BIGFOOT: Static Check Placement for Dynamic Race Detection

Dustin Rhodes Cormac Flanagan
University of California, Santa Cruz, USA

Stephen N. Freund
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, 37, 53] and dynamic [15, 38, 40, 44, 45, 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`)

PLDI'17, June 18–23, 2017, Barcelona, Spain
© 2017 ACM. 978-1-4503-4988-8/17/06...$15.00
http://dx.doi.org/10.1145/3062341.3062350

Standard Race Checks

```
class Point {
  int x, y, z;
  void move(int dx, int dy, int dz) {
    int tmp;
    CheckRead(this.x); tmp = this.x;
    CheckWrite(this.x); this.x = tmp + dx;
    CheckRead(this.y); tmp = this.y;
    CheckWrite(this.y); this.y = tmp + dy;
    CheckRead(this.z); tmp = this.z;
    CheckWrite(this.z); this.z = tmp + dz;

  }
}
void movePts(Point[] a, int lo, int hi) {
  for(int i = lo; i < hi; i++) {
    CheckRead(a[i]);
    a[i].move(1, 1, 1);
  }
}
```

BIGFOOT Race Checks

```
class Point {
  int x, y, z;
  void move(int dx, int dy, int dz) {
    int tmp;
    tmp = this.x;
    this.x = tmp + dx;
    tmp = this.y;
    this.y = tmp + dy;
    tmp = this.z;
    this.z = tmp + dz;
    CheckWrite(this.x/y/z);
  }
}
void movePts(Point[] a, int lo, int hi) {
  for(int i = lo; i < hi; i++) {
    a[i].move(1, 1, 1);
  }
  CheckRead(a[lo..hi]);
}
```

Figure 1. Check placement for precise data race detection.

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 $\texttt{lo}, \texttt{lo}+1, \ldots, \texttt{hi}-2, \texttt{hi}-1$.

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:

```
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

Race Detector	Check Motion and Coalescing		Red. Check Elimination	Metadata Compression		Run-Time Overhead
	objects	arrays		objects	arrays	
FASTTRACK [23]	no	no	no	no	no	7.3x
REDCARD [25]	no	no	static	static proxy	static proxy, global	6.0x
SLIMSTATE [55]	no	dynamic	no	no	dynamic	6.0x
SLIMCARD	no	dynamic	static	static proxy	dynamic	5.1x
BIGFOOT	**static**	**static**+dynamic	static, better	static proxy	dynamic	2.5x

Figure 2. Comparison to prior precise dynamic race detectors, and SLIMCARD (which combines REDCARD and SLIMSTATE).

"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 6). 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.[1] 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.

[1] REDCARD and SLIMCARD exhibit similar precision properties for the same reasons.

```
1: acq(lock);
2: x = b.f;
3: rel(lock);
4: y = b.f;
5: check(b.f);
6: acq(lock);
7: z = b.f;
8: rel(lock);
```

$$\emptyset \bullet \{\texttt{b.f}^{\diamond}\}$$
$$\{\texttt{b.f}^{\triangleleft}\} \bullet \{\texttt{b.f}^{\diamond}\}$$
$$\emptyset \bullet \{\texttt{b.f}^{\diamond}\}$$
$$\{\texttt{b.f}^{\triangleleft}\} \bullet \emptyset$$
$$\{\texttt{b.f}^{\triangleleft}, \texttt{b.f}^{\checkmark}\} \bullet \emptyset$$
$$\{\texttt{b.f}^{\triangleleft}, \texttt{b.f}^{\checkmark}\} \bullet \{\texttt{b.f}^{\diamond}\}$$
$$\{\texttt{b.f}^{\triangleleft}, \texttt{b.f}^{\checkmark}\} \bullet \emptyset$$
$$\emptyset \bullet \emptyset$$

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.)

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.

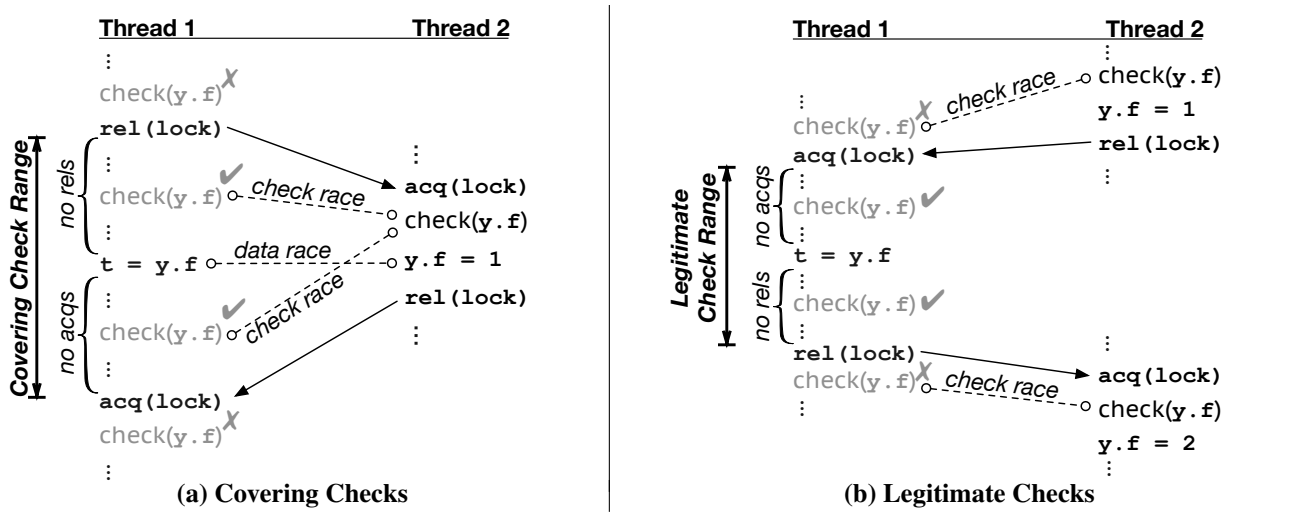

Figure 4. Precise and imprecise check placement locations.

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 are detected. 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 3, 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 $\overline{D}$ and a collection of concurrent threads $s_1 \| ... \| s_n$. Each class definition D contains field and method declarations. Each field declaration is simply a field name f. Each method declaration $m(\overline{x})\{s;\ \texttt{return}\ z\}$ includes a unique method m, formal parameters $\overline{x}$, and a body s followed by a return of the local variable z. We omit static

$$
\begin{array}{lcll}
P & \in & \mathit{Program} & ::= \quad \overline{D}\ s_1 \| \ldots \| s_n \\
D & \in & \mathit{Defn} & ::= \quad \texttt{class}\ c\ \{\ \overline{f}\ \overline{meth}\ \} \\
meth & \in & \mathit{Method} & ::= \quad m(\overline{x})\ \{\ s;\ \texttt{return}\ z\ \}
\end{array}
$$

$$
\begin{array}{lcll}
s & \in & \mathit{Stmt} & ::= \quad \texttt{skip} \mid s; s \mid \texttt{if}\ be\ s\ s \\
 & & & \quad \mid \texttt{loop}\{\ s;\ \{\ \texttt{if}\ be\ \texttt{break}\ \};\ s\ \} \\
 & & & \quad \mid x = e \mid x \leftarrow y \mid \texttt{acq}(y) \mid \texttt{rel}(y) \\
 & & & \quad \mid x = \texttt{new}\ c \mid y.f = x \mid x = y.f \\
 & & & \quad \mid x = \texttt{new\_array}\ z \mid y[z] = x \mid x = y[z] \\
 & & & \quad \mid x = y.m(\overline{z}) \mid \texttt{check}(C)
\end{array}
$$

$$
\begin{array}{lcll}
e & \in & \mathit{Expr} & ::= \quad x \mid v \mid e = e \mid \ldots \\
be & \in & \mathit{BoolExpr} & \subseteq \quad \mathit{Expr} \\
C & \in & \mathit{PathSet} & ::= \quad 2^{\mathit{Path}} \\
p & \in & \mathit{Path} & ::= \quad x.f \mid x[r] \\
r & \in & \mathit{StridedRange} & ::= \quad e..e:e
\end{array}
$$

$$
\begin{array}{rcl@{\qquad}rcl}
c & \in & \mathit{ClassName} & f & \in & \mathit{FieldName} \\
m & \in & \mathit{MethodName} & x, y, z & \in & \mathit{Var}
\end{array}
$$

Figure 5. BFJ Syntax.

types and local variable declarations, which are orthogonal to our formal development. We leave the set of expressions e unspecified but assume it includes at least `null`, boolean values, and local variables.

To facilitate our technical development, BFJ statements are in A-normal form [27] and include a loop construct with the exit test in the middle of the loop body. We motivate and describe the renaming operator $x \leftarrow y$ below.

BFJ includes the statement `check`(C) to explicitly check for races on each heap location described by a path $p \in C$. A path of the form $x.f$ describes an object field, and path of the form $x[r]$ describes array accesses, where r is a *strided range* of the form "$b..e : k$" represents the set of indices $\{b + ik \ : \ b \leq b + ik < e\}$ to be checked. We use b and $b..e$ to abbreviate singleton ("$b..(b+1) : 1$") and continuous ("$b..e:1$") strided ranges, respectively. We defer distinguishing read checks and write checks until Section 5.

3.2 Analysis Contexts

The BIGFOOT analysis is intraprocedural, analyzing and inserting checks into each method one at a time. Within each method, the analysis infers a *context* $H \bullet A$ for each program point that describes the known *history properties* H and *anticipated properties* A at that point:

$$\mathit{Context} ::= H \bullet A \qquad H \subseteq \mathit{History} \quad A \subseteq \mathit{Anticipated}$$

$$
\begin{array}{lcl}
h \in \mathit{History} & ::= & be \mid p^{\triangleleft} \mid p^{\checkmark} \\
a \in \mathit{Anticipated} & ::= & p^{\diamond}
\end{array}
$$

These properties capture the following notions:

- Boolean expressions be from, *e.g.*, branch tests.
- Past accesses $p^{\triangleleft}$, meaning that path p was previously accessed, with no subsequent release. The analysis must ensure there is a corresponding covering check.
- Past checks $p^{\checkmark}$, meaning that p was previously checked within the method, with no subsequent release.
- Anticipated accesses $p^{\diamond}$, meaning that the continuation after the program point will access p (and therefore check p), with no intervening acquire.

3.3 Check Placement Algorithm Overview

The BIGFOOT check placement 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 BIGFOOT uses context information to place checks, we examine the analysis contexts in Figure 3. 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.

BIGFOOT adds a past access $p^{\triangleleft}$ to the history whenever the code accesses p, and before an acquire it inserts a check for any past access $p^{\triangleleft}$ with no past covering check $p^{\checkmark}$, 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, BIGFOOT removes each past access $p^{\triangleleft}$ 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 $p^{\triangleleft}$ like this typically requires BIGFOOT 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 ($p^{\checkmark}$ 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 $p^{\checkmark}$ 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 6(a). The merged context $\emptyset \bullet \{\texttt{b.f}^{\diamond}\}$ after the `if` describes properties holding after both branches, and it omits past accesses occurring only on one branch. BIGFOOT 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.

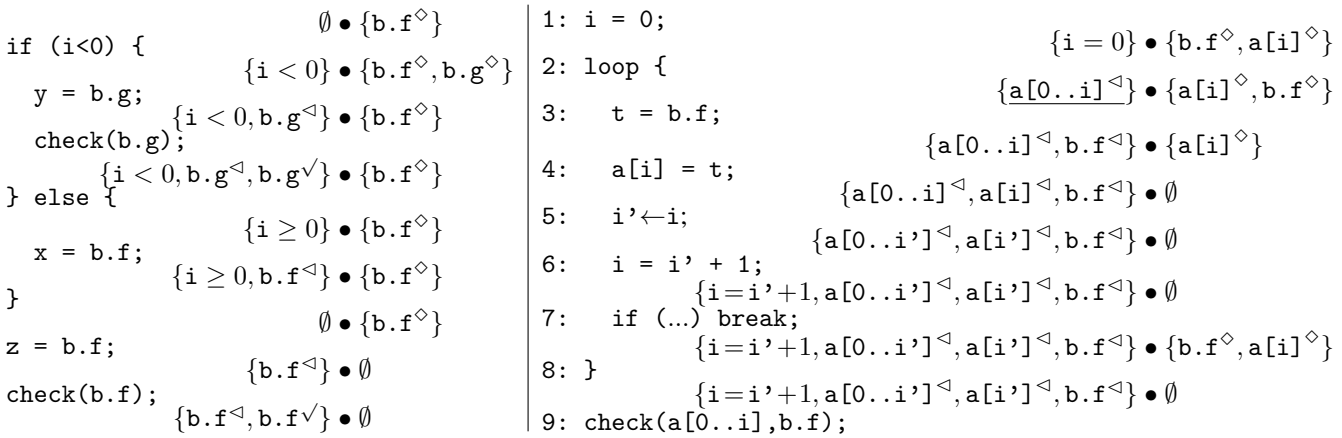

Figure 6. Analysis contexts and check placements for BFJ method bodies containing (a) an `if` statement and (b) a loop.

Figure 6(b) illustrates how loops are handled. To simplify our analysis, we require that the target x of any assignment be a "fresh" variable not mentioned in the preceding history, as the assignment would otherwise invalidate that history information. The operation `i' ← i` copies the value of `i` into a fresh variable `i'` and replaces all mentions of `i` in the history by `i'`, thereby ensuring `i` is afterwards fresh, that is, not mentioned in the history. BIGFOOT inserts renaming statements on demand, but for simplicity our presentation assumes any necessary renamings already exist.

BIGFOOT places all necessary checks at line 9 after the loop using the following technique. First, BIGFOOT synthesizes a loop invariant history that captures the set of accesses that have been performed whenever execution reaches line 2. The invariant for our example is the underlined history $H_{inv} = \{\underline{\texttt{a[0..i]}}^{\lhd}\}$. On entry to the loop, H_{inv} holds because $\texttt{i} = 0$, meaning no array elements have been accessed. On the loop back edge, H_{inv} is entailed by the loop body's final history $\{\texttt{i}=\texttt{i'}+1, \texttt{a[0..i']}^{\lhd}, \texttt{a[i']}^{\lhd}, \texttt{b.f}^{\lhd}\}$.

BIGFOOT defers checks until after the loop whenever possible. In this case, the history at the loop exit on line 7 contains $\texttt{a[0..i']}^{\lhd}$ (the invariant rewritten due to the renaming of `i` to `i'` at line 5) and $\texttt{a[i']}^{\lhd}$ (the similarly rewritten access from line 4). That history context captures all accesses that must be checked after the loop. Given that `i' = i + 1`, BIGFOOT places the single check of `a[0..i]` at line 9 to cover all array accesses from inside the loop.

BIGFOOT requires no global analysis to move the checks out of the loop because all variables referenced in the code are local and cannot be changed by other methods or threads.

This example also demonstrates that anticipation is crucial for moving some checks out of loops. At the end of the loop on line 8, the history contains $\texttt{b.f}^{\lhd}$, but the back edge returns to loop head on line 2, where $\texttt{b.f}^{\lhd}$ is not in the history. This would normally necessitate placing a check on `b.f` inside the loop before the back edge. However, since $\texttt{b.f}^{\Diamond}$ is anticipated at the loop head, we can avoid checking `b.f` inside the loop and defer the check until after the 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 $\vdash s : H \bullet A \rightarrow H' \bullet A'$ defined in Figure 7. The contexts $H \bullet A$ and $H' \bullet A'$ 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 $\vdash s : H \bullet A \rightarrow H' \bullet A'$ 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 $H = \{z[i]^{\lhd}, i = j\}$ then we can safely infer that H entails $z[j]^{\lhd}$, written $H \vdash z[j]^{\lhd}$. Similarly, we introduce *anticipated entailment* ($H \bullet A \vdash a$), as in $\{i < 10\} \bullet \{x[0..10]^{\Diamond}\} \vdash x[0..i]^{\Diamond}$. Our implementation uses Z3 [16] 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:

$$H_1 \sqsubseteq H_2 \quad \textit{iff} \quad \forall h \in H_1.\ H_2 \vdash h$$
$$H \vdash A_1 \sqsubseteq A_2 \quad \textit{iff} \quad \forall a \in A_1.\ H \bullet A_2 \vdash a$$

$\boxed{\vdash s : H\bullet A \to H'\bullet A'}$

(We assume $x \notin \mathit{Vars}(H)$ in the rules modifying x: [ASSIGN], [RENAME], [NEW], [READ], [A-NEW], [A-READ], and [CALL].)

$$
\begin{array}{lrcll}
\text{[SKIP]} & \vdash \texttt{skip} : H\bullet A & \to & H\bullet A & \\
\text{[ACQ]} & \vdash \texttt{check}(C);\texttt{acq}(x) : H\bullet\emptyset & \to & (H \cup C^{\surd})\bullet A & \text{where } C = \mathit{Checks}(H,\emptyset) \\
\text{[REL]} & \vdash \texttt{check}(C);\texttt{rel}(x) : H\bullet A & \to & (H \setminus \{\_^{\surd}, \_^{\lhd}\})\bullet A & \text{where } C = \mathit{Checks}(H, A) \\
\text{[ASSIGN]} & \vdash x = e : H\bullet A[x := e] & \to & (H \cup \{x = e\})\bullet A & \text{where } x \notin \mathit{Vars}(e) \\
\text{[RENAME]} & \vdash x \leftarrow y : H\bullet A[x := y] & \to & H[y := x]\bullet A & \\
\text{[NEW]} & \vdash x = \texttt{new}\ c : H\bullet(A \setminus x) & \to & H\bullet A & \\
\text{[A-NEW]} & \vdash x = \texttt{new\_array}\ z : H\bullet(A \setminus x) & \to & H\bullet A & \\
\text{[WRITE]} & \vdash y.f = x : H\bullet(A \cup \{y.f^{\diamond}\}) & \to & (H \cup \{y.f^{\lhd}\})\bullet A & \\
\text{[A-WRITE]} & \vdash y[z] = x : H\bullet(A \cup \{y[z]^{\diamond}\}) & \to & (H \cup \{y[z]^{\lhd}\})\bullet A & \\
\text{[READ]} & \vdash x = y.f : H\bullet(A \setminus x \cup \{y.f^{\diamond}\}) & \to & (H \cup \{y.f^{\lhd}\})\bullet A & \\
\text{[A-READ]} & \vdash x = y[z] : H\bullet(A \setminus x \cup \{y[z]^{\diamond}\}) & \to & (H \cup \{y[z]^{\lhd}\})\bullet A &
\end{array}
$$

[IF]
$$
\frac{\begin{array}{c} H_1 = H_{in} \cup \{be\} \quad \vdash s_1 : H_1\bullet A_1 \to H_1'\bullet A_{out} \\ H_2 = H_{in} \cup \{\neg be\} \quad \vdash s_2 : H_2\bullet A_2 \to H_2'\bullet A_{out} \\ C_1 = \mathit{Checks}(H_1', H_1' \sqcap H_2', A_{out}) \quad C_2 = \mathit{Checks}(H_2', H_1' \sqcap H_2', A_{out}) \\ A_{in} = H_1\bullet A_1 \sqcap H_2\bullet A_2 \quad H_{out} = (H_1' \cup C_1^{\surd}) \sqcap (H_2' \cup C_2^{\surd}) \end{array}}{\vdash \texttt{if}\ be\ \{s_1;\ \texttt{check}(C_1)\}\ \{s_2;\ \texttt{check}(C_2)\} : H_{in}\bullet A_{in} \to H_{out}\bullet A_{out}}
$$

[SEQ]
$$
\frac{\vdash s_1 : H_1\bullet A_1 \to H_2\bullet A_2 \quad \vdash s_2 : H_2\bullet A_2 \to H_3\bullet A_3}{\vdash s_1; s_2 : H_1\bullet A_1 \to H_3\bullet A_3}
$$

[LOOP]
$$
\frac{\begin{array}{c} \vdash s : H_{inv}\bullet A_{in} \to H\bullet A_{inv} \\ H_{back} = H \cup \{\neg be\} \quad H_{out} = H \cup \{be\} \\ C_{in} = \mathit{Checks}(H_{in}, H_{inv}, A_{in}) \quad H_{in} \cup C_{in}^{\surd} \sqsupseteq H_{inv} \\ C_{back} = \mathit{Checks}(H_{back}, H_{inv}, A_{in}) \quad H_{back} \cup C_{back}^{\surd} \sqsupseteq H_{inv} \\ H_{back} \vdash A_{inv} \sqsubseteq A_{in} \quad H_{out} \vdash A_{inv} \sqsubseteq A_{out} \end{array}}{\begin{array}{c} \vdash \texttt{check}(C_{in});\texttt{loop}\{\ s;\ \{\ \texttt{if}\ be\ \texttt{break}\ \};\ \texttt{check}(C_{back})\ \} : \\ H_{in}\bullet A_{in} \to H_{out}\bullet A_{out} \end{array}}
$$

[CALL]
$$
\frac{\begin{array}{c} C = \mathit{Checks}(H, H \setminus \mathit{KillSetHistory}(m), A) \\ H' = (H \cup C^{\surd}) \setminus \mathit{KillSetHistory}(m) \\ A = A' \setminus x \setminus \mathit{KillSetAnticipated}(m) \end{array}}{\vdash \texttt{check}(C);\ x = y.m(\overline{z}) : H\bullet A \to H'\bullet A'}
$$

$\boxed{\vdash s}$ [STMT]
$$
\frac{\vdash s : \emptyset\bullet A \to H\bullet\emptyset \quad C = \mathit{Checks}(H,\emptyset)}{\vdash s;\texttt{check}(C)}
$$

$\boxed{\vdash \mathit{meth}}$ [METHOD]
$$
\frac{\vdash s}{\vdash m(\overline{x})\ \{\ s;\texttt{return}\ z\ \}}
$$

$\boxed{\vdash D}$ [CLASS]
$$
\frac{\forall\ \mathit{meth} \in \overline{\mathit{meth}}.\ \vdash \mathit{meth}}{\vdash \texttt{class}\ c\ \{\ \overline{f}\ \overline{\mathit{meth}}\ \}}
$$

$\boxed{\vdash \overline{D}\ \overline{s}}$ [PROGRAM]
$$
\frac{\forall D \in \overline{D}.\ \vdash D \quad \forall i.\ \vdash s_i}{\vdash \overline{D}\ s_1 \| ... \| s_n}
$$

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.

$$
\begin{array}{rl}
H_1 \sqcap H_2 & = \{h \in H_1 \cup H_2 : H_1 \vdash a, H_2 \vdash a\} \\
H_1\bullet A_1 \sqcap H_2\bullet A_2 & = \{a \in A_1 \cup A_2 : H_1\bullet A_1 \vdash a, H_2\bullet A_2 \vdash a\}
\end{array}
$$

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 all of their details.

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

$$
\mathit{Checks}(H, A) = \{\ p\ :\ p^{\lhd} \in H, H \not\vdash p^{\surd}, H\bullet A \not\vdash p^{\diamond}\ \}
$$

to ensure that the path set C contains any path p that was accessed ($p^{\lhd} \in H$) 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 $\_^{\surd}$) because these checks do not cover accesses after the release and (2) all prior accesses (denoted $\_^{\lhd}$) because we are leaving the legitimate check range for them.

[ACQ]: This rule for $\texttt{check}(C);\texttt{acq}(x)$ ensures C contains any path p that was accessed but not checked. The post-history contains the newly checked paths (where $C^{\surd}$ abbreviates $\{p^{\surd} \mid p \in C\}$). 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 to a "fresh" variable not mentioned in the pre-history H,

as the assignment would otherwise invalidate that history information. The [READ] rule adds past access $y.f^{\triangleleft}$ to the post-history. The pre-anticipated paths become $A \backslash x \cup \{y.f^{\diamond}\}$, 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 $y.f^{\triangleleft}$. 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 ← i'` on line 5 in Figure 6(b). The history prior to the renaming contains `a[0..i]`$^{\triangleleft}$ and `a[i]`$^{\triangleleft}$. After renaming, we have `a[0..i']`$^{\triangleleft}$ and `a[i']`$^{\triangleleft}$, enabling us to continue deferring the checks for those accesses.

[WRITE]: This rule for $y.f = x$ adds the access $y.f^{\triangleleft}$ to the post-history, and $y.f^{\diamond}$ 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 $x \notin \mathit{Vars}(e)$ 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 $p^{\diamond} \in A$. 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` be $\{s_1; \texttt{check}(C_1)\}$ $\{s_2; \texttt{check}(C_2)\}$. 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 $H_1' \sqcap H_2'$ describes properties holding after both branches but may leave out accesses that occurred only on one branch. We introduce the following variant of the *Checks* function to compute the unanticipated unchecked past accesses in H that must be checked when H is approximated by H':

$$\mathit{Checks}(H, H', A) = \{\, p : p^{\triangleleft} \in H, H' \not\vdash p^{\triangleleft}, H \not\vdash p^{\checkmark}, H \bullet A \not\vdash p^{\diamond} \,\}$$

Thus, $C_1 = \mathit{Checks}(H_1', H_1' \sqcap H_2', A_{out})$ 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 $H_1 \cup C_1^{\checkmark}$ and $H_2 \cup C_2^{\checkmark}$, and these are merged via $\sqcap$ to yield the final history H_{out}. The anticipated pre-context A_{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:

$$\texttt{check}(C_{in})\texttt{;loop}\{\ s;\ \{\ \texttt{if}\ be\ \texttt{break}\ \};\ \texttt{check}(C_{back})\ \}$$

In this rule, H_{in} and H_{back} are the pre-histories of `check`(C_{in}) and `check`(C_{back}), respectively, and H_{inv} is the loop-invariant history at the loop head. As in [IF], the *Checks* function uses these sets and A_{in}, the anticipated set at the loop head, to compute C_{in} and C_{back}. The side conditions $H_{in} \cup C_{in}^{\checkmark} \sqsupseteq H_{inv}$ and $H_{back} \cup C_{back}^{\checkmark} \sqsupseteq H_{inv}$ ensure that properties in H_{inv} are true on all paths into the loop head.

Note that H_{inv}, H, and H_{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_{inv} that is then refined via a form of predicate abstraction. (See Section 5.) An analogous anticipated set A_{inv} characterizing what is anticipated prior to the loop exit test is used in the computation of A_{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 `check`(C); $x = y.m(\overline{z})$. The function $\mathit{KillSetHistory}(m)$ denotes the set of history properties killed by the side effects of method m, and contains:

$$\begin{array}{ll} \{\ \_^{\triangleleft}\} & \text{if } m \text{ acquires a lock} \\ \{\ \_^{\triangleleft}, \_^{\checkmark}\ \} & \text{if } m \text{ releases a lock} \end{array}$$

The function $\mathit{KillSetAnticipated}(m)$ describes anticipated accesses killed by m. It is $\{\_^{\diamond}\}$ if m acquires a lock and $\emptyset$ otherwise. Our implementation pre-computes $\mathit{KillSetHistory}(m)$ and $\mathit{KillSetAnticipated}(m)$ 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 $P = \overline{D}\ s_1 \| \ldots \| s_n$ via a sequence of states $\Sigma_0 \rightarrow^{a_1} \Sigma_1 \rightarrow^{a_2} \ldots \rightarrow^{a_n} \Sigma_n$, where Σ_0 is an initial state for P and Σ_n is a final terminating state. This evaluation sequence yields a trace $\alpha = a_1.a_2 \ldots a_n$ describing the memory accesses, race checks, and synchronization operations performed by P.

We also define a judgement $\overline{D}; \alpha \Vdash \Sigma$ describing when a run-time state has correct checks in the context of an execution history α. This judgement most notably ensures that, for each thread t, the context $H \bullet A$ for thread t's current program point is consistent with Σ and α. This judgement entails the following: 1) Each expression $be \in H$ is true when evaluated by t in the current state Σ. 2) If $p^{\triangleleft} \in H$ and p denotes a memory l, there is an access to l in by t α with no later release. (Each $p^{\checkmark} \in H$ must have similar check). 3) Each check by t in α is legitimate for a preceding access. 4) Each access to a location l by t in α 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 $p^{\triangleleft}$ is in H or $p^{\diamond}$ 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.*, $\overline{D}; \epsilon \Vdash \Sigma_0$), and we show via a preservation argument that it holds for each subsequent state, including the last, *i.e.*, $\overline{D}; \alpha \Vdash \Sigma_n$. Since each thread in Σ_n has terminated and will perform no subsequent checks or accesses, the rules for ($\Vdash$) imply that α has precise checks. Consequently, the checks in P are address-precise. That is, if $\vdash P$ and P generates a trace α, then for any address l, α 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(`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* $d_1.f_1/f_2/\cdots/f_n$. We also coalesce each group of paths $d_1[b_1..e_1 : k_1], \ldots, d_n[b_n..e_n : k_n]$ 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 coalescible 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 `check(pt.x/y/z)` performs three shadow-location operations. Similarly, an array a of n elements requires n shadow locations, and `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 `check(p.`$\cdots$`/y/`$\cdots$`)` 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.[2] 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 the array all have the form `check(a[0..`n`])`. Similarly `a[i%2]` could be the proxy for each `a[i]` if all checks have the form `check(a[0..`n`:2])` or `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 `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.

[2] While 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`.

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`.

```
acq(lock);
x = a.f; // x = a.f
s = x.g;
y = a.f; // y = a.f
t = y.g;
check (a.f, x.g);
rel(lock);
```

While utlizing 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 [54]. 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 $\{h : h \in History\} \bullet \{a : a \in Anticipated\}$, 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 ϕ-functions as they were handled in REDCARD [25]. Also as in REDCARD, STATICBF tracks extended paths containing multiple field/array references (as in `a[i].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 $p^{\triangleleft}$ and $p^{\diamond}$ is a read or write access, and whether each $p^{\checkmark}$ 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, 56] 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 [23] 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. 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* ($\frac{\#\ Checks}{\#\ Accesses}$) 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:

$$\mathrm{CheckerOverhead} = \mathrm{CheckerTime} - \mathrm{BaseTime}$$

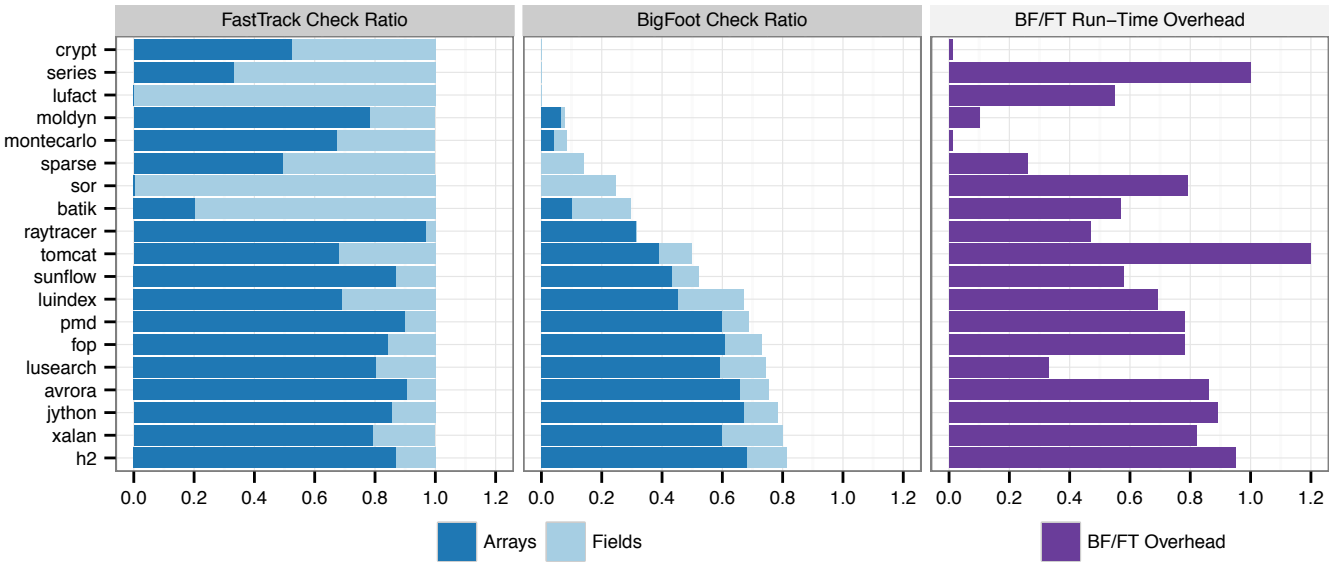

Figure 8. Check Ratio for FASTTRACK and BIGFOOT, and BIGFOOT's overhead relative to the FASTTRACK overhead.

***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 interal 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 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.

Program	StaticBF		Dynamic Analyses										
	Methods Optimized (count)	Time / Method (sec)	BigFoot Check Ratio	Base Time (sec)	Time Overhead (x Base Time)					Time Overhead vs. FT			
					FT	RC	SS	SC	BF	$\left(\frac{RC}{FT}\right)$	$\left(\frac{SS}{FT}\right)$	$\left(\frac{SC}{FT}\right)$	$\left(\frac{BF}{FT}\right)$
crypt	148	0.67	0.00000028	0.39	96.21	62.41	16.87	16.11	0.07	(0.65)	(0.18)	(0.17)	(0.01)
series	144	0.10	0.000042	119.39	0.01	0.01	0.01	0.01	0.01	(1.00)	(1.00)	(1.00)	(1.00)
lufact	168	0.15	0.0022	0.68	71.67	74.31	70.53	74.08	39.53	(1.04)	(0.98)	(1.03)	(0.55)
moldyn	172	0.27	0.077	4.67	27.56	8.73	27.18	6.58	2.72	(0.32)	(0.99)	(0.24)	(0.10)
montecarlo	480	0.05	0.085	2.23	7.38	6.81	2.73	2.02	0.08	(0.92)	(0.37)	(0.27)	(0.01)
sparse	140	0.20	0.14	1.27	26.86	22.57	30.78	27.20	6.68	(0.84)	(1.15)	(1.01)	(0.25)
sor	136	0.24	0.25	0.84	13.37	13.03	15.39	13.85	10.73	(0.97)	(1.15)	(1.04)	(0.80)
batik	20,140	0.16	0.29	1.27	3.96	3.92†	4.07	4.06	2.26	(0.99)	(1.03)	(1.03)	(0.57)
raytracer	308	0.07	0.32	1.84	13.46	6.46	12.64	7.72	6.37	(0.48)	(0.94)	(0.57)	(0.47)
tomcat	27,940	0.12	0.50	0.81	2.05	1.49†	2.22	1.56	2.43	(0.73)	(1.08)	(0.76)	(1.19)
sunflow	3,088	0.21	0.52	1.44	25.94	17.13	26.12	20.50	15.14	(0.66)	(1.01)	(0.79)	(0.58)
luindex	4,728	0.07	0.67	0.54	16.35	15.75	19.00	17.64	11.34	(0.96)	(1.16)	(1.08)	(0.69)
pmd	18,604	0.18	0.69	0.93	3.08	2.98†	2.75	2.65	2.38	(0.97)	(0.89)	(0.86)	(0.77)
fop	24,756	0.15	0.73	0.44	6.51	5.12†	5.65	5.54	5.01	(0.79)	(0.87)	(0.85)	(0.77)
lusearch	3,544	0.07	0.74	0.65	19.45	22.79	7.79	7.24	6.57	(1.17)	(0.40)	(0.37)	(0.34)
avrora	9,936	0.04	0.75	7.82	1.45	1.34†	1.46	1.38	1.24	(0.92)	(1.01)	(0.95)	(0.86)
jython	81,140	0.11	0.78	4.97	9.31	9.32†	8.77	8.58	8.28	(1.0)	(0.94)	(0.92)	(0.89)
xalan	13,420	0.05	0.80	0.86	5.68	5.63†	5.62	5.43	4.64	(0.99)	(0.99)	(0.96)	(0.82)
h2	16,748	0.08	0.81	22.60	3.23	3.08†	3.20	3.23	3.07	(0.95)	(0.99)	(1.00)	(0.95)
Mean		0.16	0.43		7.26	6.00	6.03	5.05	2.47	(0.83)	(0.83)	(0.70)	(0.39)

Table 1. Checker performance. Mean StaticBF time and Check Ratios are arithmetic means. Mean checker overheads for FastTrack (FT), RedCard (RC), SlimState (SS), SlimCard (SC), and BigFoot (BF) are geometric means. The † symbol indicates that RedCard's proxy analysis failed to terminate within 4 hours. We turned off that analysis in those cases.

***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 2. 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, but 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

Program	Base Mem (MB)	Space Overhead				
		$\frac{\text{FT}}{\text{Base}}$	$\left(\frac{\text{RC}}{\text{FT}}\right)$	$\left(\frac{\text{SS}}{\text{FT}}\right)$	$\left(\frac{\text{SC}}{\text{FT}}\right)$	$\left(\frac{\text{BF}}{\text{FT}}\right)$
crypt	193.76	26.27	(0.97)	(0.04)	(0.04)	(0.04)
series	22.01	4.45	(1.02)	(0.58)	(0.59)	(0.57)
lufact	32.15	10.16	(1.00)	(1.10)	(1.10)	(1.11)
moldyn	16.20	5.44	(0.82)	(0.91)	(0.80)	(0.82)
montecarlo	622.83	3.67	(1.00)	(0.30)	(0.30)	(0.30)
sparse	98.11	5.64	(1.01)	(1.44)	(1.05)	(0.79)
sor	32.12	5.11	(1.00)	(1.40)	(1.40)	(2.48)
batik	44.74	3.78	(0.99)	(0.75)	(0.95)	(1.00)
raytracer	16.42	3.67	(0.96)	(0.60)	(0.57)	(0.60)
tomcat	19.59	4.81	(0.99)	(0.98)	(0.99)	(1.14)
sunflow	10.42	9.50	(0.91)	(0.93)	(0.88)	(0.86)
luindex	6.15	16.3	(0.98)	(0.96)	(0.96)	(0.52)
pmd	30.24	6.02	(1.05)	(1.02)	(1.03)	(1.09)
fop	28.07	6.35	(1.00)	(0.98)	(0.97)	(0.99)
lusearch	12.04	7.00	(1.00)	(0.57)	(0.57)	(0.57)
avrora	2.09	15.22	(1.01)	(1.01)	(1.01)	(1.01)
jython	24.06	5.97	(1.03)	(0.96)	(0.96)	(1.02)
xalan	8.20	11.00	(1.00)	(0.84)	(0.84)	(0.82)
h2	259.71	3.90	(1.06)	(1.10)	(1.10)	(0.93)
Geo Mean		6.84	(0.99)	(0.73)	(0.74)	(0.72)

Table 2. Checker space overhead relative to FASTTRACK.

DJIT$^+$ [40], use vector clocks [35], 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, 39, 47, 57].

Several dynamic checkers defer the processing of accesses. RecPlay [43] records all locations accessed within each synchronization-free region and then verifies that concurrent regions access disjoint locations during replay. DRD [17] and ThreadSanitizer [46] similarly buffer accesses but do not infer patterns or compress shadow state. ThreadSanitizer also Similar buffering is also common in transactional memory systems [48]. Other work [49] uses a single shadow location for contiguous memory locations accessed within the same critical sections. However, only the first two critical sections accessing a location are considered, resulting in potential false alarms if later accesses are not correlated.

Many static analyses for identifying races have also been explored, including type-based systems [1, 2, 31], model checking [12, 36, 58] and dataflow analyses [21], as well as whole-program analyses [37, 53]. Many of the mentioned static analyses are unsound by design or unsound in their implementations to reduce the number of spurious warnings (see, *e.g.*, [1, 21]). Their focus on identifying race-free accesses rather than redundant checks also lead to different design choices in terms of precision and scalability.

Gross *et al.* present a global static analysis to improve the precision and performance of a LockSet-based detector [52]. It is primarily designed to identify objects on which no races can occur and requires global aliasing information, as well as a static approximation of the happens-before graph for the whole program. Moreover, their reliance on an imprecise race detector leads their system to both miss races and report spurious warnings. They also do not support arrays. Choi *et al.* present a different global analysis for removing runtime race checks for accesses guaranteed to be race-free [14]. Their analysis eliminates some redundant checks via a simple intra-procedural forward analysis.

Properties related to accesses or checks within release-free spans have been used in other settings. For example, the IFRit race detector uses similar insights in its notion of interference-free regions [20], which were originally designed to facilitate compiler optimizations for race-free programs [19]. The IFRit race detector monitors execution and reports a data race when multiple concurrently executing interference-free regions access the same variable. IFRit prioritizes performance over precision, and so may possibly miss races (but nicely guarantees no false alarms). IFRit uses a static analysis to insert and minimize monitor start/stop calls, which is analogous to BigFoot's check insertion algorithm. BIGFOOT's approach necessitates a more complex static analysis to ensure sufficient precision to perform check motion, and so is at a different point in the design space.

8. Summary

BIGFOOT leverages our theory of precise check placement to substantially improve the efficiency of dynamic data race detection. This work may enable more wide-spread use of data race detectors, and it opens the door for further studies on statically optimizing dynamic concurrency analyses.

One interesting direction is to extend our techniques to compress memory locations across multiple arrays or objects, which could yield further time and space savings. Another important avenue for future work is to improve STATICBF's performance by adapting it to be modular or incremental and by tailoring its data structures and decision procedures to the most common cases encountered in practice.

Acknowledgment

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.

References

[1] Martín Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for Java. *Transactions on Programming Languages and Systems*, 28(2):207–255, 2006.

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

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. 2006.

[4] Alexander Aiken and David Gay. Barrier inference. In *POPL*, pages 243–354, 1998.

[5] Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Brandon Lucia. Valor: efficient, software-only region conflict exceptions. In *OOPSLA*, pages 241–259, 2015.

[6] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, 2006.

[7] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE*, pages 241–250, 2011.

[8] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. Pacer: Proportional detection of data races. In *PLDI*, 2010.

[9] Michael D. Bond, Milind Kulkarni, Man Cao, Minjia Zhang, Meisam Fathi Salmi, Swarnendu Biswas, Aritra Sengupta, and Jipeng Huang. OCTET: capturing and controlling cross-thread dependences efficiently. In *OOPSLA*, pages 693–712, 2013.

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

[11] Cardelli, L. A semantics of multiple inheritance. In *Semantics of Data Types*, Lecture Notes in Computer Science 173, Berlin, 1984. Springer Verlag.

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

[13] Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In *ICFP*, pages 66–77, 2005.

[14] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridhara. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, pages 258–269, 2002.

[15] Mark Christiaens and Koenraad De Bosschere. TRaDe: Data Race Detection for Java. In *International Conference on Computational Science*, pages 761–770, 2001.

[16] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[17] DRD 2014. DRD: a thread error detector. Available at `http://valgrind.org/docs/manual/drd/manual.html`, 2014.

[18] Matthew B. Dwyer and Lori A. Clarke. Data flow analysis for verifying properties of concurrent programs. Technical Report 94-045, Department of Computer Science, University of Massachusetts at Amherst, 1994.

[19] Laura Effinger-Dean, Hans-Juergen Boehm, Dhruva R. Chakrabarti, and Pramod G. Joisha. Extended sequential reasoning for data-race-free programs. In *MSPC*, pages 22–29, 2011.

[20] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-Juergen Boehm. IFRit: interference-free regions for dynamic data-race detection. In *OOPSLA*, pages 467–484, 2012.

[21] Dawson R. Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP*, 2003.

[22] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *OSDI*, pages 151–162, 2010.

[23] Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and precise dynamic race detection. In *PLDI*, pages 121–133, 2009.

[24] Cormac Flanagan and Stephen N. Freund. The RoadRunner dynamic analysis framework for concurrent programs. In *PASTE*, pages 1–8, 2010.

[25] Cormac Flanagan and Stephen N. Freund. RedCard: Redundant check elimination for dynamic race detectors. In *ECOOP*, pages 255–280, 2013.

[26] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *POPL*, pages 191–202, 2002.

[27] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI*, pages 237–247, 1993.

[28] Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA. *Transactions on Programming Languages and Systems*, 17(1):85–122, 1995.

[29] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckle. *The Java Language Specification, Java SE 8 Edition*. 2015.

[30] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.

[31] Dan Grossman. Type-safe multithreading in Cyclone. In *Proceedings of the ACM Workshop on Types in Language Design and Implementation*, pages 13–25, 2003.

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

[33] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[34] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *POPL*, pages 378–391, 2005.

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

[36] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.

[37] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *PLDI*, pages 308–319, 2006.

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

[39] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *PPOPP*, 2003.

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

[41] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.

[42] Dustin Rhodes, Cormac Flanagan, and Stephen N. Freund. BigFoot: Static check placement for dynamic race detection. Technical Report CSTR-201702, Williams College, 2017. Available at `http://www.cs.williams.edu/~freund/papers/bigfoot-tr.pdf`.

[43] Michiel Ronsse and Koenraad De Bosschere. RecPlay: A fully integrated practical record/replay system. *TCS*, 17(2):133–152, 1999.

[44] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *TOCS*, 15(4):391–411, 1997.

[45] Edith Schonberg. On-the-fly detection of access anomalies. In *PLDI*, pages 285–297, 1989.

[46] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, pages 62–71, 2009.

[47] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. Dynamic race detection with LLVM compiler - compile-time instrumentation for ThreadSanitizer. In *RV*, pages 110–114, 2011.

[48] Nir Shavit and Dan Touitou. Software transactional memory. In *ACM Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[49] Young Wn Song and Yann-Hang Lee. Efficient data race detection for C/C++ programs using dynamic granularity. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 679–688, 2014.

[50] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada*, page 13, 1999.

[51] Christoph von Praun and Thomas Gross. Object race detection. In *OOPSLA*, 2001.

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

[53] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Relay: static race detection on millions of lines of code. In *FSE*, pages 205–214, 2007.

[54] WALA. T.J. Watson Libraries for Analysis (WALA). Available at `http://github.com/wala`, 2016.

[55] James R. Wilcox, Parker Finch, Cormac Flanagan, and Stephen N. Freund. Array shadow state compression for precise dynamic race detection. In *ASE*, pages 155–165, 2015.

[56] Michael Wolfe. Beyond induction variables. In *PLDI*, pages 162–174, 1992.

[57] Xinwei Xie and Jingling Xue. Acculock: Accurate and efficient detection of data races. In *CGO*, pages 201–212, 2011.

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

[59] Yuan Yu, Tom Rodeheffer, and Wei Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP*, pages 221–234, 2005.