in Figure 7. The contexts \( H \cdot A \) and \( H' \cdot A' \) are the pre- and post-contexts of \( s \). The analysis is a combined forward/backward analysis; history properties flow forward from pre-history \( H \) to post-history \( H' \), while anticipated properties flow backwards from post-anticipated \( A' \) to pre-anticipated \( A \).

For conciseness, we do not express check placement as a rewriting transformation on program syntax. Instead, we assume that a pre- transformation has already inserted a check check(C) wherever one may be required. The goal of the check placement algorithm is then to resolve each path set variable \( C \) into the appropriate set of paths to be checked at that point. The rules for \( \vdash s : H \cdot A \rightarrow H' \cdot A' \) include antecedents constraining each \( C \) appropriately.

Context Entailment and Ordering Our rules use the notation \( h \in H \) for the usual syntactic notion of set membership for history properties. In addition, we introduce a richer notion of history entailment (\( H \vdash h \)) that accounts for other information in \( H \). For example, if \( H = \{z[i]^\circ, i = j\} \) then we can safely infer that \( H \) entails \( z[j]^\circ \), written \( H \vdash z[j]^\circ \). Similarly, we introduce anticipated entailment (\( H \cdot A \vdash a \)), as in \( \{i < 10\} \cdot \{x[0..10]^\circ\} \vdash x[0..10]^\circ \). Our implementation uses Z3 [10] to reason about entailment.

While history and anticipated sets could be ordered by the subset relation (\( \subseteq \)), we employ a stronger ordering (\( \sqsubseteq \)) based on entailment to achieve greater precision:

\[
H_1 \sqsubseteq H_2 \iff \forall h \in H_1. H_2 \vdash h
\]

\[
H \vdash A_1 \sqsubseteq A_2 \iff \forall a \in A_1. H \cdot A_1 \vdash a
\]
We assume $x \notin \text{Vars}(H)$ in the rules modifying $x$: [Assign], [Rename], [New], [Read], [A-New], [A-Read], and [Call].

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Skip]</td>
<td>$\vdash \text{skip} : H \rightarrow H$</td>
</tr>
<tr>
<td>[ACQ]</td>
<td>$\vdash \text{check}(C); \text{acq}(x) : H \bullet \emptyset \rightarrow (H \cup C \checkmark \bullet A)$ where $C = \text{Checks}(H, \emptyset)$</td>
</tr>
<tr>
<td>[REL]</td>
<td>$\vdash \text{check}(C); \text{rel}(x) : H \bullet A \rightarrow (H \setminus {\checkmark }) \bullet A$ where $C = \text{Checks}(H, A)$</td>
</tr>
<tr>
<td>[Assign]</td>
<td>$\vdash x = e : H \bullet A[x := e] \rightarrow (H \cup {x = e}) \bullet A$ where $x \notin \text{Vars}(e)$</td>
</tr>
<tr>
<td>[Rename]</td>
<td>$\vdash x \leftarrow y : H \bullet A[x := y] \rightarrow H[y := x] \bullet A$</td>
</tr>
<tr>
<td>[New]</td>
<td>$\vdash x = \text{new} e : H \bullet (A \setminus x) \rightarrow H \bullet A$</td>
</tr>
<tr>
<td>[A-New]</td>
<td>$\vdash x = \text{new_array} z : H \bullet (A \setminus x) \rightarrow H \bullet A$</td>
</tr>
<tr>
<td>[Write]</td>
<td>$\vdash y.f = x : H \bullet (A \cup {y.f^{\circ}}) \rightarrow (H \cup {y.f^{\circ}}) \bullet A$</td>
</tr>
<tr>
<td>[A-Write]</td>
<td>$\vdash y[z] = x : H \bullet (A \cup {y[z^{\circ}]}) \rightarrow (H \cup {y[z^{\circ}]}) \bullet A$</td>
</tr>
<tr>
<td>[Read]</td>
<td>$\vdash x = y.f : H \bullet (A \setminus x \cup {y.f^{\circ}}) \rightarrow (H \cup {y.f^{\circ}}) \bullet A$</td>
</tr>
<tr>
<td>[A-Read]</td>
<td>$\vdash x = y[z] : H \bullet (A \setminus x \cup {y[z^{\circ}]}) \rightarrow (H \cup {y[z^{\circ}]}) \bullet A$</td>
</tr>
</tbody>
</table>

Figure 7. Check Placement Rules.

These orderings generate corresponding meet operators, where the meet on anticipated sets additionally takes history sets to reason about entailment.

$H_1 \cap H_2 = \{h \in H_1 \cup H_2 : H_1 \vdash a, H_2 \vdash a\}$

$H_1 \bullet A_1 \cap H_2 \bullet A_2 = \{a \in A_1 \cup A_2 : H_1 \bullet A_1 \vdash a, H_2 \bullet A_2 \vdash a\}$

**Analysis Rules** The analysis rules are somewhat complex due to their bidirectional nature and the subtle properties being captured. We present the technical details of our core rules below, but subsequent paper sections do not assume an in-depth understanding of their details.

**[Rel]**: Since past accesses need to be checked before a release, this rule targets the syntax check$(C); \text{rel}(x)$ and uses the function

$\text{Checks}(H, A) = \{p : p^\prec \in H, H \not\vdash p^\checkmark, H \bullet A \not\vdash p^\Diamond\}$

to ensure that the path set $C$ contains any path $p$ that was accessed ($p^\prec \in H$) but not yet checked and is not anticipated. (If $p$ is anticipated, then the future check on the anticipated access serves as the covering check for the past access.)

The post-history removes (1) all prior checks (denoted $\checkmark$) because these checks do not cover accesses after the release and (2) all prior accesses (denoted $\prec$) because we are leaving the legitimate check range for them.

**[Acq]**: This rule for check$(C); \text{acq}(x)$ ensures $C$ contains any path $p$ that was accessed but not checked. The post-history contains the newly checked paths (where $C^\checkmark$ abbreviates $\{p^\checkmark \mid p \in C\}$). The pre-anticipated set must be empty because any anticipated access would need to occur before this acquire.

**[Read]**: This rule matches the syntax $x = y.f$. To simplify our analysis, we require that the target of any assignment be a “fresh” variable not mentioned in the pre-history $H$. 

147
as the assignment would otherwise invalidate that history information. The [read] rule adds past access \( y.f^\sqsubset \) to the post-history. The pre-anticipated paths become \( A \setminus x \sqcup \{ y.f^\sqsubset \} \), where \( A \setminus x \) removes all properties mentioning \( x \) from \( A \).

[rename]: As mentioned above, assignments can only target “fresh” variables not in \( H \), but in some cases, e.g. before a loop back edge, we may need to modify an existing non-fresh variable \( y \). We cannot simply remove \( y \) from the history, as that might remove past accesses with pending checks, such as \( y.f^\sqsubset \). Instead, the renaming operation \( x \leftarrow y \) copies the value of \( y \) into a fresh variable \( x \), and replaces all mentions of \( y \) in the history \( H \) by \( x \), with the result that \( y \) is now “fresh” (not mentioned in the history) and can be an assignment target. To illustrate this rule, consider the renaming \( i \leftarrow i^\dag \) on line 5 in Figure 4(b). The history prior to the renaming contains \( a[0..1]^\dag \) and \( a[1]^\sqsubset \). After renaming, we have \( a[0..1]^\dag \) and \( a[1]^\sqsubset \), enabling us to continue deferring the checks for those accesses.

[write]: This rule for \( y.f = x \) adds the access \( y.f^\sqsubset \) to the post-history, and \( y.f^\circ \) to the pre-anticipated set.

[assign]: This rule for the assignment \( x = e \) adds the boolean expression \( x = e \) to the post-history. We require \( x \notin \text{vars}(e) \) to ensure the post-history does not refer to the pre-value of \( x \). The pre-anticipated set is computed from the \( A \) via the substitution \( A[x := e] \), which replaces all occurrences of \( x \) with \( e \) in each \( p^\circ \in A \). Since anticipated paths are not closed under this substitution, we remove from the result any syntactically ill-formed anticipated paths.

[if]: Conditionals may require checks to be placed at the end of each branch, and so this rule targets the syntax if \( \{ s_1; \text{check}(C_1) \} \{ s_2; \text{check}(C_2) \} \). This rule first computes the post-histories \( H'_1 \) and \( H'_2 \) and pre-anticipated sets \( A_1 \) and \( A_2 \) for \( s_1 \) and \( s_2 \). The merged history \( H'_1 \cap H'_2 \) describes properties holding after both branches but may leave out accesses that occurred only on one branch. We introduce the following variant of the \( \text{Checks} \) function to compute the unanticipated unchecked past accesses in \( H \) that must be checked when \( H \) is approximated by \( H' \):

\[
\text{Checks}(H, H', A) = \\
\{ p : p^\sqsubset \in H, H' \not\vdash p^\sqsubset, H \not\vdash p^\circ, H \cdot A \not\vdash p^\circ \}
\]

Thus, \( C_1 = \text{Checks}(H'_1, H'_1 \cap H'_2, A_{\text{out}}) \) are those paths that must be checked at the end of the “then” branch, and similarly for \( C_2 \) on the “else” branch. The contexts at the end of the branches are then \( H_1 \sqcup C'_1 \) and \( H_2 \sqcup C'_2 \), and these are merged via \( \sqcup \) to yield the final history \( H_{\text{out}} \). The anticipated pre-context \( A_{\text{in}} \) is computed by merging together the anticipated contexts preceding \( s_1 \) and \( s_2 \).

[loop]: Loops similarly require checks on the two paths meeting at the loop head, and this rule targets the form:

\[
\text{check}(C_{\text{in}}); \text{loop} \{ s; \{ \text{if be break } \}; \text{check}(C_{\text{back}}) \}
\]

In this rule, \( H_{\text{in}} \) and \( H_{\text{back}} \) are the pre-histories of \( \text{check}(C_{\text{in}}) \) and \( \text{check}(C_{\text{back}}) \), respectively, and \( H_{\text{inv}} \) is the loop-invariant history at the loop head. As in [if], the \( \text{Checks} \) function uses these sets and \( A_{\text{in}} \), the anticipated set at the loop head, to compute \( C_{\text{in}} \) and \( C_{\text{back}} \). The side conditions \( H_{\text{in}} \sqcup C'_{\text{in}} \sqsubseteq H_{\text{inv}} \) and \( H_{\text{back}} \sqcup C'_{\text{back}} \sqsubseteq H_{\text{inv}} \) ensure that properties in \( H_{\text{inv}} \) are true on all paths into the loop head.

Note that \( H_{\text{inv}}, H_{\text{in}}, \) and \( H_{\text{back}} \) are defined via mutual recursion; they are computed as part of a greatest fixed point computation over a method body. The computation is seeded with an initial conjecture for \( H_{\text{inv}} \) that is then refined via a form of predicate abstraction. (See Section 5). An analogous anticipated set \( A_{\text{inv}} \) characterizing what is anticipated prior to the loop exit test is used in the computation of \( A_{\text{in}} \).

[call]: A method call may require checks prior to the call if the callee performs synchronization (either directly or indirectly via a nested method call). Thus we match syntax of the form \( \text{check}(C); x = y.m(\tau) \). The function \( \text{KillSetHistory}(m) \) denotes the set of history properties killed by the side effects of method \( m \), and contains:

\[
\{ -\sigma^\circ \} \quad \text{if } m \text{ acquires a lock} \\
\{ -\sigma^\circ, -\circ^\circ \} \quad \text{if } m \text{ releases a lock}
\]

The function \( \text{KillSetAnticipated}(m) \) describes anticipated accesses killed by \( m \). It is \( \{ -\circ \} \) if \( m \) acquires a lock \( \emptyset \) otherwise. Our implementation pre-computes \( \text{KillSetHistory}(m) \) and \( \text{KillSetAnticipated}(m) \) using a separate whole program analysis. Checks are added before the call for any unchecked accesses \( C \) that are killed by the call, and the post-history \( H' \) is derived from the pre-history \( H \) and \( C \) by removing all such killed properties.

Correctness Sketch Our companion technical report [42] provides a detailed proof showing that the BigFoot algorithm described so far is correct in that it is address-precise. We present a short outline of our argument below.

We first formalize an operational semantics for BFJ that evaluates program \( P = \overline{D_1} s_1 || \ldots || s_n \) via a sequence of states \( \Sigma_0 \rightarrow s_1 \Sigma_1 \rightarrow s_2 \Sigma_2 \ldots \rightarrow s_n \Sigma_n \), where \( \Sigma_0 \) is an initial state for \( P \) and \( \Sigma_n \) is a final terminating state. This evaluation sequence yields a trace \( \alpha = a_1,a_2,a_3 \ldots a_n \) describing the memory accesses, race checks, and synchronization operations performed by \( P \).

We also define a judgement \( \overline{D_1}; \alpha \vdash \Sigma \) describing when a run-time state has correct checks in the context of an execution history \( \alpha \). This judgement most notably ensures that, for each thread \( t \), the context \( H \cdot A \) for thread \( t \)’s current program point is consistent with \( \Sigma \) and \( \alpha \). This judgement entails the following: 1) Each expression \( be \in H \) is true when evaluated by \( t \) in the current state \( \Sigma \). 2) If \( p^\circ \in H \) and \( p \) denotes a memory \( l \), there is an access to \( l \) in by \( t \) with no later release. (Each \( p^\circ \in H \) must have similar check). 3) Each check by \( t \) in \( \alpha \) is legitimate for a preceding access. 4) Each access to a location \( l \) by \( t \) in \( \alpha \) is either covered by a check, or \( t \) is still in that access’s covering check range and
there is some path $p$ denoting $l$ such that either $p^<$ is in $H$ or $p^<$ is in $A$.

The first two requirements show that the history context soundly approximates program behavior. The third and fourth guarantee that each check performed by $t$ is legitimate and that each access by $t$ has either been covered by a check or will be covered by deferred check performed later in the trace.

Provided $\vdash P$, the initial state satisfies the criteria for well-formed states (i.e., $D; e \not\vdash \Sigma_0$), and we show via a preservation argument that it holds for each subsequent state, including the last, i.e., $D; \alpha \not\vdash \Sigma_n$. Since each thread in $\Sigma_n$ has terminated and will perform no subsequent checks or accesses, the rules for (↓) imply that $\alpha$ has precise checks. Consequently, the checks in $P$ are address-precise. That is, if $\vdash P$ and $P$ generates a trace $\alpha$, then for any address $l$, $\alpha$ has a data race on $l$ if and only if it has a check race on $l$.

4. Check Coalescing & Shadow Compression

Post-Analysis Path Coalescing

In preparation for our shadow compression algorithms, we perform one last coalescing step on each set of checks added to the program. Specifically, for each check $\text{check}(C)$ statement, we divide the paths in $C$ into equivalence classes based on the path designator: that is, $d_1, f_1$ and $d_2, f_2$ are in the same class if $d_1$ and $d_2$ refer to the same object in the check’s pre-history written $H \vdash d_1 = d_2$, and similarly for array paths.

We then coalesce each group $d_1, f_1, d_2, f_2, \ldots, d_n, f_n$ sharing equivalent designators to the coalesced field path $d_1, f_1 / f_2 / \ldots / f_n$. We also coalesce each group of paths $d_1[b_1..c_1:k_1], \ldots, d_n[b_n..e_n:k_n]$ to one array path $d_1[b..e:k]$ such that the strided range “$b..e:k$” captures the exact same set of indices as the $n$ original strided ranges. This step necessitates solving a collection of integer constraints over program expressions, but those constraints have a form that cannot be handled by, e.g., Omega [41] or effectively solved directly via Z3. Thus, to find a suitable $b, e,$ and $k$, our implementation tries various combinations of the bounds and step sizes from the original strided ranges. This combinatorial approach can be expensive if there are a large number of strided ranges, but we have found it effective in practice. If a coalesced path cannot be found, we simply keep the original set of paths. We could alternatively try to divide the set into two or more coalesceable subsets, but this provided little benefit in practice.

Shadow Compression

A precise dynamic race detector typically maintains a distinct shadow location for each object field or array element. Thus, an object $pt$ with three fields requires three shadow locations and $\text{check}(pt.x/y/z)$ performs three shadow-location operations. Similarly, an array $a$ of $n$ elements requires $n$ shadow locations, and $\text{check}(a[0..n])$ performs $n$ shadow-location operations.

However, check coalescing enables us to identify groups of shadow locations that can be compressed into a single shadow location at run time with no loss in precision. Moreover, a coalesced check covering a compressible group only requires a single shadow-location operation, yielding substantial performance benefits. Compressible locations can be identified statically or dynamically. We have found the combination of static compression for object fields and dynamic compression for array elements yields the best performance.

Static Field Compression

We identify fields of a class that are compressible via a static shadow proxy analysis [25]. Given a class with fields $x$ and $y$, field $x$ is a proxy for $y$ if every check $\text{check}(p.0/x/y/.)$ also checks $p.x$. In this situation, any trace exhibiting a race on $p.y$ will also have a race on $p.x$. Hence, we can compress the shadow locations for $x$ and $y$ into a single location while still being able to distinguish race-free executions from those with races.

Identifying field proxies requires a single pass over all checks.

Dynamic Array Compression

We could express similar proxy relationships for array elements. For example, $a[0]$ could be a proxy for all array entries $a[0..n]$ if all checks on $a$ all have the form $\text{check}(a[0..n])$. Similarly, $a[1,2]$ could be the proxy for each $a[1]$ if all checks have the form $\text{check}(a[0..n:2])$ or $\text{check}(a[1..n:2])$. REDCARD [25] used this approach, but its static array proxy analysis failed to scale and was too imprecise to capture many proxy relationships, as we demonstrate in Section 6.

BIGFOOT instead makes array shadow compression choices dynamically using an extension of the approach introduced in the SLIMSTATE checker [55]. Specifically, BIGFOOT augments static array check coalescing with a complementary dynamic coalescing technique based on array footprints. For each array $A$, the BIGFOOT run time maintains a per-thread footprint of which indices must be checked prior to that thread’s next synchronization operation. When a thread $t$ performs $\text{check}(a[b..e:k])$, BIGFOOT adds the strided range $b..e:k$ to $t$’s footprint for $A$. In this way, many individual check operations that were not coalesced statically may be coalesced dynamically into a single, large footprint. At thread $t$’s next synchronization point, its footprint for $A$ is “committed” and the necessary shadow-location operations are performed to verify race freedom.

BIGFOOT initially compresses the shadow state for the entire array into a single shadow location. It then adaptively refines that representation whenever it must commit a footprint that is not consistent with the array’s current representation. As in SLIMSTATE, BIGFOOT supports compression modes matching common patterns of array accesses, including block-based and stride-based patterns. SLIMSTATE processes every individual array access at run time to build its dynamic footprints. By statically coalescing checks, BIGFOOT eliminates much of that overhead.

2While this optimization guarantees that we precisely identify race-free traces, we may not identify all memory locations with races since a race on $x$ may or may not imply a race on $y$. This subtlety goes away if we consider only symmetric proxy relations, e.g., when $y$ is also a proxy for $x$. 

149
5. Implementation

We have implemented our analysis in the BigFoot checker for Java. BigFoot consists of a static component (StaticBF) and a dynamic component (DynamicBF). StaticBF reads in a bytecode program and a list of classes and methods to transform, and it outputs a version of the program with explicit race checks for all object and array accesses in the specified methods. DynamicBF is the complementary dynamic race detector that reads in the instrumented program, runs it, and reports any races observed.

Extending the BFJ analysis to the full Java language is straightforward, and we describe the most important aspects of StaticBF below. BigFoot handles all basic synchronization operations present in Java, including locks, volatile variables, fork/join, and wait/notify, as described in [23].

Alias Expressions and Precision StaticBF augments BFJ’s set of boolean expression be with heap alias expressions of the form \( x = y.f \) and \( x = y[z] \), which enable us to reason about aliasing when deciding entailment. Those expressions are added to the history on field/array reads and are retained as long as they are valid under the assumption that the target is race free. If an alias expression is invalidated by a data race at run time, we may miss reporting some subsequent data races (because race checks were not placed in the necessary positions), but we will always detect the initial race.

For example, consider the code fragment to the right, which includes the alias expressions recorded by StaticBF. Those alias expressions enable StaticBF to conclude \( x = y \) at the check operation, meaning that the check on \( x.g \) covers the access to \( y.g \). Thus, no check on \( y.g \) is inserted. However, those alias assumptions could be violated by a racy write to \( a.f \) in between the two reads, and thus the race on \( a.f \) could effectively hide a race on \( y.g \).

While utilizing local alias expressions enables StaticBF to better optimize check placement, it means that, in theory, BigFoot is trace precise but not address precise. In practice, however, BigFoot was address-precise for all of our benchmark runs, which we verified via an additional dynamic analysis that checks that each observed execution trace performs precise checks (in the sense of Section 2).

5.1 StaticBF

StaticBF is built on top of the WALA analysis framework [34]. WALA represents methods as CFGs over SSA instructions and analyzes all methods in a call graph constructed using a 0-CFA analysis. To ensure method CFGs are amenable to our analysis, StaticBF performs an initial pass over the target to (1) rewrite each loop as an if statement containing a do-while loop matching BFJ’s syntax, and (2) eliminate all critical edges from the CFGs (see, e.g., [3]). We use Soot [50] for this pass. We also precompute KillSetHistory and KillSetAnticipated via a simple interprocedural dataflow analysis. StaticBF then inserts checks into each method using a method-local dataflow analysis.

The initial context for each program point is \( \{ h : h \in \text{History} \} \cup \{ a : a \in \text{Anticipated} \} \), and the analysis computes the greatest fixed point solution for those contexts according to the rules in Figure 7. To simplify the implementation, we compute context properties via separate passes for (1) boolean and alias expressions, (2) past accesses, (3) anticipated accesses, and finally (4) past checks and the set \( C \) for each check \( (C) \). All passes are forward analyses, except for the anticipated accesses pass.

StaticBF handles SSA \( \phi \)-functions as they were handled in RedCard [25]. Also as in RedCard, StaticBF tracks extended paths containing multiple field/array references (as in \( a[1].f \) or \( b.f.g \)), which are necessary for maintaining precision when merging contexts encoding equivalent aliasing facts via different local variables. We implement the entailment relations via the Z3 SMT Solver [16].

After applying the final coalescing step and static field proxy analysis described in Section 4, StaticBF generates a new version of the target code with the necessary checks inserted. These checks take the form of method calls into the DynamicBF run time. Paths in check statements refer to SSA variables and variables introduced via the [Rename] rule, and not the stack slots and locals present in the original bytecode. Thus, StaticBF inserts additional locals and load/store instructions to reify them in the instrumented target. Our relatively naive algorithm may introduce extraneous memory loads/stores, and we apply the Soot optimizer in a post-transformation pass to eliminate them.

Distinguishing Reads and Writes Up to this point, we have not distinguished reads and writes. However, StaticBF must do so because precise dynamic race detectors treat them differently. In particular, two concurrent accesses are considered conflicting only when at least one is a write.

To account for this, we extend our notions of legitimate and covering checks. A write check is only legitimate for a write access, but a read check is legitimate for both write and read accesses. A write check can cover write or read accesses, but a read check can only cover read accesses. In addition, contexts record whether each \( p^w \) and \( p^r \) is a read or write access, and whether each \( p^v \) is a read or write check. The analysis rules and coalescing operations are also extended appropriately.

Loop Invariants StaticBF infers the loop invariant \( H_{inv} \) for rule [Loop] via a form of Cartesian predicate abstraction [26] [30]. Specifically, StaticBF identifies the loop’s linear induction variables and trip count [28] [50] and then builds an initial set \( H_{heuristic} \) of boolean constraints and past accesses consistent with that information. Since this algorithm does not reason precisely about synchronization, function
calls, and various other bytecode features, it may produce some incorrect properties. Thus, StaticBF repeatedly analyzes the loop body to infer the maximal $H_{inv} \subseteq H_{heuristic}$ that is valid loop invariant as part of its dataflow analysis passes. StaticBF similarly infers the anticipated invariant $A_{inv}$ by constructing an initial $A_{heuristic}$ and computing the maximal valid $A_{inv} \subseteq A_{heuristic}$. If no induction variables can be identified, then $A_{heuristic}$ is the empty set, and no loop invariant are inferred. Irreducible loops and complex computations may be problematic for our algorithm, but it is quite effective in practice.

**Static Fields** In the JVM, a thread’s first access to a static field may synchronize with the declaring class’s static initializer to ensure proper behavior [34]. StaticBF provides a command line flag to treat static field accesses as potential synchronization so that checks will not be deferred across them. We use this flag for several benchmarks where this matters. (Other instructions that may synchronize with static initializers, e.g. type casts, are handled similarly.)

**Exceptions** StaticBF reasons about control paths for checked exceptions [29], but assumes unchecked exceptions, such as NullPointerExceptions, are errors in the target program and guarantees precision only for error-free traces. This is an artifact of our current implementation and not a fundamental limitation. Unchecked exceptions could be fully handled via a more sophisticated code translation scheme inside StaticBF, but given the complexity of the resulting code, a better approach would be to integrate parts of the analysis into the JVM’s exception mechanism. Our current treatment of exceptions did not lead to missed race checks in any of our benchmark experiments.

### 5.2 DynamicBF

We built our complementary DynamicBF dynamic analysis in the RoadRunner framework [24]. Dynamic footprinting and array shadow compression are implemented as in the earlier SlimState checker and we use FastTrack’s adaptive epoch representation [29] for shadow locations. BigFoot follows RoadRunner’s standard treatment of libraries: fields of Java’s core library classes are not checked for races, and synchronization operations internal to those libraries are assumed not to be used to protect any of the target’s data and are ignored. However, several key library methods from java.lang.Object and java.lang.Thread, such as Object.notify and Thread.start, are treated specially as synchronizing operations. These assumptions are shared by all checkers we evaluate, and also included in StaticBF. Their violation may impact precision.

### 6. Validation

We validate BigFoot’s performance by comparing it to FastTrack [23], SlimState [55], RedCard [25], and SlimCard (Section 6.2) on the JavaGrande [32] and DaCapo [6] benchmark suites. To facilitate comparison the detectors share as much common implementation as possible.

We configured the JavaGrande programs to use their largest data sizes and 16 worker threads. We also fixed racy barrier implementations in several of them. We configured the DaCapo benchmarks to use their default sizes, but we exclude tradebeans and eclipse because of incompatibilities with our underlying framework and other known issues [55]. We additionally exclude several specific methods from the other programs that RoadRunner cannot properly instrument because the resulting code would exceed a JVM limit on method size. Several DaCapo programs use reflection heavily. To facilitate building the call graph for those programs in StaticBF and RedCard, we used a modified version of Tamiflex [7] to eliminate reflection.

Since RoadRunner does not support the specialized class loading features used by the DaCapo test harness, we implemented a simplified version of that harness. It runs a target’s workload several times in a warm up phase and then measures the running time for 10 iterations of the workload. We used that harness for the JavaGrande programs as well. We report the means of ten such trials.

We verified all race detection tools examined reported the same races (modulo variations due scheduling) manually. All experiments were performed on a 2.4GHz 16-core AMD Opteron processor with 64GB running Ubuntu Linux and Oracle’s Java HotSpot 64-bit Server VM version 1.8.

#### 6.1 StaticBF

BigFoot took 0.16 seconds per method on average to process the benchmark programs, as shown in Table 1.

<table>
<thead>
<tr>
<th>Program</th>
<th>BaseTime</th>
<th>BF</th>
<th>UB</th>
<th>BF Time Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avrora</td>
<td>151</td>
<td>151</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>Flink</td>
<td>151</td>
<td>151</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>Jikes</td>
<td>151</td>
<td>151</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>Whetstone</td>
<td>151</td>
<td>151</td>
<td>0</td>
<td>0.00</td>
</tr>
</tbody>
</table>

With careful caching of SMT solver results, only about 10% of this time was spent solving Z3 queries. Together, call graph construction for computing method kill sets and reasoning about heap and boolean constraints accounted for more than half of the running time in most cases. We have focused on implementation simplicity and high precision. More careful tuning would likely lead to significant improvements.

#### 6.2 DynamicBF Time Overhead

Figure 8 shows, for each program, how many race checks on shadow locations FastTrack (left graph) and BigFoot (middle graph) perform relative to the number of heap accesses. FastTrack performs a check on each access, meaning its check ratio ($\frac{\text{Checks}}{\text{HeapAccesses}}$) is always 1. For BigFoot, the average check ratio is 0.43, and much smaller for some programs, particularly those in which traversals over large arrays are covered by a single coalesced check. BigFoot’s check ratio is also substantially lower than that of RedCard (0.73), SlimState (1.0), and SlimCard (0.76).

Table 1 shows the base running time for each program and the overhead of each checker. Overhead is the additional time beyond the base time necessary to check a program:

$$\text{CheckerOverhead} = \text{CheckerTime} - \text{BaseTime}$$
Comparison to FastTrack. BigFoot is significantly faster than the other detectors. As shown in the last column of Table 1, BigFoot incurs only 39% of the overhead of FastTrack. The right-most graph in Figure 8 shows this improvement visually. BigFoot is most effective on programs exhibiting highly-structured access patterns to large data sets, and thus low check ratios, such as crypt, moldyn, montecarlo, and sunflow. Moving checks out of loops and coalescing them accounts for much of this improvement. BigFoot is also effective on programs with many redundant checks that can be eliminated altogether, such as sparse.

It is interesting to note that several programs do not follow the expected trend. For series, the FastTrack overhead of only 1% is mostly due to internal ROADRUNNER bookkeeping, which leaves little opportunity for improvement. The lufact benchmark performs a triangular array computation whose array accesses are readily coalesced by BigFoot, resulting in a small check ratio. However, that triangular pattern is not amenable to our online array state compression algorithm, meaning that the array’s shadow representation becomes fine-grained and each coalesced check induces many shadow location operations.

In other benchmarks, such as h2 and avrora, bookkeeping for synchronization operations accounts for a greater fraction of checking overhead, diminishing the benefit of optimizing memory operations with BigFoot. The degraded performance for tomcat appears to be caused by higher contention on internal ROADRUNNER data structures when using BigFoot.

Field compression via proxies accounted for about 5% of the savings in general, but over 50% of the savings in raytracer and sunflow.

Comparison to RedCard. RedCard eliminates one form of redundant check.25, namely checks on accesses where the current thread has already accessed (and checked) that location within the same release-free span. The BigFoot check placement algorithm is able to eliminate other forms of redundancy by both reasoning about anticipated accesses and moving checks. For example, BigFoot can eliminate more redundant checks and move checks out of loops, as shown in Figure 6.

RedCard also performs static proxy analysis, but the array component crucially depends upon globally-computed allocation-site points-to information. As such, RedCard’s static analysis fails to terminate within four hours on many benchmarks, as indicated by the † symbol in Table 1. We use RedCard’s redundancy analysis without proxies for those programs. Moreover, imprecisions in the proxy analysis limit its effectiveness even on small programs.

Overall, the check ratio and overhead reduction for RedCard were 0.73 and 17%, respectively. In contrast, the check ratio and overhead reduction for BigFoot were 0.43 and 61%. BigFoot’s ability to move checks out of loops is key to achieving this improvement, particularly when coupled with dynamic array shadow compression.

Comparison to SlimState. SlimState introduced the dynamic array compression scheme we use in BigFoot, but its check ratio is 1 because it processes every access at run time. BigFoot offers two crucial improvements: 1) BigFoot eliminates many redundant checks. 2) While SlimState must process every individual array access at run time to build its footprints, BigFoot statically coalesces array checks where possible, thereby reducing the amount of run-time footprint processing and eliminating much of SlimState’s dynamic footprint construction overhead. BigFoot’s overhead is less than half of SlimState’s as a result. Field compression, and moving field checks out of loops, contributes to the performance savings as well.
While we have focused primarily on running time, we also report the target program’s memory requirements, as well as the overheads for each checker in Table 1. Following the methodology of earlier work [55], we measure memory as the smallest heap permitting successful execution of the target program, which we find by iteratively shrinking the JVM’s maximum heap until the program crashes or fails to terminate within thrice the time to run with a 64 GB heap.

**Comparison to SLIMCARD** SLIMCARD combines REDCard’s static check elimination and field proxy analysis with SLIMSTATE’s dynamic array state compression. We did not include static proxy analysis for arrays in SLIMCARD because integrating the run-time bookkeeping necessary to support static array proxies [25] into SLIMSTATE’s analysis led to worse performance. As a result, SLIMCARD has an overall check ratio of 76%, which is a few percent higher than REDCard’s ratio (73%).

As expected, the combined analysis improves upon SLIMSTATE by eliminating many redundant checks and incurs only 70% of FASTTrack’s overhead. However, SLIMCARD still experiences the same overheads related to the construction of footprints at run time as SLIMSTATE. Moreover, it cannot move checks out of loops and coalesce them, which are crucial for achieving BigFoot’s much better performance. SLIMCARD’s memory overhead did not differ significantly from SLIMSTATE’s or BigFoot’s.

### 6.3 DynamicBF Memory Overhead

While we have focused primarily on running time, we also report the target program’s memory requirements, as well as the overheads for each checker in Table 1. Following the methodology of earlier work [55], we measure memory as the smallest heap permitting successful execution of the target program, which we find by iteratively shrinking the JVM’s maximum heap until the program crashes or fails to terminate within thrice the time to run with a 64 GB heap.

**BigFoot, SLIMSTATE, and SLIMCARD** reduce space overhead by about 26–28% when compared to FASTTrack. These three tools utilize the same dynamic array compression scheme. SLIMCARD and BigFoot additionally uses field compression, while field compression improved time, it did not lead to sizable space reductions. Inspection of the programs for which field compression made the greatest speed difference revealed that there were never sufficiently many objects with compressed fields alive at the same time to sizably impact overall space needs.

The limited impact of static compression on space can also be seen by comparing the space overhead of REDCard to FASTTrack. The only fundamental space difference is due to REDCard’s use of compression for field and array proxies, but again, there is little overall impact.

### 7 Other Related Work

In addition to REDCard and SLIMSTATE, described earlier, much work has focused on improving the performance of dynamic race detection. Many precise tools, such as
Many static analyses for identifying races have also been introduced to reduce these overheads. A common approach for further reducing overhead is to use a single shadow location for whole arrays and objects [9, 13, 23, 39, 40, 51], although this may generate false alarms, motivating additional technology to see if a reported warning reflects a real race [11, 21].

Another approach for reducing overheads is to use sampling [8, 20, 22], again with some loss of soundness. Eraser verifies race-freedom for data that is thread-local, read-shared, or lock protected [44], and has been extended to produce fewer false alarms [11, 21, 59, 47, 57].

Several dynamic checkers defer the processing of accessesRecPlay [53], which are expensive. FASTTrack introduced epochs [25] to reduce these overheads. A common approach for further reducing overhead is to use a single shadow location for whole arrays and objects [9, 13, 23, 39, 40, 51], although this may generate false alarms, motivating additional technology to see if a reported warning reflects a real race [11, 21].

Another approach for reducing overheads is to use sampling [8, 20, 22], again with some loss of soundness. Eraser verifies race-freedom for data that is thread-local, read-shared, or lock protected [44], and has been extended to produce fewer false alarms [11, 21, 59, 47, 57].

Several dynamic checkers defer the processing of accessesRecPlay [53], which are expensive. FASTTrack introduced epochs [25] to reduce these overheads. A common approach for further reducing overhead is to use a single shadow location for whole arrays and objects [9, 13, 23, 39, 40, 51], although this may generate false alarms, motivating additional technology to see if a reported warning reflects a real race [11, 21].

Another approach for reducing overheads is to use sampling [8, 20, 22], again with some loss of soundness. Eraser verifies race-freedom for data that is thread-local, read-shared, or lock protected [44], and has been extended to produce fewer false alarms [11, 21, 59, 47, 57].

Several dynamic checkers defer the processing of accessesRecPlay [53], which are expensive. FASTTrack introduced epochs [25] to reduce these overheads. A common approach for further reducing overhead is to use a single shadow location for whole arrays and objects [9, 13, 23, 39, 40, 51], although this may generate false alarms, motivating additional technology to see if a reported warning reflects a real race [11, 21].

Another approach for reducing overheads is to use sampling [8, 20, 22], again with some loss of soundness. Eraser verifies race-freedom for data that is thread-local, read-shared, or lock protected [44], and has been extended to produce fewer false alarms [11, 21, 59, 47, 57].

Entries in Table 2 are rounded. We report the mean over 1,000 runs of each benchmark under each setting, except for FT which reports the mean over 100 runs. We thank our shepherd Michael Pradel, the anonymous reviewers, Shaz Qadeer, and James Wilcox for their feedback. This work was supported, in part, by NSF Grants 1337278, 1421051, 1421016, and 1439042.

<table>
<thead>
<tr>
<th>Program</th>
<th>Base Mem (MB)</th>
<th>Space Overhead</th>
<th>Base</th>
<th>FT RC</th>
<th>SS SC</th>
<th>SC FT</th>
<th>BF FT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>crypt</td>
<td>193.73</td>
<td>26.27</td>
<td>(0.97)</td>
<td>(0.04)</td>
<td>(0.04)</td>
<td>(0.04)</td>
<td>(0.04)</td>
</tr>
<tr>
<td>series</td>
<td>22.01</td>
<td>4.45</td>
<td>(1.02)</td>
<td>(0.58)</td>
<td>(0.59)</td>
<td>(0.57)</td>
<td></td>
</tr>
<tr>
<td>lufact</td>
<td>32.15</td>
<td>10.16</td>
<td>(1.00)</td>
<td>(1.10)</td>
<td>(1.10)</td>
<td>(1.11)</td>
<td></td>
</tr>
<tr>
<td>moldyn</td>
<td>16.20</td>
<td>5.44</td>
<td>(0.82)</td>
<td>(0.91)</td>
<td>(0.80)</td>
<td>(0.82)</td>
<td></td>
</tr>
<tr>
<td>montecarlo</td>
<td>622.83</td>
<td>3.67</td>
<td>(1.00)</td>
<td>(0.30)</td>
<td>(0.30)</td>
<td>(0.30)</td>
<td></td>
</tr>
<tr>
<td>sparse</td>
<td>98.11</td>
<td>5.64</td>
<td>(1.01)</td>
<td>(1.44)</td>
<td>(1.05)</td>
<td>(0.79)</td>
<td></td>
</tr>
<tr>
<td>sor</td>
<td>32.12</td>
<td>5.11</td>
<td>(1.00)</td>
<td>(1.40)</td>
<td>(1.40)</td>
<td>(2.48)</td>
<td></td>
</tr>
<tr>
<td>batik</td>
<td>44.74</td>
<td>3.78</td>
<td>(0.99)</td>
<td>(0.75)</td>
<td>(0.95)</td>
<td>(1.00)</td>
<td></td>
</tr>
<tr>
<td>raytracer</td>
<td>16.42</td>
<td>3.67</td>
<td>(0.96)</td>
<td>(0.60)</td>
<td>(0.57)</td>
<td>(0.60)</td>
<td></td>
</tr>
<tr>
<td>tomcat</td>
<td>19.59</td>
<td>4.81</td>
<td>(0.99)</td>
<td>(0.98)</td>
<td>(0.99)</td>
<td>(1.14)</td>
<td></td>
</tr>
<tr>
<td>sunflow</td>
<td>10.42</td>
<td>9.50</td>
<td>(0.91)</td>
<td>(0.93)</td>
<td>(0.88)</td>
<td>(0.86)</td>
<td></td>
</tr>
<tr>
<td>luindex</td>
<td>6.15</td>
<td>16.3</td>
<td>(0.98)</td>
<td>(0.96)</td>
<td>(0.96)</td>
<td>(0.52)</td>
<td></td>
</tr>
<tr>
<td>pmd</td>
<td>30.24</td>
<td>6.02</td>
<td>(1.05)</td>
<td>(1.02)</td>
<td>(1.03)</td>
<td>(1.09)</td>
<td></td>
</tr>
<tr>
<td>fop</td>
<td>28.07</td>
<td>6.35</td>
<td>(1.00)</td>
<td>(0.98)</td>
<td>(0.97)</td>
<td>(0.99)</td>
<td></td>
</tr>
<tr>
<td>lusearch</td>
<td>12.04</td>
<td>7.00</td>
<td>(1.00)</td>
<td>(0.57)</td>
<td>(0.57)</td>
<td>(0.57)</td>
<td></td>
</tr>
<tr>
<td>avrora</td>
<td>2.09</td>
<td>15.22</td>
<td>(1.01)</td>
<td>(1.01)</td>
<td>(1.01)</td>
<td>(1.01)</td>
<td></td>
</tr>
<tr>
<td>jython</td>
<td>24.06</td>
<td>5.97</td>
<td>(1.03)</td>
<td>(0.96)</td>
<td>(0.96)</td>
<td>(1.02)</td>
<td></td>
</tr>
<tr>
<td>xalan</td>
<td>8.20</td>
<td>11.00</td>
<td>(1.00)</td>
<td>(0.84)</td>
<td>(0.84)</td>
<td>(0.82)</td>
<td></td>
</tr>
<tr>
<td>h2</td>
<td>259.71</td>
<td>3.90</td>
<td>(1.06)</td>
<td>(1.10)</td>
<td>(1.10)</td>
<td>(1.10)</td>
<td>(0.93)</td>
</tr>
</tbody>
</table>

Geo Mean 6.84 (0.99) (0.73) (0.74) (0.72)

Table 2. Checker space overhead relative to FASTTrack.


