**BIGFOOT: Static Check Placement for Dynamic Race Detection**

Williams College Technical Report CSTR201702, March 14, 2017

Dustin Rhodes  Cormac Flanagan  
University of California, Santa Cruz, USA  
dustin@soe.ucsc.edu  cormac@cs.ucsc.edu

Stephen N. Freund  
Williams College, USA  
freund@cs.williams.edu

**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, 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
"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. 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. As noted for other systems [5], if such a data race can corrupt the analysis state maintained by the race detector, the data race may go undetected. However, 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 is not amenable to static coalescing.

```
1: acq(lock); {∅•{b.f}}
2: x = b.f; {b.f^<}•{b.f^>}
3: rel(lock); ∅•{b.f^>}
4: y = b.f; {b.f^<}•∅
5: check(b.f); {b.f^<, b.f^>}•∅
6: acq(lock); {b.f^<, b.f^>}•∅
7: z = b.f; {b.f^<, b.f^>}•∅
8: rel(lock); ∅•∅
```

Figure 3. A code fragment with precise checks, and the corresponding BigFoot analysis contexts, Section 3. (All variables are thread-local, and objects thread-shared.)
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 release 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. 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 \( s_1 \parallel s_n \). 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 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 \( \text{null} \), 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.

![Figure 4. Precise and imprecise check placement locations.](image-url)
A path of the form \( p_{b..e} \) describes array accesses, where \( r \) is a \textit{strided} range of the form \( b + ik : b \leq b + ik < e \) to be checked. We use \( b \) and \( b..e \) to abbreviate singleton \( \langle b..(b + 1) : 1 \rangle \) and continuous \( \langle b..e : 1 \rangle \) strided ranges, respectively. We defer distinguishing read checks and write checks until Section 5.

\subsection{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 \( H \cdot A \) for each program point that describes the known \textit{history} properties \( H \) and \textit{anticipated} properties \( A \) at that point:

\[
\text{Context} ::= H \cdot A \quad H \subseteq \text{History} \quad A \subseteq \text{Anticipated}
\]

\[
h \in \text{History} ::= be \mid p^\leq \mid p^\triangleright
\]

\[
a \in \text{Anticipated} ::= p^\triangleright
\]

These properties capture the following notions:

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

\section{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 \( p^\leq \) to the history whenever the code accesses \( p \), and before an acquire it inserts a check for any past access \( p^\leq \) with no past covering check \( p^\triangleright \), 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 \( p^\leq \) 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 \( p^\leq \) 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 \( (p^\triangleright \text{ is in the history}) \), 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 \( p^\triangleright \) 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 \( \{ \cdot \cdot b..f \} \) 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(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 \( i' \leftarrow i \) 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
if (i<0) {  
  \emptyset \bullet \{ b.f^\circ \}
}

1: i = 0;

2: loop {
  \{i = 0\} \bullet \{ b.f^\circ, a[i]^\circ \}
}

3: t = b.f;

\{a[0..i]^\circ, b.f^\circ\} \bullet \{a[i]^\circ, b.f^\circ\}

4: a[i] = t;

\{a[0..i]^\circ, a[i]^\circ, b.f^\circ\} \bullet \emptyset

5: i' = i - 1;

\{a[0..i']^\circ, a[i]^\circ, b.f^\circ\} \bullet \emptyset

6: i = i + 1;

\{i = i' + 1, a[0..i']^\circ, a[i]^\circ, b.f^\circ\} \bullet \emptyset

7: if (\ldots) break;

\{i = i' + 1, a[0..i']^\circ, a[i]^\circ, b.f^\circ\} \bullet \{b.f^\circ, a[i]^\circ\}

8: \}

\{i = i' + 1, a[0..i']^\circ, a[i]^\circ, b.f^\circ\} \bullet \emptyset

9: check(a[0..i], b.f);

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

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 $H_{inv} = \{a[0..i]^\circ\}$. On entry to the loop, $H_{inv}$ holds because 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 $\{i = i' + 1, a[0..i']^\circ, a[i]^\circ, b.f^\circ\}$.

BIGFOOT defers checks until after the loop whenever possible. In this case, the history at the loop exit on line 7 contains $a[0..i']^\circ$ (the invariant rewritten due to the renaming of i to i’ at line 5) and $a[i]^\circ$ (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.

We note that BIGFOOT requires no global analysis to move the checks out of the loop because all variables referenced in the code are local variables that cannot be changed by other methods or other 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 $b.f^\circ$, but the back edge returns to loop head on line 2, where $b.f^\circ$ is not in the history. This would normally necessitate placing a check on $b.f$ inside the loop before the back edge. However, since $b.f^\circ$ is anticipated at the loop head, we can avoid checking $b.f$ inside the loop and instead defer the check until after the loop at line 9.

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.

### 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 $H = \{z[i]^\circ, i = j\}$ then we can safely infer that $H$ entails $z[j]^\circ$, written $H \vdash z[j]^\circ$. Similarly, we introduce anticipated entailment ($H \bullet A \vdash a$), as in $\{i < 10\} \bullet \{x[0..10]^\circ\} \vdash x[0..i]^\circ$. 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:

$$H_1 \sqsubseteq H_2 \iff \forall h \in H_1. \ H_2 \vdash h$$

$$H \vdash A_1 \sqsubseteq A_2 \iff \forall a \in A_1. \ H \bullet A_2 \vdash a$$

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

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

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

1: i = 0;

2: loop {
  \{i = 0\} \bullet \{ b.f^\circ, a[i]^\circ \}
}

3: t = b.f;

\{a[0..i]^\circ, b.f^\circ\} \bullet \{a[i]^\circ, b.f^\circ\}

4: a[i] = t;

\{a[0..i]^\circ, a[i]^\circ, b.f^\circ\} \bullet \emptyset

5: i' = i - 1;

\{a[0..i']^\circ, a[i]^\circ, b.f^\circ\} \bullet \emptyset

6: i = i + 1;

\{i = i' + 1, a[0..i']^\circ, a[i]^\circ, b.f^\circ\} \bullet \emptyset

7: if (\ldots) break;

\{i = i' + 1, a[0..i']^\circ, a[i]^\circ, b.f^\circ\} \bullet \{b.f^\circ, a[i]^\circ\}

8: \}

\{i = i' + 1, a[0..i']^\circ, a[i]^\circ, b.f^\circ\} \bullet \emptyset

9: check(a[0..i], b.f);
\[ s: H \bullet A \rightarrow H' \bullet A' \]

(We assume \( x \notin \text{Vars}(H) \) in the rules modifying \( x \): [Assign], [Rename], [New], [Read], [A-New], [A-Read], and [Call].)

**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 attempt 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 \( \text{check}(C):\text{rel}(x) \) and uses the function

\[ \text{Checks}(H, A) = \{ p : p^\triangledown \in H, H \not\equiv p^\triangledown, H \bullet A \not\equiv p^\triangledown \} \]

\[ \text{Checks}(H, A) = \{ p : p^\triangledown \in H, H \not\equiv p^\triangledown, H \bullet A \not\equiv p^\triangledown \} \]

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

**[ACQ]**: This rule for \( \text{check}(C):\text{acq}(x) \) ensures \( C \) contains any path \( p \) that was accessed but not checked. The pos-history contains the newly checked paths (where \( C^\triangledown \) abbreviates \( \{p^\triangledown \mid p \in C\} \)). 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 \( y.f^\triangledown \) to the post-history. The pre-anticipated paths become \( A \setminus x \cup \{y,f^\triangledown\} \), 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 $y.f^c$. Instead, the renaming operation $x ← y$ 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 $i \leftarrow i'$ on line 5 in Figure 6(b). The history prior to the renaming contains $a[0..i]^c$ and $a[i]^c$. After renaming, we have $a[0..i']^c$ and $a[i']^c$, enabling us to continue deferring the checks for those accesses.

[WRITE]: This rule for $y.f = x$ adds the access $y.f^c$ to the post-history, and $y.f^c$ 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 $p^c \in A$. Since anticipated paths are not closed under this substitution, we remove from the result any anticipated paths that are syntactically ill-formed.

[IF]: Conditional may require checks to be placed at the end of each branch, and so this rule targets the syntax $\text{if} \ b \ \text{then} \ \{c_1\} \ \text{else} \ \{c_2\}$. This rule first computes the post-histories $H_1^c$ and $H_2^c$ and pre-anticipated sets $A_1$ and $A_2$ for $s_1$ and $s_2$. The merged history $H_1^c \cap H_2^c$ 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 $H'$:

$$\text{Checks}(H, H', A) = \{ p : p^c \in H, H' \not\vdash p^c, H \not\vdash p^c, H \bullet A \not\vdash p^c \}$$

Thus, $C_1 = \text{Checks}(H_1^c, H_1^c \cap H_2^c, A_{\text{out}})$ 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 $H_1^c \cup C_1^c$ and $H_2^c \cup C_2^c$, and these are merged via $\cap$ 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:

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

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 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 $H_{\text{in}} \cup C_{\text{in}}^c \supseteq H_{\text{inv}}$ and $H_{\text{back}} \cup C_{\text{back}}^c \supseteq H_{\text{inv}}$ ensure that properties in $H_{\text{inv}}$ are true on all paths into the loop head.

Note that $H_{\text{inv}}$, $H$, 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 KillSetHistory($m$) denotes the set of history properties killed by the side effects of method $m$, and contains:

$$\{-y\} \quad \text{if } m \ \text{acquires a lock}$$
$$\{-y, -, \sqrt{\_}\} \quad \text{if } m \ \text{releases a lock}$$

The function KillSetAnticipated($m$) describes anticipated accesses killed by $m$. It is $\{-\}$ if $m$ acquires a lock and $\emptyset$ otherwise. Our implementation pre-computes KillSetHistory($m$) and 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 The Appendix contains 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 $P = \overrightarrow{D} \ s_1 \ldots \ s_n$ via a sequence of states

$$\Sigma_0 \rightarrow^{a_1} \Sigma_1 \rightarrow^{a_2} \ldots \rightarrow^{a_n} \Sigma_n$$

where $\Sigma_0$ is an initial state for $P$ and $\Sigma_n$ is a final terminating state. This evaluation sequence yields a trace $\alpha = a_1, a_2 \ldots a_n$ describing the memory accesses, race checks, and synchronization operations performed by $P$.

We also define a judgement $D; \alpha \vdash \Sigma$ 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 $H \bullet A$ 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 $p^c \in H$ and $p$ denotes a memory $l$, there is an access to $l$ in by $t$ with no later release. (Each $p^c \in H$ 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 $p^c$ is in $H$ or $p^c$ is in $A$. 8 2017/3/14
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., \( \text{DF; } e \vdash \Sigma_0 \)), and we show via a preservation argument that it holds for each subsequent state, including the last, i.e., \( \text{DF; } \alpha \vdash \Sigma_n \). Since each thread in \( \Sigma_n \) has terminated and will perform no subsequent checks or accesses, the rules for (\( \vdash \)) 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 \( (C) \) statement, we divide the paths in \( C \) into equivalence classes based on the path designator: that is, \( d_1 \cdot f_1 \) and \( d_2 \cdot f_2 \) are in the same class if \( d_1 \cdot f_1 \) and \( d_2 \cdot f_2 \) 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 \( d_1 \cdot f_1, d_2 \cdot f_2, \ldots, d_n \cdot f_n \) sharing equivalent designators to the coalesced field path \( d_1 \cdot f_1 / f_2 / \ldots / f_n \). We also coalesce each group of paths \( d_1 \cdot b_1 : e_1 : k_1, \ldots, d_n \cdot b_n : e_n : k_n \) to one array path \( d_1 \cdot b : e : k \) such that the striped range “\( b : e : k \)” captures the exact same set of indices as the \( n \) original striped 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 striped ranges. This combinatorial approach can be expensive if there are a large number of striped 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 \( \text{check}(pt.x/y/z) \) 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 \( \text{check}(p.x/y) \) 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 the array 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 striped 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 \( y.g \).

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 [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 inter-procedural dataflow analysis. StaticBF then inserts checks into each method using a method-local dataflow analysis.

The initial context for each program point is \( \{ h : h \in \text{History}\} \cdot \{ a : a \in \text{Anticipated}\} \), 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 read or write accesses, but a read check can only cover read accesses. In addition, contexts record whether each \( p^\triangledown \) and \( p^\triangledown \) is a read access or a write access, and whether each \( p^\triangledownd \) is a read check or write check. The analysis rules and coalescing operations are also extended appropriately.

Loop Invariants  StaticBF infers the loop invariant \( H_{\text{inv}} \) 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 \( H_{\text{heuristic}} \) 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 `NullPointerException`, 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`, although a better approach may 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 target 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 $(\# \text{Checks}) / (\# \text{Accesses})$ 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:

$$\text{CheckerOverhead} = \text{CheckerTime} - \text{BaseTime}$$
Comparison to FastTrack  

BigFoot performance is significantly better than the other detectors examined. 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 that have 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 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 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.

**Comparison to SLIMSTATE**  SLIMSTATE 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 SLIMSTATE because integrating the run-time bookkeeping necessary to support static array proxies [25] into SLIMSTATE’s analysis led to worse performance. As a result, SLIMSTATE 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, SLIMSTATE 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. SLIMSTATE’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 SLIMSTATE reduce space overhead by about 26–28% when compared to FASTTrack. These three tools utilize the same dynamic array compression scheme. SLIMSTATE 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
DJIT\textsuperscript{+} [40], use vector clocks [35], which are expensive. FASTTRACK introduced 	extit{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 accessesRecPlay [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 uses a similar approach 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.

**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\textsuperscript{+}\textsuperscript{′}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


We next prove that the check placement algorithm is correct. In particular it inserts checks so that any generated trace has precise checks, and so has a data race if and only if there is a check race detected by DYNAMICBF.

- Appendix ?? formalizes the operational semantics of BFJ.
- Appendix ?? shows that a trace with precise checks has a data race if and only if it has a check race.
- Appendix ?? formalizes a GOODCHECKS judgment that satisfies preservation.
- Appendix ?? shows that the CHECKPLACEMENT algorithm inserts checks satisfying GOODCHECKS.
- Appendix ?? shows that the programs satisfying the GOODCHECKS judgment generate traces with precise checks.
- Appendix ?? shows that correctness of BIGFOOT.
A. Semantics

We specify the operational semantics of BFJ in Figure ?? This semantics evaluates a program by stepping through a sequence of states. Each state $\Sigma$ consists of two components: a heap $S$ and a collection of threads $T$. The heap maps locations to values, where each location $\rho.f$ or $\rho[i]$ combines an address $\rho$ with a field name $f$ or array index $i$. The heap also maps each object address $\rho$ to the thread identifier (or $Tid$) of the thread holding the object’s lock (or $\bot$ if it is not held). The thread set $T$ maps each thread identifier $t \in Tid$ to a thread state $\langle \sigma, s \rangle$ that combines a statement $s$ with a (thread-local) store $\sigma$ mapping variables in $s$ to values.

In the context of a set of definitions $\mathcal{D}$, the relation

$$\mathcal{D} \vdash S \cdot \langle \sigma, s \rangle \longrightarrow^a S' \cdot \langle \sigma', s' \rangle$$

models the effect of a single step by thread $\langle \sigma, s \rangle$ on the heap $S$ and the thread’s local state. The Action $a$ captures the heap operation performed by the step. For example, if thread $t$ accesses location $\rho.f$, $a$ would be $t: \text{acc}(\rho, f)$. The special action $t: \epsilon$ indicates that a step has no heap effect.

Figure ?? defines the evaluation rules for each statement. In these rules, the heap $S[\rho.f := v]$ is identical to $S$ except that it maps the location $\rho.f$ to the value $v$. Similar update operations are used on the other state components. For example, $S[\rho := t]$ updates $S$ to indicate that the lock for the object at location $\rho$ is held by $t$. The term $\sigma(e)$ evaluates an expression $e$ using local store $\sigma$ for the values of variables.

The rule [E-CHKSET] unrolls a check on a set of paths to separate checks on each path. The rule [E-CHKINDEX] checks a strided range of array indices by explicitly checking the first index and generating a new check for the remainder of the strided range. Rule [E-CHKEMPTY] handles empty strided array indices.

To invoke a method $x = y.m(\overline{z})$, we first look up the method $m$ in the program definitions. We then construct a substitution $\theta$ that maps 1) $m$’s local variables, which are fresh names, 2) the parameters $\overline{z'}$ to the arguments $\overline{z}$, 3) the self-reference $\text{this}$ to $y$, and 4) the return variable $r$ to $x$. If $s$ is the method body of $m$, $\theta(s)$ may be inserted into the evaluation context surrounding the call without variable capture. Moreover, the result of the call is placed in $x$, as expected.

The relation $\mathcal{D} \vdash \Sigma \longrightarrow^a \Sigma'$ describes a single step of multithreaded program execution. That rule selects an arbitrary thread $t$ to take a step and updates the global state $\Sigma$ accordingly. As above, $a$ captures the memory or synchronization operation performed by the step. We use the notation $t:\_\_ \longrightarrow^a$ to represent an arbitrary action by thread $t$.

The relation $\mathcal{D} \vdash \Sigma \longrightarrow^\alpha \Sigma'$ denotes the reflexive-transitive closure of $\longrightarrow^a$, where the trace $\alpha$ is a sequence of actions $a_1, a_2, \ldots, a_n$. Given this definition, $\mathcal{D} \vdash \Sigma \longrightarrow^\alpha \Sigma'$ models the arbitrary interleaving of the various threads of a multithreaded program $\mathcal{D}$.

For a program $\mathcal{D} s_1 \parallel \ldots \parallel s_n$, its initial state is $\Sigma_0 = S_0 \cdot T_0$, where

- $S_0$ maps all locations to null and all addresses to $\bot$; and
- $T_0$ maps each thread $t \in 1..n$ to $\langle \sigma, s_t \rangle$, where $\sigma$ assigns a distinct global address to each free variable in $s_{1..n}$. Thus, free variables in $s_{1..n}$ implicitly denote potentially thread-shared objects.
\[ \Sigma \in \text{State} ::= S \cdot T \]
\[ S \in \text{Store} ::= (\text{Location} \rightarrow \text{Value}) \cup (\text{Address} \rightarrow \text{Tid}_\bot) \]
\[ \rho \in \text{Address} \]
\[ t \in \text{Location} ::= \rho.f | \rho[i] \]
\[ u,t \in \text{Tid} ::= 1 | 2 | ... \]
\[ T \in \text{Threads} ::= \text{Tid} \rightarrow \langle \sigma, s \rangle \]
\[ \sigma \in \text{Store} ::= \text{Var} \rightarrow \text{Value} \]
\[ v \in \text{Value} ::= \rho | \text{true} | \text{false} | \text{null} | i | ... \]
\[ i \in \text{Nat} \]

\[ \alpha, \beta \in \text{Trace} ::= \overline{\pi} \]
\[ a,b,c,d \in \text{Action} ::= t: \text{acc}(l) | t: \text{check}(l) | t: \text{acq}(\rho) | t: \text{rel}(\rho) | t: \epsilon \]

\[ \overline{D} \vdash \Sigma \rightarrow^a \Sigma' \]

\[ \overline{D} \vdash S \cdot T[t := \langle \sigma, s \rangle] \rightarrow^a S \cdot T[t := \langle \sigma', s' \rangle] \quad \text{if } \overline{D} \vdash S \cdot \langle \sigma, s \rangle \rightarrow^a S' \cdot \langle \sigma', s' \rangle \text{ and } a = t: \]

\[ \overline{D} \vdash S \cdot \langle \sigma, s \rangle \rightarrow^a S' \cdot \langle \sigma', s' \rangle \]

\[ \begin{array}{ll}
\overline{D} \vdash S \cdot \langle \sigma, \text{check}\{x.f\} \rangle & \rightarrow^t \langle \sigma, \text{skip} \rangle \\
& \quad \text{if } \sigma(x) = \rho \quad \text{[E-ChkField]} \\
\overline{D} \vdash S \cdot \langle \sigma, \text{check}\{p_1, \ldots, p_n\} \rangle & \rightarrow^t \langle \sigma, \text{check}\{p_1: \ldots, \text{check}\{p_n\}\} \rangle \\
& \quad \text{if } \sigma(e_1) = \sigma(e_2) \quad \text{[E-ChkSet]} \\
\overline{D} \vdash S \cdot \langle \sigma, \text{check}\{e_1:e_2:e_3\} \rangle & \rightarrow^t \langle \sigma, \text{check}\{x[e_1:e_2:e_3]\} \rangle \\
& \quad \text{if } \rho = \sigma(x), i = \sigma(e_1), i < \sigma(e_2) \quad \text{[E-ChkIndex]} \\
\overline{D} \vdash S \cdot \langle \sigma, \text{skip}; s \rangle & \rightarrow^a \langle \sigma, s \rangle \\
& \quad \text{[E-Seq]} \\
\overline{D} \vdash S \cdot \langle \sigma, \text{if be } s_1; s_2 \rangle & \rightarrow^t \langle \sigma, s_1 \rangle \\
& \quad \text{if } \sigma(\text{be}) = \text{true} \quad \text{[E-If]} \\
\overline{D} \vdash S \cdot \langle \sigma, \text{if be } s_1; s_2 \rangle & \rightarrow^t \langle \sigma, s_2 \rangle \\
& \quad \text{if } \sigma(\text{be}) = \text{false} \quad \text{[E-If2]} \\
\overline{D} \vdash S \cdot \langle \sigma, L \rangle & \rightarrow^t \langle \sigma, s_1; \text{if be skip } s_2; L \rangle \\
& \quad L = \text{loop } s_1 \{ \text{if be break } }; s_2 \} \quad \text{[E-Loop]} \\
\overline{D} \vdash S \cdot \langle \sigma, x = y.f \rangle & \rightarrow^t \langle \sigma, x := v, \text{skip} \rangle \\
& \quad \text{if } \sigma(y) = \rho \text{ and } S(\rho.f) = v \quad \text{[E-Read]} \\
\overline{D} \vdash S \cdot \langle \sigma, x := x \rangle & \rightarrow^t \langle \sigma, x := v, \text{skip} \rangle \\
& \quad \text{if } \sigma(y) = \rho \text{ and } S(\rho.i) = v \quad \text{[E-Write]} \\
\overline{D} \vdash S \cdot \langle \sigma, y := y.i \rangle & \rightarrow^t \langle \sigma, x := v, \text{skip} \rangle \\
& \quad \text{if } \sigma(y) = \rho \text{ and } S(\rho.i) = v \quad \text{[E-ARead]} \\
\overline{D} \vdash S \cdot \langle \sigma, \text{acq}(x) \rangle & \rightarrow^t \langle \sigma, \text{acq}(\rho) \rangle \\
& \quad \text{if } \sigma(y) = \rho \text{ and } S(\rho.i) = v \quad \text{[E-AWrite]} \\
\overline{D} \vdash S \cdot \langle \sigma, \text{rel}(x) \rangle & \rightarrow^t \langle \sigma, \text{acq}(\rho) \rangle \\
& \quad \text{if } \sigma(y) = \rho \text{ and } S(\rho.i) = v \quad \text{[E-ACQ]} \\
\overline{D} \vdash S \cdot \langle \sigma, \text{rel}(x) \rangle & \rightarrow^t \langle \sigma, \text{rel}(\rho) \rangle \\
& \quad \text{if } \rho \text{ is fresh} \quad \text{[E-Rel]} \\
\overline{D} \vdash S \cdot \langle \sigma, x = \text{new } \epsilon \rangle & \rightarrow^t \langle \sigma, x := \rho, \text{skip} \rangle \\
& \quad \text{if } \rho \text{ is fresh} \quad \text{[E-New]} \\
\overline{D} \vdash S \cdot \langle \sigma, x = \text{new_array } z \rangle & \rightarrow^t \langle \sigma, x := \rho, \text{skip} \rangle \\
& \quad \text{if } \sigma(\epsilon) = v \quad \text{[E-ANew]} \\
\overline{D} \vdash S \cdot \langle \sigma, x = e \rangle & \rightarrow^t \langle \sigma, x := v, \text{skip} \rangle \\
& \quad \text{if } \sigma(\epsilon) = v \quad \text{[E-Assign]} \\
\overline{D} \vdash S \cdot \langle \sigma, x := y \rangle & \rightarrow^t \langle \sigma, x := \sigma(y), \text{skip} \rangle \\
& \quad \text{if } \sigma(\epsilon) = v \quad \text{[E-Rename]} \\
\overline{D} \vdash S \cdot \langle \sigma, x = y.m(\overline{z}) \rangle & \rightarrow^t \langle \sigma, \theta(s) \rangle \\
& \quad \text{if } m(\overline{z}) \{ s; \text{return } r \} \in \overline{D} \quad \text{[E-Call]} \\
\end{array} \]

and \( \theta \) maps \( FV(s) \setminus \{ r \} \) to fresh names
and \( \theta \) maps \( \overline{z}, \text{this}, r \) to \( \overline{z}, y, x \), respectively.
B. Data Races and Check Races

The happens-before relation $\prec_\alpha$ for a trace $\alpha$ is the smallest transitively-closed relation over the operations in $\alpha$ such that the relation $a \prec_\alpha b$ holds whenever $a$ occurs before $b$ in $\alpha$ and one of the following holds:

- Program order: The two operations performed by the same thread.
- Locking: The two operations acquire or release the same lock.

We introduce the following definitions:

- Two operations are concurrent if they are not ordered by happens before.
- Two accesses conflict if they access the same location.
- Two checks conflict if they check the same location.
- A trace has a data race on a location $l$ if it has a pair of conflicting concurrent accesses to $l$.
- A trace has a check race on a location $l$ if it has a pair of conflicting concurrent checks on $l$.

A check $c = t\text{.check}(l)$ covers an access $a = t\text{.acc}(l)$ if:

- $c$ precedes $a$ with no intervening $t\text{.rel}(l)$.
- $c$ succeeds $a$ with no intervening $t\text{.acq}(l)$.

A check $c = t\text{.check}(l)$ is legitimate for an access $a = t\text{.acc}(l)$ if:

- $c$ precedes $a$ with no intervening $t\text{.acq}(l)$.
- $c$ succeeds $a$ with no intervening $t\text{.rel}(l)$.

A trace $\alpha$ has precise checks if each access has a covering check and each check is legitimate for some access.

We start with two technical lemmas that show how the notions of covering and legitimate checks constrain the happens-before relation for a trace.

**Lemma 1.** If a trace $\alpha$ has an access $a$ with a covering check $c$ then for any action $d$ by a different thread in $\alpha$ we have that:

1. $c \prec_\alpha d \Rightarrow a \prec_\alpha d$
2. $d \prec_\alpha c \Rightarrow d \prec_\alpha a$

**Proof.** The check $c$ can become either before or after the access in $\alpha$.

- Case Before:
  $\alpha = \alpha_1.c.\alpha_2.a.\alpha_3$, where $\alpha_2$ has no releases by thread $t$ since check $c$ covers $a$.
  Program order then shows that $d \prec_\alpha c \Rightarrow d \prec_\alpha a$.
  Since $\alpha_2$ does not contain a release by thread $t$, $c \prec_\alpha d \Rightarrow a \prec_\alpha d$.

- Case After:
  $\alpha = \alpha_1.a.\alpha_2.c.\alpha_3$, where $\alpha_2$ has no acquires by $t$ since check $c$ covers $a$.
  By program order $c \prec_\alpha d \Rightarrow a \prec_\alpha d$.
  Since $\alpha_2$ does not contain an acquire by thread $t$, $d \prec_\alpha c \Rightarrow d \prec_\alpha a$.

**Lemma 2.** If a trace $\alpha$ has a check $c$ that is legitimate for an access $a$ then for any action $d$ by a different thread in $\alpha$ we have:

1. $a \prec_\alpha d \Rightarrow c \prec_\alpha d$
2. $d \prec_\alpha a \Rightarrow d \prec_\alpha c$

**Proof.** The proof is similar to the above lemma.

We next show that the notion of covering checks guarantees no missed races (false negatives), and the notion of legitimate checks guarantees no false alarms (false positives).
Lemma 3. Let \( l \) be a location and suppose each access to \( l \) in \( \alpha \) has a covering check. If \( \alpha \) has no check race on \( l \) then \( \alpha \) has no data race on \( l \).

Proof. Let \( t : \text{acc}(l) \) and \( u : \text{acc}(l) \) be two accesses in \( \alpha \) whose covering checks are not racy. Without loss of generality we assume \( t : \text{check}(l) <_{\alpha} u : \text{check}(l) \). By Lemma ??(1), \( t : \text{acc}(l) <_{\alpha} u : \text{check}(l) \), and hence by Lemma ??(2) \( t : \text{acc}(l) <_{\alpha} u : \text{acc}(l) \), so the accesses are race-free.

Lemma 4. Let \( l \) be a location and suppose each check in \( \alpha \) on \( l \) is legitimate for some access. If \( \alpha \) has a check race on \( l \) then \( \alpha \) has a data race on \( l \).

Proof. Suppose \( \alpha \) has two race-free accesses \( t : \text{acc}(l) \) and \( u : \text{acc}(l) \), where \( t : \text{acc}(l) <_{\alpha} u : \text{acc}(l) \). Each access has a covering check \( t : \text{check}(l) \) and \( u : \text{check}(l) \). By Lemma ??(1), \( t : \text{check}(l) <_{\alpha} u : \text{acc}(l) \). By Lemma ??(2), \( t : \text{check}(l) <_{\alpha} u : \text{check}(l) \).

By combining these ideas of covering and legitimate checks, we prove that a trace with precise checks has a check race if and only if the trace has a data race (and therefore running a dynamic race detector with these checks will report a race if and only if there is a data race).

Theorem B.1. Suppose \( \alpha \) has precise checks. Then for all locations \( l \), \( \alpha \) has a check race on \( l \) if and only if \( \alpha \) has a data race on \( l \).

Proof. By the application of Lemma ?? and ??.
C. GOODCHECKS Judgment

Studies of type systems typically separate the problems of type inference and type checking. In our setting, we also separate the problems of inferring where to place checks and verifying that check placement is precise. BtGFOOT’s check placement judgment shown in Figure 7 performs the former; the “good checks” judgment presented in this section performs the latter. We refer to those judgments as CHECKPLACEMENT and GOODCHECKS, respectively.

The GOODCHECKS rules shown in Figure 22 include a subsumption rule [CC-SUB], and so it is not a syntax-directed algorithm like CHECKPLACEMENT; instead it is a mathematical definition designed to satisfy the usual preservation property plus other correctness properties discussed below regarding precise race detection.

The CHECKPLACEMENT algorithm uses both a history context \( H \) and anticipated context \( A \) to represent the forwards and backwards analysis. GOODCHECKS combines these two to form a single context \( \Pi = H \cup A \). We define the entailment relation \( \Pi \vdash h \) from a context \( \Pi = H \cup A \) as simply \( H \vdash h \). The GOODCHECKS rules are defined as follows:

- [CC-Skip], [CC-New], and [CC-ANew] do not change the context and always succeed.
- [CC-Assign] adds a new constraint representing the assignment to the post-context.
- [CC-Chk] adds the checked paths \( (C^\vee) \) to the post context. It only succeeds if each path \( p \) to check has already been accessed. This condition prevents false positives by preventing checking locations which have not yet been accessed. We always delay checks and never bring them forward so a location must have been accessed in order to be checked.
- [CC-Read] removes any anticipated accesses to \( y.f \) and adds an access to \( y.f \). Removing the anticipated access is safe because we are adding in an access to the same location.
- [CC-Write] also removes any anticipated accesses to \( y.f \) and adds in an access to \( y.f \).
- [CC-ARead] and [CC-AWrite] are similar to the above.
- [CC-Seq] allows for the chaining of two statements with the post-context of the first becoming the pre-context of the second.
- [CC-If] checks both the then and the else branches using the pre-context along with the information gained about \( be \). The resulting post-contexts must match and are used for the post-context of the whole expression. Rule [CC-SUB] below can be used to bring the two post-contexts into alignment.
- [CC-Loop] enters the loop with a context of \( \Pi_{inv} \). Statement \( s_1 \) is checked with this context and produces a new context \( \Pi_1 \cup \{\neg be\} \). Statement \( s_2 \) is then checked with \( \Pi_1 \cup \{\neg be\} \) and produces the post context \( \Pi_{inv} \) which can safely check \( s_1 \). Upon exiting the loop \( s_1 \) has run and the \( be \) is true so the resulting post-context is \( \Pi_1 \cup \{be\} \).
- [CC-Acq] does not change the context. However, it does check that all accesses in the context have already been checked (as checking them after acquiring the lock may cause a false negative). It also checks that there are no anticipated accesses in the context as in the backward analysis the anticipated accesses can not be safely moved before an acquire.
- [CC-Rel] removes all history information from the context except boolean expressions. We must remove accesses and checks as the checks made so far are only valid while the lock about to be released is held. For every variable that has been accessed but not checked yet there must be an anticipated access in the post context. This constraint allows the delaying of checks outside of critical sections but only when a later access can be guaranteed.
- [CC-Call] does not modify the context and requires that it not contain any items in the kill set of that method. All accesses which may be killed in the method must be checked before the call. We can use [CC-SUB], shown below, to remove checked access that may be killed but we can not use subsumption to remove unchecked accesses so the analysis remains sound.
- [CC-SUB] allows us to conservatively approximate contexts according to the following context ordering:

\[
(H_1 \cup A_1) \preceq (H_2 \cup A_2) \quad \text{iff} \quad \begin{cases} 
H_1 \supseteq H_2 \\
H_1 \vdash A_1 \subseteq A_2 \\
\forall p^\omega \in H_1, (H_2 \vdash p^\omega) \lor (H_1 \vdash p^\omega) \lor (H_2 \vdash A_2 \vdash p^\omega)
\end{cases}
\]

Run-Time States We introduce the rules shown in Figure 22 to extend the GOODCHECKS relation to run-time states. The judgment \( \sigma; \alpha \vdash h \) determines when a history property \( h \) holds in a given thread-local store \( \sigma \) of thread \( t \). The execution history \( \alpha \) is used to validate past checks and accesses in \( h \). The judgment \( \sigma; \alpha \vdash \Pi \) extends the previous judgment to contexts, and ensures (via C1) each check in \( \alpha \) has a legitimizing previous access in \( \alpha \), and also each access in \( \alpha \) either 1) has a covering check in \( \alpha \) (via A1 or A2) or 2) the context \( \Pi \) records a corresponding past or anticipated access (via A3).
We assume \( x \notin \text{Vars}(\Pi) \) in [CC-New], [CC-Assign], [CC-Read], [CC-ARead], [CC-ANew], [CC-Rename], [CC-Call].

\[
\begin{array}{l}
\vdash s : \Pi \rightarrow \Pi' \\
[CC-Skip] \quad \Pi' = \Pi \cup \{ x = e \} \\
\vdash x = e : \Pi \rightarrow \Pi' \\
[CC-Assign] \quad \Pi' = H[y := x] \cup A \\
\vdash x \Leftarrow y : H \cup A[x := y] \rightarrow \Pi' \\
[CC-Rename] \quad \forall p \in C. \Pi \vdash p^c \\
\vdash \text{check}(C) : \Pi \rightarrow \Pi \cup C^\vee
\end{array}
\]

\[
\begin{array}{l}
\vdash x = \text{new } c : \Pi \rightarrow \Pi \\
[CC-New] \quad \Pi' = \Pi \setminus \{ y.f^\circ \} \cup \{ y.f^c \} \\
\vdash x = y.f : \Pi \rightarrow \Pi' \\
[CC-Read] \quad \Pi' = \Pi \setminus \{ y.f^\circ \} \cup \{ y.f^c \} \\
\vdash y.f = x : \Pi \rightarrow \Pi' \\
[CC-Write] \quad \Pi' = \Pi \setminus \{ y.z^\circ \} \cup \{ y.z^c \} \\
\vdash y.z = x : \Pi \rightarrow \Pi'
\end{array}
\]

\[
\begin{array}{l}
\vdash s_1 : \Pi \rightarrow \Pi_1 \\
[CC-Seq] \quad \vdash s_1 : \Pi \cup \{ e \} \rightarrow \Pi' \\
\vdash s_1 : \Pi \cup \{ \neg e \} \rightarrow \Pi' \\
\vdash \text{if} \ be s_1 \ s_2 : \Pi \rightarrow \Pi' \\
\vdash \Pi \cap \text{KillSetHistory}(m) = \emptyset \\
\vdash \Pi \cap \text{KillSetAnticipated}(m) = \emptyset
\end{array}
\]

\[
\begin{array}{l}
\vdash \text{acq}(x) : \Pi \rightarrow \Pi \\
[CC-Acq] \quad \forall p. p^\circ \notin \Pi \\
\vdash \text{rel}(x) : \Pi \rightarrow \Pi \\
[CC-Rel] \quad \forall p. p^c \notin \Pi \text{ and } p^\vee \notin \Pi
\end{array}
\]

\[
\begin{array}{l}
\vdash \text{m} \{ s; \text{return } z \} \\
[CC-Method] \quad \forall \text{m} \in \text{m}. \vdash \text{m} \\
\vdash \text{class } c \{ f \text{m} \} \\
\vdash s : \emptyset \rightarrow \emptyset \\
\vdash s : \emptyset \\
[CC-Classifier] \quad \forall D \in D. \vdash D \\
\vdash D \Pi
\end{array}
\]

\[
\begin{array}{l}
\vdash s : \Pi_1 \rightarrow \Pi_2 \\
[CC-Sub] \quad \Pi_1 \leq \Pi_1' \\
\vdash s : \Pi_1 \rightarrow \Pi_2 \\
\vdash s : \Pi_1 \rightarrow \Pi_2
\end{array}
\]

\[
\begin{array}{l}
\vdash \text{m} \{ s; \text{return } z \} \\
\vdash \text{m} \{ s; \text{return } z \} \\
\vdash \text{class } c \{ f \text{m} \}
\end{array}
\]

\[
\begin{array}{l}
\forall i. \vdash s_i \\
[CC-Program] \quad \forall D \in D. \vdash D \\
\vdash D \Pi
\end{array}
\]

\textbf{Figure 10. GoodChecks Rules.}

We extend the store \( \sigma \) to map paths (used by the static analysis) to sets of locations (used by the dynamic semantics) as follows:

\[
\begin{align*}
\sigma(x.f) &= \{ \sigma(x).f \} \\
\sigma(x[e_1..e_2:e_3]) &= \{ p[j] : \begin{cases}
p = \sigma(x), \\
j = \sigma(e_1) + i \sigma(e_3), \\
\sigma(e_1) \leq j < \sigma(e_2)
\end{cases} \}
\end{align*}
\]

The rules [CC-Thread] and [CC-State] then extend this well-formedness criteria to threads and states, respectively. Note that [CC-State] does not constraint the heap \( S \) in any way, since we do not reason about heap contents statically. If we were to, for example, include alias assumptions in our core analysis, then we would need to ensure that all of our alias assumptions are true for \( S \).
\[ \forall D \in \mathcal{D}. \models D \quad \forall t \in Tid. \mathcal{D}; \alpha \models t \langle \sigma, s \rangle \]

\[ \forall t \in Tid. \mathcal{D}; \alpha \models S \cdot T \]

\[ \sigma; \alpha \models \Pi \]

\[ \forall h \in \Pi. \sigma; \alpha \models t h \]

Each \( t : \text{check}(l) \) in \( \alpha \) is preceded by \( t : \text{acc}(l) \) with no intervening \( t : \text{rel}(_) \) \hspace{1cm} (C1)

Each \( t : \text{acc}(l) \) in \( \alpha \) is \hspace{1cm} (A1)

preceded by \( t : \text{check}(l) \) with no intervening \( t : \text{rel}(_) \) or

followed by \( t : \text{check}(l) \) with no intervening \( t : \text{acq}(_) \) or

not followed by \( t : \text{acq}(_) \) and \( \exists p. (l \in \sigma(p) \text{ and } (p^\sigma \in \Pi \text{ or } p^\pi \in \Pi)) \) \hspace{1cm} (A3)

\[ \sigma; \alpha \models \Pi \]

\[ \sigma; \alpha \models t h \]

\[ \sigma; \alpha \models t \text{ be} \]

\[ \forall l \in \sigma(p). \left( \frac{\alpha = \alpha_1.t : \text{acc}(l) \cdot \alpha_2 \quad \alpha_2 \text{ does not contain } t : \text{rel}(\rho)}{\sigma; \alpha \models \Pi^\prime} \right) \]

\[ \forall l \in \sigma(p). \left( \frac{\alpha = \alpha_1.t : \text{check}(l) \cdot \alpha_2 \quad \alpha_2 \text{ does not contain } t : \text{rel}(\rho)}{\sigma; \alpha \models \Pi^\prime} \right) \]

**Figure 11.** GOODCHECKS rules for Runtime States.
D. Correctness of CHECKPLACEMENT

Any program satisfying the CHECKPLACEMENT judgment will satisfy the GOODCHECKS judgment.

Lemma 5. If \( \vdash D s_1 \ldots \parallel s_n \) then \( \models D s_1 \ldots \parallel s_n \).

Proof. Follows from Lemma ??.

Lemma 6. If \( \vdash s \) then \( \models s \).

Proof. If \( \vdash s \) then \( s = s' ; \text{check}(C) \) where \( \vdash s' : \emptyset \rightarrow H \bullet \emptyset \) and \( C = \text{Checks}(H, \emptyset) \) from [STMT]. Hence \( \vdash s' : A \rightarrow H \) by Lemma ?? so \( \vdash s : A \rightarrow H \cup C' \) by [CC-SEQ], and so by [CC-SUB] \( \vdash s : \emptyset \rightarrow 0 \), and hence \( \models s \).

Lemma 7. If \( \vdash s : H \bullet A' \rightarrow H' \bullet A \) then \( (\vdash s : H \cup A' \rightarrow H' \cup A) \).

Proof. By induction on the derivation of \( (\vdash s : H \bullet A' \rightarrow H' \bullet A) \) and case analysis on the rule concluding that derivation.

- [ASSIGN] where \( s = (x = e) \): In this case we have
  \[ \vdash x = e : H \bullet A[x := e] \rightarrow H \cup \{x = e\} \bullet A \]
  where \( x \not\in \text{Vars}(e, H) \). Rule [CC-ASSIGN] gives us
  \[ \models x = e : H \cup A[x := e] \rightarrow H \cup \{x = e\} \cup A[x := e] \]
  Finally, \( H \cup A[x := e] \cup \{x = e\} \leq H \cup \{x = e\} \cup A \) by our assumption about entailment, so by [CC-SUB]
  \[ \models x = e : H \cup A[x := e] \rightarrow H \cup \{x = e\} \cup A \]
  as required.

- [RENAME] where \( s = (x \leftarrow y) \): In this case we have
  \[ \vdash x \leftarrow y : H \bullet A[x := y] \rightarrow H[y := x] \bullet A \]
  where \( x \not\in \text{Vars}(H) \). Rule [CC-ASSIGN] gives us the desired
  \[ \models x \leftarrow y : H \cup A[x := y] \rightarrow H[y := x] \cup A \]

- [WRITE] where \( s = (y.f = x) \): In this case we have
  \[ \vdash y.f = x : H \bullet A \cup \{y.f^\circ\} \rightarrow H \cup \{y.f^\circ\} \bullet A \]
  Rule [CC-WRITE] gives us
  \[ \models y.f = x : H \cup A \cup \{y.f^\circ\} \rightarrow H \cup A \cup \{y.f^\circ\} \]

- [READ] where \( s = (x = y.f) \): In this case we have
  \[ \vdash x = y.f : H \bullet A \setminus x \cup \{y.f^\circ\} \rightarrow H \cup \{y.f^\circ\} \bullet A \]
  Rule [CC-READ] gives us
  \[ \models x = y.f : H \cup A \setminus x \cup \{y.f^\circ\} \rightarrow H \cup \{y.f^\circ\} \cup A \setminus y.f^\circ \]
  Finally, \( H \cup \{y.f^\circ\} \cup A \setminus y.f^\circ \leq H \cup \{y.f^\circ\} \cup A \) so by [CC-SUB] we reach our desired
  \[ \models x = y.f : H \cup A \setminus x \cup \{y.f^\circ\} \rightarrow H \cup \{y.f^\circ\} \cup A \]
• [SKIP] where \( s = \text{skip} \): Skip does not modify or place restrictions on the context in either GOODCHECKS or CHECKPLACEMENT and so is trivial.

• [NEW] where \( s = (x = \text{new } c) \): In this case we have
\[
\vdash x = \text{new } c : H \cdot A \xrightarrow{x} H \cdot A
\]
where \( x \notin \text{Vars}(H) \). Rule [CC-NEW] gives us
\[
\Vdash x = \text{new } c : H \cup A \xrightarrow{x} H \cup A \setminus x
\]
Finally, \( H \cup A \setminus x \subseteq H \cup A \) so by [CC-SUB]
\[
\Vdash x = \text{new } c : H \cup A \setminus x \rightarrow H \cup A
\]

• [A-NEW], [A-WRITE], and [A-READ] follow the same proofs as [NEW], [WRITE], and [READ].

• [ACQ] where \( s = \text{check}(C) ; \text{acq}(x) \): In this case we have
\[
\vdash \text{check}(C) ; \text{acq}(x) : H \cdot \emptyset \rightarrow H \cup C^\vee \cdot A
\]
where \( C = \text{Checks}(H, \emptyset) \). Rule [CC-CHK] gives us
\[
\Vdash \text{check}(C) : H \rightarrow H \cup C^\vee
\]
and requires that \( \forall p \in C, H \vdash p^\triangleleft \) which holds by the construction of \( C \). Next [CC-Acq] gives us
\[
\Vdash \text{acq}(x) : H \cup C^\vee \rightarrow H \cup C^\vee
\]
and requires that \( \forall p, p^\triangleleft \in H \cup C^\vee \), which is trivially true, and \( \forall p, p^\triangleleft \in H \cup C^\vee \Rightarrow H \cup C^\vee \vdash p^\vee \) which is true by the construction of \( C \). Finally, \( H \cup C^\vee \subseteq H \cup C^\vee \cup A \) and so by [CC-SUB] and [CC-SEQ] we have the desired
\[
\Vdash \text{check}(C) ; \text{acq}(x) : H \rightarrow H \cup C^\vee \cup A
\]

• [REL] where \( s = \text{check}(C) ; \text{rel}(x) \): In this case we have
\[
\vdash \text{check}(C) ; \text{rel}(x) : H \cdot A \rightarrow H \setminus \{ _\vee, _\triangleleft \} \cdot A
\]
where \( C = \text{Checks}(H, A) \). GOODCHECKS gives us
\[
\Vdash \text{check}(C) : H \cup A \rightarrow H \cup A \cup C^\vee
\]
and requires that \( \forall p \in C, H \vdash p^\triangleleft \) which it does by the construction of \( C \).
\( H \cup A \cup C^\vee \subseteq H \setminus \{ _\vee, _\triangleleft \} \cup A \) because we will never remove an access \( p^\triangleleft \) where \( p^\vee \notin H \cup C^\vee \). Thus by [CC-SUB] we have that
\[
\Vdash \text{check}(C) : H \cup A \rightarrow H \setminus \{ _\vee, _\triangleleft \} \cup A
\]
and by [CC-REL] and the fact that \( \forall p, p^\triangleleft \in H \setminus \{ _\vee, _\triangleleft \} \) and \( p^\vee \notin H \setminus \{ _\vee, _\triangleleft \} \) we have
\[
\Vdash \text{rel}(x) : H \setminus \{ _\vee, _\triangleleft \} \cup A \rightarrow H \setminus \{ _\vee, _\triangleleft \} \cup A
\]
Finally, by [CC-SEQ] we have the desired
\[
\Vdash \text{check}(C) ; \text{rel}(x) : H \cup A \rightarrow H \setminus \{ _\vee, _\triangleleft \} \cup A
\]
• [Ir] where \( s = \text{if } b \text{ e} (s_1; \text{check}(C_1)) (s_2; \text{check}(C_2)) \): In this case we have

\[
\vdash \text{if be} (s_1; \text{check}(C_1)) (s_2; \text{check}(C_2)) : H_i \bullet A_i \rightarrow H_o \bullet A_o
\]

\[
\vdash s_1 : H_i \cup \{b\} \bullet A_i \rightarrow H_o' \bullet A_o
\]

\[
\vdash s_2 : H_i \cup \{-b\} \bullet A_i \rightarrow H_o' \bullet A_o
\]

\[
C_1 = \text{Checks}(H_1, H_1' \cap H_2, A_o)
\]

\[
C_2 = \text{Checks}(H_2, H_2' \cap H_2, A_o)
\]

\[
A_{in} = H_1 \bullet A_1 \cap H_2 \bullet A_2
\]

\[
H_{out} = (H_1' \cup C_1') \cap (H_2' \cup C_2')
\]

In order to conclude

\[
\vdash \text{if be} (s_1; \text{check}(C_1)) (s_2; \text{check}(C_2)) : H_i \cup A_i \rightarrow H_o \cup A_o
\]

We need to show that

\[
\vdash s_1; \text{check}(C_1) : H_i \cup A_i \cup \{b\} \rightarrow H_o \cup A_o
\]

\[
\vdash s_2; \text{check}(C_2) : H_i \cup A_i \cup \{-b\} \rightarrow H_o \cup A_o
\]

By induction, [CC-Chk], and [CC-Seq] we have

\[
\vdash s_1; \text{check}(C_1) : H_i \cup A_i \cup \{b\} \rightarrow H_1' \cup C_1' \cup A_o
\]

\[
\vdash s_2; \text{check}(C_2) : H_i \cup A_i \cup \{-b\} \rightarrow H_2' \cup C_2' \cup A_o
\]

Finally by [CC-SUB]

\[
\vdash s_1; \text{check}(C_1) : H_i \cup A_i \cup \{b\} \rightarrow H_o \cup A_o
\]

\[
\vdash s_2; \text{check}(C_2) : H_i \cup A_i \cup \{-b\} \rightarrow H_o \cup A_o
\]

• [Seq] where \( s = s_1; s_2 \): This case holds trivial by induction.

• [Call] where \( s = \text{check}(C); x = y.m(\pi) \): In this case we have

\[
\vdash \text{check}(C); x = y.m(\pi) : H \bullet A \rightarrow H' \bullet A'
\]

such that

\[
C = \text{Checks}(H, H \setminus \text{KillSetHistory}(m), A)
\]

\[
\{p : p^? \in H, H \setminus \text{KillSetHistory}(m) \not\vdash p^\#, H \bullet A \not\vdash p^\circ\}
\]

\[
H' = (H \cup C^\vee) \setminus \text{KillSetHistory}(m)
\]

\[
A = A' \setminus x \setminus \text{KillSetAnticipated}(m)
\]

By [CC-Check]

\[
\vdash \text{check}(C) : H \cup A \rightarrow H \cup C^\vee \cup A
\]

By [CC-Call]

\[
\vdash x = y.m(\pi) : H' \cup A \rightarrow H' \cup A
\]

Also \( H' \cup A \preceq H' \cup A' \).

Finally, we need to show

\[
H \cup C^\vee \cup A \preceq H' \cup A = (H \cup C^\vee) \setminus \text{KillSetHistory}(m) \cup A
\]

and in particular that

\[
\forall p^\# \in H. H' \vdash p^\# \text{ or } H' \bullet A \vdash p^\circ
\]
If $C \not\vdash p^\circ$ then $H \setminus \text{KillSetHistory}(m) \vdash p^\circ$ or $H \cup C \vdash p^\circ$ or $H' \bullet A \vdash p^\circ$ and so the desired context ordering follows. Hence
\[
\vdash \text{check}(C) : H \cup A \rightarrow H \cup C \cup A
\]
\[
H \cup C \cup A \preceq H' \cup A
\]
\[
\vdash x = y.m(z) : H' \cup A \rightarrow H' \cup A
\]
\[
H' \cup A \preceq H' \cup A'
\]

• [LOOP] where $s = \text{check}(C_{in})$; loop\{ $s_1$: if be break; $\text{check}(C_{back})$ \}: In this case we have the following:
\[
\vdash s : H_{in} \bullet A_{in} \rightarrow H_{out} \bullet A_{inv}
\]
\[
\vdash s_1 : H_{inv} \bullet A_{in} \rightarrow H \bullet A_{inv}
\]
\[
H_{back} = H \cup \{ \neg \text{be} \}
\]
\[
H_{out} = H \cup \{ \text{be} \}
\]
\[
C_{in} = \text{Checks}(H_{in}, H_{inv}, A_{in})
\]
\[
H_{in} \cup C_{in}^\circ \preceq H_{inv}
\]
\[
C_{back} = \text{Checks}(H_{back}, H_{inv}, A_{in})
\]
\[
H_{back} \cup C_{back} \preceq H_{inv}
\]

By induction,
\[
\vdash s_1 : H_{inv} \cup A_{in} \rightarrow H \cup A_{inv}
\]

Also, by [CC-Chk],
\[
\vdash \text{check}(C_{back}) : H_{back} \cup A_{inv} \rightarrow H_{back} \cup A_{inv} \cup C_{back}^\circ
\]

Also, $H_{back} \cup A_{inv} \cup C_{back}^\circ \preceq H_{inv} \cup A_{in}$. Hence by [CC-Loop]
\[
\vdash \text{check}(C_{in})$; loop\{ $s_1$: if be break; $\text{check}(C_{back})$ \}: $H_{inv} \cup A_{in} \rightarrow H_{out} \cup A_{inv}$

This case concludes via [CC-Seq] and [CC-Sub] based on:
\[
H_{out} \cup A_{inv} \preceq H_{out} \cup A_{out}
\]
\[
\vdash \text{check}(C_{in}) : H_{in} \cup A_{in} \rightarrow H_{in} \cup A_{in} \cup C_{in}^\circ
\]
\[
H_{in} \cup A_{in} \cup C_{in}^\circ \preceq H_{inv} \cup A_{in}
\]

Assumption 1 (Entailment Assumptions). We rely on the following assumptions about the entailment relationship.

1. $H \bullet A \vdash p^\circ$ is monotonic in $H$ and $A$.
2. $H \vdash h$ is monotonic in $H$.
3. $\{x = e\} \bullet \{p^\circ\} \vdash p[x := e]^\circ$
4. $\{x = e\} \bullet \{p[x := e]^\circ\} \vdash p^\circ$
5. $\{h\} \vdash h$
6. $H \bullet A \vdash p^\circ$ only depends on boolean expressions in $H$
7. $\{x_{e_1}^\circ, x[(e_1 + e_3) .. e_2 : e_3]^\circ\} \vdash x[e_1 .. e_2 : e_3]^\circ$

28 2017/8/14
E. Correctness of GOODCHECKS

We now show that if a program has passed GOODCHECKS then the program generates traces that have precise checks.

**Definition E.1.** A state $\Sigma$ is terminated if all threads are skipt.

**Theorem E.1 (Correctness of GOODCHECKS).** If $\overline{D}; \epsilon \vdash \Sigma_0$ and $\overline{D} \vdash \Sigma_0 \rightarrow^\alpha \Sigma'$ and $\Sigma'$ is terminated then $\alpha$ has precise checks.

**Proof.** By the Preservation Theorem, we have that $\overline{D}; \alpha \vdash \Sigma'$. Pick any thread $t$, and let $\langle \sigma, \text{skip} \rangle = T(t)$. By [CC-STATE] and [CC-THREAD], $\overline{D}; \alpha \vdash_t (\sigma, \text{skip})$ where $\sigma;\alpha \vdash_t \Pi$ and $\vdash \text{skip} : \Pi \rightarrow \emptyset$ for some $\Pi \leq \emptyset$. Hence [CC-CONTEXT] implies that (from C1) each check by $t$ has a preceding legitimizing access. Moreover, since $\Pi \leq \emptyset$, from the definition of $\leq$ we know that $\Pi$ has no anticipated access, and $\forall p^\Sigma \in \Pi$ we have that $\Pi \vdash p^\Sigma$.

Consider any access $t: \text{acc}(l)$ in $\alpha$, which must satisfy one of the antecedents A1, A2, A3 in [CC-CONTEXT]. If the access satisfies A1 or A2, then it clearly has a covering check. If the access satisfies A3, then, since $\Pi$ has no anticipated accesses, $\exists p. \sigma(p) = l$ and $p^\Sigma \in \Pi$. Then, from above, $\Pi \vdash p^\Sigma$, and so by [CC-PASTCHECK] $\alpha$ contains a check covering the access. Hence, $\alpha$ has precise checks. \hfill $\Box$

In order to prove the above induction we must prove that evaluation preserves well-formed states.

**Theorem E.2 (Preservation).** If $\overline{D}; \alpha \vdash \Sigma$ and $\Sigma \rightarrow^\alpha \Sigma'$ then $\overline{D}; (\alpha.a) \vdash \Sigma'$.

**Proof.** Suppose the action $a$ is performed by thread $t$. From the rule [CC-STATE] and the definition of our transition relation, we have:

$$\vdash D \quad \forall D \in \overline{D}$$
$$\overline{D}; \alpha \vdash_t T(i) \quad \forall i \in \text{Tid}$$
$$\Sigma = S \cdot T[i := \langle \sigma, s \rangle]$$
$$\Sigma' = S' \cdot T[i := \langle \sigma', s' \rangle]$$
$$\overline{D} \vdash S \cdot (\sigma, s) \rightarrow^a S' \cdot (\sigma', s')$$

In addition, for any thread $T(i) = \langle \sigma_i, s_i \rangle$, rule [CC-THREAD] requires that there is a $\Pi_i$ such that:

$$\sigma_i;\alpha \vdash_t \Pi_i$$
$$\vdash s_i ; \Pi_i \rightarrow \emptyset$$

If $i \neq t$, then an inspection of [CC-CONTEXT] shows that $\sigma_i;\alpha.a \vdash_t \Pi_i$, and hence $\overline{D}; (\alpha.a) \vdash_t T(i)$. For thread $t$, from Lemma ?? below we have that there exists $\Pi_3$ such that:

$$\vdash s' ; \Pi_3 \rightarrow \emptyset$$
$$\sigma';\alpha.a \vdash_t \Pi_3$$

Hence $\overline{D}; (\alpha.a) \vdash_t \langle \sigma', s' \rangle$. Finally, $\overline{D}; (\alpha.a) \vdash \Sigma'$ then follows by rule [CC-STATE]. \hfill $\Box$

To prove the above we must prove preservation for an individual thread step. Given a well formed thread state, if we take one step of evaluation the thread state remains well-formed.

**Lemma 8 (Preservation for Statements).** If $\forall D \in \overline{D}. \vdash D$ and $a$ is an action by thread $t$ and

$$\vdash s : \Pi_1 \rightarrow \Pi_2$$
$$\sigma;\alpha \vdash_t \Pi_1$$
$$\overline{D} \vdash S \cdot (\sigma, s) \rightarrow^a S' \cdot (\sigma', s')$$

then there exist $\Pi_3$ such that:

$$\vdash s' : \Pi_3 \rightarrow \Pi_2$$
$$\sigma' ; (\alpha.a) \vdash_t \Pi_3$$

**Proof.** By induction on the derivation of $\vdash s : \Pi_1 \rightarrow \Pi_2$ and case analysis on the rule used to conclude that derivation.

- [CC-If] where $s = \text{if } b \text{ then } s_1 \text{ else } s_2$: There are two cases:
  - if $\sigma(b) = \text{true}$:
    - $s' = s_1$, $\sigma' = \sigma, a = t : \epsilon$
    - Via Evaluation
    - Let $\Pi_3 = \Pi_1 \cup \\{b\}$
    - $\sigma;\alpha \vdash_t \Pi_1$
    - Given
    - Need to show $\sigma;\alpha \vdash_t \Pi_1 \cup \{b\}$
    - as $\sigma(b) = \text{true}$
    - Need to show $\vdash s_1 : \Pi_1 \cup \{b\} \rightarrow \Pi_2$
    - Shown via [CC-If]
• If \( \sigma(be) = \text{false} \):
  The false case is similar.

• [CC-Rel] where \( s = \text{rel}(x) \): In this case,
  \( s = \text{rel}(x) \)
  \( \Pi_2 = \Pi_1 \)
  \( \forall p. p^\sigma \notin \Pi_1, p^{\sigma'} \notin \Pi_1 \)
  \( s' = \text{skip} \)
  \( a = t:\text{rel}(p) \)
  \( \sigma' = \sigma \)

Let \( \Pi_3 = \Pi_1 \). We have \( \models s' : \Pi_1 \rightarrow \Pi_1 \) via [CC-Skip].
Since \( \Pi_1 \) does not contain prior accesses or checks, it only contains boolean expressions, and so \( \forall h \in \Pi_1 \) from \( \sigma; \alpha \models_t h \) we can conclude \( \sigma; \alpha.a \models_t h \). Also, properties C1, A1, A2, A3 are not invalidated by adding \( a \) to \( \alpha \) so we conclude \( \sigma; \alpha.a \models_t \Pi_1 \) as required.

• [CC-Acq] where \( s = \text{acq}(x) \): In this case,
  \( \Pi_2 = \Pi_1 \) Via [CC-Acq]
  \( \forall p. p^\sigma \notin \Pi_1 \) Via [CC-Acq]
  \( \forall p. p^\sigma \in \Pi_1 \Rightarrow \Pi_1 \models p^{\sigma'} \) Via [CC-Acq]
  \( a = t:\text{acq}(p) \) Via Evaluation
  \( s' = \text{skip}, \sigma' = \sigma \) Via Evaluation

Let \( \Pi_3 = \Pi_2 \)
\( \models s' : \Pi_2 \rightarrow \Pi_2 \) Shown via [CC-Skip]
We must show \( \sigma; (\alpha.a) \models_t \Pi_1 \).
For all \( h \in \Pi_1 \), we have \( \sigma; (\alpha) \models_t h \) and hence \( \sigma; (\alpha.a) \models_t h \) as \( a \) is not a release.
Since \( \Pi_1 \) does not contain any anticipated access, the requirements C1, A2, A2, and A3 on \( \alpha \) also hold for \( \alpha.a \). Hence \( \sigma; (\alpha.a) \models_t \Pi_1 \).

• [CC-Chk] where \( s = \text{check}(C) \)
  There are four evaluation rules for \( s \).

  • [E-CheckField] where \( s = \text{check}(\{x.f\}) \).
    In this case,
    \( s' = \text{skip}, \sigma' = \sigma \) Via Evaluation
    \( a = t:\text{check}(\sigma(x).f) \) Via Evaluation
    \( \Pi_1 \models x.f^\sigma \) Via [CC-Chk]
    \( \Pi_2 = \Pi_1 \cup \{x.f^{\sigma'}\} \) Via [CC-Chk]
    Let \( \Pi_3 = \Pi_2 \)
    \( \models s' : \Pi_2 \rightarrow \Pi_2 \) Via [CC-Skip]
    It remains to show \( \sigma; \alpha.a \models_t \Pi_2 \).
    Clearly \( \sigma; \alpha.a \models_t x.f^\sigma \) and so \( \forall h \in \Pi_2 \). \( \sigma; \alpha.a \models_t h \).
    Since we are adding \( t:\text{check}(\sigma(x).f) \) to \( \alpha \), we need to show by C1 there was a \( t:\text{acc}(\sigma(x).f) \) in \( \alpha \) with no later release, which is already guaranteed by \( \sigma; \alpha \models_t x.f^\sigma \).
    Hence we conclude \( \sigma; \alpha.a \models_t \Pi_2 \).

  • [E-CheckSet] where \( C = \{p_1, \ldots, p_n\} \).
    In this case,
    \( s' = \text{check}(\{p_1\}) \ldots \text{check}(\{p_n\}) \) Via Evaluation
    \( \sigma' = \sigma \) Via Evaluation
    \( a = t:\epsilon \) Via Evaluation
    Let \( \Pi_3 = \Pi_1 \)
    From \( \models s : \Pi_1 \rightarrow \Pi_2 \) we clearly have \( \models s' : \Pi_1 \rightarrow \Pi_2 \) and \( \sigma; \alpha.a \models_t \Pi_1 \).

  • [E-CheckEmpty] where \( C = x[e_1 \ldots e_2; e_3] \) and \( \sigma(e_1) \geq \sigma(e_2) \).
    We have
\(s' = \text{skip}\) & Via Evaluation
\(s' = \sigma\) & Via Evaluation
\(a = t: \epsilon\) & Via Evaluation

\(\Pi_2 = \Pi_1 \cup \{x[e_1..e_2:e_3]\}\) & Via [CC-CHECK]
Let \(\Pi_3 = \Pi_2\) & Via [CC-SKIP]

We need to show \(\sigma; a. a \vdash \Pi_2\), which reduces to showing \(\sigma; a. a \vdash x[e_1..e_2:e_3]\), which holds via [CC-PAST-CHECK] as \(\sigma(x[e_1..e_2:e_3])\) is empty.

\(\text{• [E-CHKINDEX]}\) where \(C = \{x[e_1..e_2:e_3]\}\).

We have
\(\rho = \sigma(x)\) & Via Evaluation
\(i = \sigma(e_1)\) & Via Evaluation
\(i < \sigma(e_2)\) & Via Evaluation
\(s' = \text{check}\{x[(e_1 + e_3)..e_2:e_3]\}\) & Via Evaluation
\(\sigma' = \sigma\) & Via Evaluation
\(a = t: \text{check}(\rho[i])\) & Via Evaluation

\(\Pi_2 = \Pi_1 \cup \{x[e_1..e_2:e_3]\}\) & Via [CC-CHECK]
Let \(\Pi_3 = \Pi_1 \cup \{x[e_1]\}\)
Clearly \(\sigma; a. a \vdash x[e_1]\).

Since we are adding check \(a\) to the trace, we need to show by C1 there was a corresponding access \(t: \text{acc} (\rho[i])\) in \(\alpha\) with no later release, which is already guaranteed by \(\sigma; a. a \vdash x[e_1..e_2:e_3]\).

Hence we conclude \(\sigma; a. a \vdash \Pi_2\).

Also, we have \(\vdash s' : \Pi_1 \cup \{x[e_1]\} \rightarrow \Pi_1 \cup \{x[e_1]\} \cup \{x[e_1 + e_3..e_2:e_3]\}\) via [CC-CHECK], which via [CC-SUB] gives \(\vdash s' : \Pi_3 \rightarrow \Pi_2\) as required.

\(\text{• [CC-CALL]}\) where \(s = (x = y.m(\pi))\): In this case,
\(\mathcal{D} \vdash m(\pi)\{s_m; \text{return } r\}\) & Via [CC-CALL]

\(x \notin \text{Vars}(\Pi_1, e)\) & Via Evaluation
\(s' = \theta(s_m)\) & Via Evaluation
\(\theta = \{z' := z, \text{this} := y, r := x\}\) & Via [CC-CALL]
\(\Pi \cap \text{KillSetHistory}(m) = \emptyset\) & Via [CC-CALL]
\(\Pi \cap \text{KillSetAnticipated}(m) = \emptyset\) & Via [CC-CALL]
\(\sigma' = \sigma, a = t: \epsilon\) & Via Evaluation

Let \(\Pi_3 = \Pi_1 = \Pi_2 = \Pi\)
Need to show \(\sigma; a. a \vdash \Pi\) & Given
Need to show \(\vdash s_m : \Pi \rightarrow \Pi\)
\(\vdash \theta(s_m) : \emptyset \rightarrow \emptyset\) & Via Lemma ??
\(\vdash s_m : \emptyset \cup \Pi \rightarrow \emptyset \cup \Pi\) & Via Lemma ??

\(\text{• [CC-LOOP]}\) where \(s = L:\)
\(L = \text{loop}\{s_1; \{\text{if be break }\}; s_2\}\)

\(s' = s_1; \text{if be skip } s_2; L\) & Via Evaluation
\(\sigma' = \sigma, a = t: \epsilon\) & Via Evaluation
\(\Pi' = \Pi_2 \setminus \{be\}\) & Via [CC-LOOP]
\(\vdash s_1 : \Pi_1 \rightarrow \Pi'\) & Via [CC-LOOP]
\(\vdash s_2 : \Pi' \cup \{\neg be\} \rightarrow \Pi_1\) & Via [CC-LOOP]
Let \(\Pi_3 = \Pi_1\)
Need to show \(\sigma; a. a \vdash \Pi_1\) & Given
\(\Pi_2 = \Pi_1 \cup \{be\}\) & Via [CC-LOOP]
Need to show \(\vdash s' : \Pi_1 \rightarrow \Pi' \cup \{be\}\)
\(\vdash L : \Pi_1 \rightarrow \Pi' \cup \{be\}\) & Given
\(\vdash s_2 : L : \Pi' \cup \{\neg be\} \rightarrow \Pi' \cup \{be\}\) & Via [CC-SEQ]
\(\vdash \text{skip} : \Pi' \cup \{be\} \rightarrow \Pi' \cup \{be\}\) & Via [CC-SKIP]
\(\vdash \text{if be skip } (s_2; L) : \Pi' \rightarrow \Pi' \cup \{\neg be\}\) & Via [CC-SEQ] and [CC-SUB]
\(\vdash s' : \Pi_1 \rightarrow \Pi' \cup \{be\}\) & Via [CC-SEQ]
• [CC-Seq] where \( s = s_1; s_2 \): There are two cases:
  
  - If \( s_1 = \text{skip} \):
    \[
    S' = s_2, \sigma' = \sigma, a = t; e \quad \text{Via Evaluation}
    \]
    Let \( \Pi_3 = \Pi_1 \)
    Need to show \( \models s_2 : \Pi_1 \rightarrow \Pi_2 \quad \text{Shown via [CC-Seq]} \)
    Need to show \( \sigma; \alpha \models t, \Pi_1 \) \quad \text{Given}

  - If \( s_1 \neq \text{skip} \):
    \[
    D \models S \cdot (\sigma, s_1) \rightarrow^a S' \cdot (\sigma', s'_1) \quad \text{Via Evaluation}
    \]
    \[
    \models s_1 : \Pi_1 \rightarrow \Pi' \quad \text{Via [CC-Seq]}
    \]
    \[
    \models s_2 : \Pi' \rightarrow \Pi_2 \quad \text{Via [CC-Seq]}
    \]
    \[
    \sigma; \alpha \models t, \Pi_1 \quad \text{Given}
    \]
    \[
    \models s'_1 : \Pi_4 \rightarrow \Pi' \quad \text{Via inductive hypothesis}
    \]
    \[
    \sigma; (\alpha.a) \models t, \Pi_4 \quad \text{Via inductive hypothesis}
    \]
    Let \( \Pi_3 = \Pi_4 \)
    Need to show \( \sigma; (\alpha.a) \models t, \Pi_4 \) \quad \text{Shown via induction above}
    Need to show \( \models s'_1; s_2 : \Pi_3 \rightarrow \Pi_2 \) \quad \text{Shown via application of [CC-Seq]}

• [CC-Read] where \( s = (x = y.f) \):
  \[
  \Pi_2 = (\Pi_1 \setminus \{y.f^0\}) \cup \{y.f^o\} \quad \text{Via [CC-Read]}
  \]
  \[
  x \notin \text{Vars(} \Pi_1, e \text{)} \quad \text{Via [CC-Read]}
  \]
  \[
  s' = \text{skip}, \sigma' = \sigma[x := v], v = S(\sigma(y.f)) \quad \text{Via Evaluation}
  \]
  Let \( \Pi_3 = \Pi_2 \)
  Need to show \( \models s' : \Pi_2 \rightarrow \Pi_2 \) \quad \text{Shown via [CC-Skip]}
  \[
  a = t \cdot \text{acc}(y.f) \quad \text{Via Evaluation}
  \]
  Need to show \( \sigma'; (\alpha.a) \models t, \Pi_2 \) \quad \text{Via Evaluation}

  All actions in \( \alpha \) proved by \( \text{A3} \) are still proved because we have removed \( y.f^0 \) but added in a \( y.f^o \). Clearly \( \sigma; \alpha.a \models t, y.f^o \).
  \( \sigma \) is proved by \( \text{A3} \) because \( y.f^o \in \Pi_3 \). All history properties in \( \Pi_1 \) remain proved in \( \sigma' \) as \( x \notin \text{Vars(} \Pi_1 \text{)} \).

• [CC-Write] where \( s = (y.f = x) \):
  The proof is similar to above.

• [CC-New] where \( s = (x = \text{new } c) \):
  \[
  \Pi_2 = \Pi_1 \quad \text{[CC-New]}
  \]
  \[
  x \notin \text{Vars(} \Pi_1, e \text{)} \quad \text{Via [CC-Write]}
  \]
  \[
  \sigma' = \sigma[x := \rho] \quad \text{Via Evaluation}
  \]
  \[
  s' = \text{skip}, a = t; e \quad \text{Via Evaluation}
  \]
  Let \( \Pi_3 = \Pi_2 \)
  Need to show \( \models s' : \Pi_2 \rightarrow \Pi_2 \) \quad \text{Shown via [CC-Skip]}
  \[
  a = t \cdot \text{acc}(y.f) \quad \text{Via Evaluation}
  \]
  Need to show \( \sigma'; (\alpha.a) \models t, \Pi_2 \) \quad \text{Via Evaluation}

  All actions in \( \alpha \) proved by \( \text{A3} \) are still proved because we have removed \( y.f^0 \) but added in a \( y.f^o \). Those proved by \( \text{A1}, \text{A2}, \) and \( \text{C1} \) do not change because we have not changed \( \alpha \). Clearly \( \sigma; \alpha.a \models t, y.f^o \), \( a \) is proved by \( \text{A3} \) because \( y.f^o \in \Pi_3 \).

• [CC-Anew]: The proof is similar to above.

• [CC-Assign] where \( s = (x = e) \):
  \[
  s' = \text{skip}, \sigma' = \sigma[x := v], a = t; e, v = \sigma(e) \quad \text{Via Evaluation}
  \]
  \[
  x \notin \text{Vars(} \Pi_1, e \text{)} \quad \text{Via [CC-Assign]}
  \]
  \[
  \Pi_2 = \Pi_1 \cup \{x = e\} \quad \text{Via [CC-Assign]}
  \]
  Let \( \Pi_3 = \Pi_2 \)
  Need to show \( \models s' : \Pi_2 \rightarrow \Pi_2 \) \quad \text{Shown via [CC-Skip]}
  Need to show \( \sigma'; \alpha \models t, \Pi_1 \cup \{x = e\} \) \quad \text{Shown via what is given and the new constraint is true based on \( \sigma' \)}}
• **[CC-Rename]** where \( s = (x \leftarrow y) \):
  \( s' = \text{skip}, \sigma' = \sigma[x := \sigma(y)], a = t : \epsilon \)  
  Via Evaluation
  \( x \notin \text{Vars}(\Pi_1) \)
  \( \Pi_1 = H \cup A[x := y] \)
  \( \Pi_2 = H[y := x] \cup A \)  
  Via [**CC-Rename**]
  Let \( \Pi_3 = \Pi_2 \)
  Need to show \( \triangledown s' : \Pi_3 \rightarrow \Pi_2 \)  
  Shown via [**CC-Skip**]
  Need to show \( \sigma'; \alpha \triangledown \Pi_2 \)  
  Shown via \( x \notin \text{Vars}(\Pi_1) \) and \( \sigma(y) = \sigma'(x) \)

We now state several technical lemmas used in the arguments above. We extend the functions **KillSetAnticipated** and **KillSetHistory** from method names to statements in the expected manner.

**Lemma 9** (Extension). If \( \triangledown s : \Pi_1 \rightarrow \Pi_2 \) and \( \Pi' = \Pi \setminus \text{KillSetHistory}(s) \setminus \text{KillSetAnticipated}(s) \) then \( \triangledown s : \Pi_1 \cup \Pi' \rightarrow \Pi_2 \cup \Pi' \).

**Proof.** By induction on the derivation of \( \triangledown s : \Pi_1 \rightarrow \Pi_2 \) and case analysis on the rule used to conclude that derivation.

• **[CC-Skip]**, **[CC-New]**, **[CC-Assign]**: These rules have no constraints on their input and output \( \Pi \) so the proof holds trivially.

• **[CC-Acq]**:
  \( \Pi_1 = \Pi_2 \)
  \( \forall p. p^\circ \notin \Pi_1 \)  
  [**CC-Acq**]
  \( \forall p. p^\circ \in \Pi_1 \Rightarrow \Pi_1 \triangledown p^\prime \)  
  [**CC-Acq**]
  Need to show \( \forall p. p^\circ \notin (\Pi_1 \cup \Pi') \)  
  [**CC-Acq**]
  \( \forall p. p^\circ \notin \Pi' \)  
  Because an acquire kills \( p^\circ \)
  Need to show \( \forall p. p^\circ \in (\Pi_1 \cup \Pi') \Rightarrow (\Pi_1 \cup \Pi') \triangledown p^\prime \)  
  [**CC-Acq**]
  \( \forall p. p^\circ \notin \Pi' \)  
  Because an acquire kills \( p^\circ \)

• **[CC-Rel]** where \( s = \text{rel}(x) \): In this case \( \Pi_1 = \Pi_2 \) and \( \forall p. p^\circ \notin \Pi_1, p^\prime \notin \Pi_1 \)
  A release kills past checks and acquires, so \( \forall p. p^\circ \notin \Pi' \) and \( p^\prime \notin \Pi' \). Hence \( \triangledown s : \Pi_1 \cup \Pi' \rightarrow \Pi_2 \cup \Pi' \) as required.

• **[CC-Read]**, **[CC-Write]**, **[CC-ARead]**, **[CC-AWrite]**: The only restriction is that the resulting \( \Pi_f \) must contain no \( y.f^\circ \) and must contain \( y.f^\circ \). Unioning with \( \Pi' \) can also not remove \( y.f^\circ \) so that condition is met. If \( \Pi' \) adds in \( a.y.f^\circ \) then by **[CC-Sub]** the rule still holds as the resulting \( \Pi_f \) is greater than the original.

• **[CC-If]**, **[CC-Seq]**, **[CC-Loop]**: By induction.

• **[CC-Call]**, **[CC-Sub]**: All requirements involving subterms are proved by induction. The only remaining requirements are proved because \( \Pi_1 \preceq \Pi_2 \Rightarrow (\Pi_1 \cup \Pi) \preceq (\Pi_2 \cup \Pi) \).

• **[CC-Check]**: If \( \Pi_1 \triangledown p^\circ \) then \( \Pi_1 \cup \Pi' \triangledown p^\circ \).


**Lemma 10** (Substitution). If \( \triangledown s : \Pi_1 \rightarrow \Pi_2 \) and \( \theta : \text{Var} \rightarrow \text{Var} \) is a permutation on variables, then \( \triangledown \theta(s) : \theta(\Pi_1) \rightarrow \theta(\Pi_2) \).

**Proof.** Proof is by induction on the derivation of \( \triangledown s : \Pi_1 \rightarrow \Pi_2 \).

**F. Correctness of BIGFOOT**

**Theorem F.1** (Correctness). Suppose \( P = \overline{D} s_1 || \ldots || s_n \) is a program with inserted checks (\( i.e. \triangledown P \)) that generates a trace \( \alpha \) via
\[
\overline{D} \triangledown \Sigma_0 \rightarrow^\alpha \Sigma
\]
where \( \Sigma_0 \) is the initial state for \( P \) and \( \Sigma \) is terminated. Then \( \alpha \) has a data race on a location \( l \) if and only if \( \alpha \) has a check race on that location.

**Proof.** By Lemma 9, we have that \( \triangledown \overline{D} s_1 || \ldots || s_n, \triangledown \Sigma_0 \). Lemma 9 implies that \( \overline{D} ; \epsilon \triangledown \Sigma_0 \). By Theorem 9, \( \alpha \) has precise checks. Finally, Theorem 9 shows that \( \alpha \) has a data race on location \( l \) if and only if it has a check race on \( l \).

**Lemma 11.** If \( P = \overline{D} s_1 || \ldots || s_n, \triangledown P, \) and \( \Sigma_0 = S_0 \cdot T_0 \) is the initial state for \( P \), then \( \overline{D} ; \epsilon \triangledown \Sigma_0 \).
Proof. By definition, \( T_0 \) maps each \( t \in 1..n \) to \( \langle \sigma, s_t \rangle \), where \( \text{Dom}(\sigma) = \text{FV}(s_1) \cup \ldots \cup \text{FV}(s_n) \). Since \( \vdash P \) can only be derived via [CC-Prog], it must be that \( \forall D \in \overline{D}. \ \vdash D \). Also, we have that \( \sigma; \epsilon \vdash_1 \emptyset \) via [CC-Ctxt]. Consider any \( t \). Since \( \vdash P \), we know that \( \vdash s_t \) via [Program], which implies that \( \vdash s_t \) via Lemma \(?\). This is only derivable via rule [CC-Stmt], which means that \( \vdash s : \emptyset \rightarrow \emptyset \). From the above, rule [CC-Thread] allows us to conclude that \( \overline{D}; \epsilon \vdash_1 \langle \sigma, s_t \rangle \). It then follows from rule [CC-State] that \( \overline{D}; \epsilon \vdash \Sigma_0 \). 

\( \square \)