Array Shadow State Compression for Precise Dynamic Race Detection

James R. Wilcox
Computer Science and Engineering
University of Washington
Seattle, WA, USA
jrw12@cs.washington.edu

Parker Finch
Cognius
Boston, MA, USA
finch.parker@gmail.com

Cormac Flanagan
Computer Science Dept.
University of California, Santa Cruz
Santa Cruz, CA, USA
cormac@cs.ucsc.edu

Stephen N. Freund
Computer Science Dept.
Williams College
Williamstown, MA, USA
freund@cs.williams.edu

***Abstract*—Precise dynamic race detectors incur significant time and space overheads, particularly for array-intensive programs, due to the need to store and manipulate analysis (or *shadow*) state for every element of every array. This paper presents SLIMSTATE, a precise dynamic race detector that uses an adaptive, online algorithm to optimize array shadow state representations. SLIMSTATE is based on the insight that common array access patterns lead to analogous patterns in array shadow state, enabling optimized, space efficient representations of array shadow state with no loss in precision. We have implemented SLIMSTATE for Java. Experiments on a variety of benchmarks show that array shadow compression reduces the space and time overhead of race detection by 27% and 9%, respectively. It is particularly effective for array-intensive programs, reducing space and time overheads by 35% and 17%, respectively, on these programs.**

***Keywords*—concurrency, data race detection, dynamic analysis**

I. INTRODUCTION

The widespread adoption of multi-core processors necessitates a software infrastructure that can reliably exploit multiple threads of control. Developing reliable multithreaded software is extremely difficult, however, due to problems caused by unexpected thread interference, which are notoriously difficult to detect, reproduce, and eliminate.

Perhaps the most fundamental interference problem is a *race condition*, which occurs when two threads concurrently access the same location, where at least one access is a write. Race conditions typically reflect synchronization errors, and they cause highly unintuitive behavior under relaxed memory models [1]. Moreover, reasoning about richer concurrency properties, such as atomicity, serializability, determinism, cooperability, functional correctness, *etc.*, often requires first identifying or limiting where races may occur.

Much prior work has explored analyses for race detection. Static analyses (*e.g.* [2], [3], [4]) can offer strong guarantees, but due to computability limitations either miss races or report false alarms, and they may not scale to large systems.

A complementary approach is to use dynamic analyses, which can be *precise* for the observed trace, meaning that they report errors if and only if there is a race condition in the observed trace of the target program. A variety of implementation techniques have been developed for precise dynamic race detection, including vector clocks [5], [6], epochs [7], accordion clocks [8], and others [9]. However, the space and time overhead of precise dynamic race detectors is still prohibitive for many applications.

For example, FASTTRACK's epoch-based representation of the happens-before relation [7] exhibits slowdowns of roughly an order of magnitude and even greater increases in memory usage, particularly for array-intensive programs. Thus, while precise dynamic analyses are critical for detecting data races and other interference problems in multithreaded systems, their current performance limitations preclude their widespread use.

For many programs, much of this space overhead is due to race detection for arrays. We focus on reducing that overhead. To detect races on an array `a`, a dynamic race detector must record sufficient information about the access history of each array element `a[`i`]` to determine if a subsequent access to `a[`i`]` is in a race with any previous access to it. Access information is typically stored in an *array shadow* S, where $S[i]$ represents the access history for `a[`i`]`. For example, in the common case for the FASTTRACK race detector, $S[i]$ contains the *epoch* (the thread identifier and clock) of the last access to `a[`i`]`.

Programs often access arrays by readily identifiable patterns, and those patterns lead to analogous patterns in their shadow arrays. We present a new race detection algorithm, SLIMSTATE, that dynamically identifies these patterns and uses them to compress shadow arrays. For example, suppose a synchronization-free region (SFR) of one thread traverses all odd elements in a newly allocated array `a` of size n while a concurrent SFR traverses all even elements. Once both SFRs have completed, all odd elements in S will be identical, as will all even elements. In this case, rather than requiring n shadow elements, SLIMSTATE represents S as a two-element array T, where the shadow state for `a[`i`]` is now $T[i \bmod 2]$, thus reducing the number of shadow locations from n, which could be very large, to 2.

This compressed representation enables a corresponding reduction in the number of race checks. As the first SFR above traverses all odd elements of `a`, rather than performing a separate race check on each access, SLIMSTATE builds a *footprint* $\{1, 3, 5, \ldots\}$ of accesses to `a` by that thread without yet verifying those accesses are race free. SLIMSTATE *commits* that footprint to the shadow state and verifies race freedom at the thread's next synchronization operation. Since all indices in this footprint map to $T[1]$, SLIMSTATE only needs to check and update that single shadow location, in contrast to the $n/2$ checks and updates necessary with a traditional shadow array

representation.

If a later SFR accesses a different footprint of the array, SLIMSTATE appropriately refines the compressed shadow representation T to avoid any missed races or false alarms. In contrast, prior techniques for compressing shadow state (*e.g.* [10], [11], [6], [12]) are prone to false alarms, as discussed in Section VII below.

For a collection of array-intensive Java benchmarks — those for which at least 50% of their data belongs to arrays — SLIMSTATE reduces the heap footprint by 35% and improves speed by 17%. SLIMSTATE is most effective on array-intensive programs that closely adhere to the access patterns currently recognized. For others, our analysis currently finds less opportunity for compression. Thus, SLIMSTATE demonstrates the potential of using access patterns to compress shadow state and also opens the door to further exploration of more sophisticated pattern matching, adaptive mechanisms to target compression where it is most likely to pay off, and static analyses for computing footprints and patterns ahead of time to reduce the need to dynamically infer them. SLIMSTATE may also be adaptable to shadow compression for objects, but the relatively small size of objects when compared to arrays will necessitate different strategies than those we have found effective for arrays.

Other race detection algorithms that track shadow state for each memory location may reap similar benefits from our compression technique. Moreover, many dynamic analyses for a variety of correctness properties, including atomicity [13], [14], [15], [16], determinism [17], [18], and cooperability [19], also maintain a shadow array for each array in the target program. We expect that shadow array compression may yield comparable benefits for these analyses.

Contributions. The primary contributions of this paper are:

- We demonstrate that array shadow states contain redundancy due to patterned accesses.
- We present the SLIMSTATE precise dynamic race detection algorithm, which partitions arrays into groups of indices with identical shadow states. SLIMSTATE infers those partitions with a dynamic analysis that tracks the footprint of array accesses within each synchronization-free region. (Section III and Section IV).
- We explore the design space for representing partitions and footprints, using techniques such as ranges, strided ranges, and bit sets. (Section V-A).
- We develop an implementation of SLIMSTATE for Java, and we report its performance on a variety of benchmark suites. (Section V-B.)
- For array-intensive programs, SLIMSTATE reduces the minimum heap size by 35% when compared to FASTTRACK. (Section VI.) SLIMSTATE also reduces the running time by 17% for those programs.

II. REVIEW OF DYNAMIC RACE DETECTION

A race condition occurs when two threads concurrently access a memory location, where at least one of those accesses is a write. Accesses are considered *concurrent* if there is no "synchronization dependence" between them, such as the dependence between a lock release by one thread and a subsequent acquire by a different thread. These synchronization

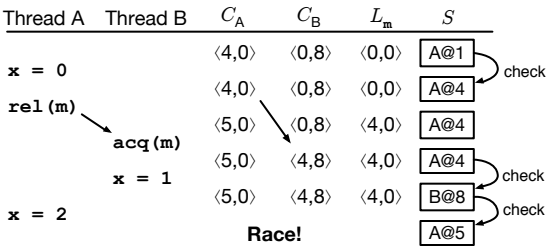


Fig. 1. Race detection using vector clocks and epochs.

dependencies form a partial order over the instructions in a trace called the *happens-before relation* [20].

Precise race detectors typically use *vector clocks* (VCs) [20], [5] to represent the happens-before relation. Given a system in which each thread has a unique identifier $t \in Tid$, a vector clock $V : Tid \to Nat$ records a clock for each thread in the system. Vector clocks are partially ordered ($\sqsubseteq$) in a point-wise manner, with an associated join operation ($\sqcup$):

$$\begin{array}{rcl} V_1 \sqsubseteq V_2 & \text{iff} & \forall t.\ V_1(t) \leq V_2(t) \\ V_1 \sqcup V_2 & = & \lambda t.\ max(V_1(t), V_2(t)) \end{array}$$

A dynamic analysis based on VCs maintains a vector clock C_t for each thread t. The clock entry $C_t(t)$ records thread t's current time. For any other thread u, the clock entry $C_t(u)$ records the clock for the last operation of thread u that happens before the current operation of thread t. Figure 1 illustrates the shadow state C_A and C_B for a trace of operations performed by threads A and B. The additional shadow state component L_m records the vector clock of the last release of lock m. Thus, when thread A performs `rel(m)`, L_m is updated to $C_A = \langle 4, 0\rangle$, and C_A is incremented to $\langle 5, 0\rangle$ to reflect that later steps of A happen after that release. When thread B performs `acq(m)`, C_B is joined via $\sqcup$ with L_m, reflecting that later steps of B happen after the release by A.

Following the FASTTRACK algorithm [7], the shadow state component $S_{\mathtt{x}}$ contains the *epoch* $t@c$ to indicate that the last access to `x` was by thread t when t's clock was c. An epoch $t@c$ *happens before* a vector clock V ($t@c \preceq V$) if and only if the clock of the epoch is less than or equal to the corresponding clock in the vector.

$$t@c \preceq V \quad \text{iff} \quad c \leq V(t)$$

A later access to `x` by thread u is race-free provided $S_{\mathtt{x}} \preceq C_u$.[1] For example, at the second access to `x` in Figure 1, $S_{\mathtt{x}} = A@4$ and $C_B = \langle 4, 8\rangle$. Since $4 \leq C_B(A) = 4$, this access is race free. At the third access to `x`, $S_{\mathtt{x}} = B@8$ and $C_A = \langle 5, 0\rangle$. Since $8 \not\leq C_A(B) = 0$, a race condition exists.

III. SHADOW COMPRESSION & CHECK COALESCING

As discussed above, dynamic race detectors typically maintain shadow state for each memory location, which incurs significant overhead. FASTTRACK's epochs are perhaps the

[1] Due to space limitations, this discussion does not distinguish reads from writes and assumes all accesses to a variable conflict. Our implementation handles concurrent reads by tracking reads and writes in separate epochs and by using FASTTRACK's adaptive epoch/VC representation to record concurrent reads when they are observed [7]. These extensions are straightforward and pose no technical challenges. (See Section V.)

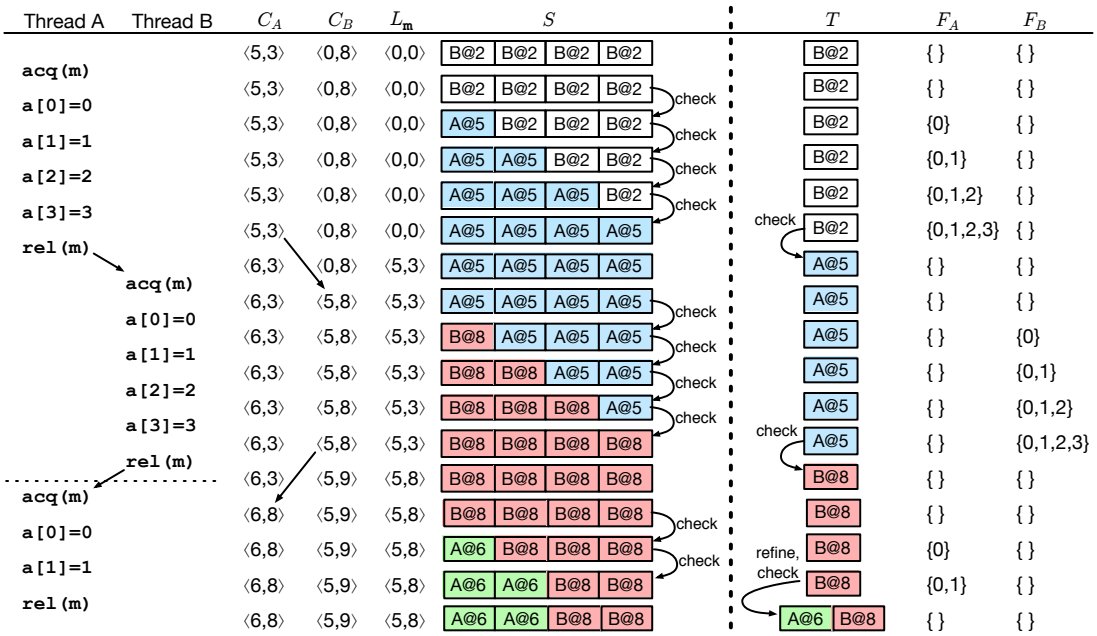


Fig. 2. Full and compressed shadow state for patterned accesses to an array.

most lightweight representation for precise race detection but can still lead to an order of magnitude or more increase in memory footprint, particularly for array-intensive programs. For example, a FASTTRACK implementation for Java [7] that supports concurrent reads maintains two epochs encoded as integers, and a (possibly null) reference to a VC, for each element in an array, even if those elements are one byte each.

SLIMSTATE significantly reduces array shadow state overhead by eliminating redundancy via state compression. Figure 2 provides a simple example of how array access patterns in the target program induce corresponding patterns in the shadow array. Two threads A and B iterate over elements in an array `a` protected by the lock `m`. We show the FASTTRACK shadow state $C_A, C_B, L_{\mathtt{m}}$, and S for each step in the execution, where $S[i]$ records the epoch of the last access to `a[`i`]`.

Note two key aspects of this trace: (1) at each synchronization operation above the dashed line, all elements in the shadow array S are *identical*; and (2) within each of those critical sections, the checker performs four *identical* checks, namely the check $B@2 \preceq \langle 5,\, 3\rangle$ for thread A's writes, and the check $A@5 \preceq \langle 5,\, 8\rangle$ for thread B's writes.

SLIMSTATE dynamically identifies such patterns and uses them to compress the array shadow S into a smaller representation T. Since $S[0], \ldots S[3]$ are identical at synchronization operations, SLIMSTATE uses a single shadow location $T[0]$ to represent all of $S[0], \ldots S[3]$. To do this, SLIMSTATE does not immediately perform a separate race check on each access to `a`; instead, for each thread t, SLIMSTATE builds a *footprint* F_t of `a`-indices that thread t has accessed since its last synchronization operation. When each thread t exits its first critical section, the *footprint* $F_t = \{0, 1, 2, 3\}$ contains all indices corresponding to shadow state $T[0]$. In this case SLIMSTATE performs one race check $T[0] \preceq C_t$, which verifies all accesses to `a[0]`, ..., `a[3]` are race-free; SLIMSTATE then updates one shadow location $T[0]$ to $t@C_t(t)$, which simultaneously records the epoch of the last access to all of `a[0]`, ..., `a[3]`. We refer to this as *committing* the footprint.

Of course, threads may access arrays in more complex patterns as well. For example, multiple threads may concurrently access disjoint blocks of elements, two threads may concurrently access alternating elements, a thread may only traverse some prefix of the array, and so on. SLIMSTATE is designed to maintain full precision in all cases by dynamically *refining* its compressed shadow representation T to precisely represent the original shadow array S regardless of how the underlying array has been accessed.

For example, at the end of the third critical section in Figure 2, footprint $F_A = \{0, 1\}$ does not cover all indices corresponding to shadow location $T[0]$. Thus T is refined to have two locations, where $T[0]$ corresponds to indices 0 and 1, and $T[1]$ corresponds to indices 2 and 3. The commit for A again requires only one check and update to $T[0]$. If a thread touched every array element within a critical section, the subsequent commit would check and update $T[0]$ and $T[1]$.

SLIMSTATE initially uses the *coarse* representation of only a single shadow state to represent all elements in an array, and then refines that representation as necessary. In the worst case, SLIMSTATE may need a *fine* representation where each shadow location in S is maintained separately, in which case $|T| = |S|$. However, our experimental results show that SLIMSTATE is often able to find more compact yet fully precise representations.

Since our algorithm defers a check until after the access, a race may not be reported until the end of the current SFR. However, this is a reasonable tradeoff for better scalability and performance in many situations, since identifying the memory location and the enclosing SFR is often sufficient to understand and fix the race.

FASTTRACK Analysis

$\sigma \in \textit{FT-State} = (C, L, S)$ where
$C : \textit{Tid} \to \textit{VC}$
$L : \textit{Lock} \to \textit{VC}$
$S : \textit{Index} \to \textit{Epoch}$

$\boxed{\sigma \to^b \sigma'}$

[FT ACCESS]
$$\frac{S_i \preceq C_t \qquad S' = S[i := E_t]}{(C, L, S) \to^{acc(t,i)} (C, L, S')}$$

[FT ACQUIRE]
$$\frac{C' = C[t := C_t \sqcup L_m]}{(C, L, S) \to^{acq(t,m)} (C', L, S)}$$

[FT RELEASE]
$$\frac{L' = L[m := C_t] \qquad C' = C[t := inc_t(C_t)]}{(C, L, S) \to^{rel(t,m)} (C', L', S)}$$

SLIMSTATE Analysis

$\tau \in \textit{SS-State} = (C, L, F, P, T)$ where
$C : \textit{Tid} \to \textit{VC}$
$L : \textit{Lock} \to \textit{VC}$
$F : \textit{Tid} \to 2^{\textit{Index}}$
$P :$ partition of $\textit{Index}$
$T : P \to \textit{Epoch}$

$\boxed{\tau \Rightarrow^b \tau'}$

[SS ACCESS]
$$\frac{F' = F[t := F(t) \cup \{i\}]}{(C, L, F, P, T) \Rightarrow^{acc(t,i)} (C, L, F', P, T)}$$

[SS ACQUIRE]
$$\frac{F(t) = \emptyset \qquad C' = C[t := C_t \sqcup L_m]}{(C, L, F, P, T) \Rightarrow^{acq(t,m)} (C', L, F, P, T)}$$

[SS RELEASE]
$$\frac{L' = L[m := C_t] \qquad F(t) = \emptyset \qquad C' = C[t := inc_t(C_t)]}{(C, L, F, P, T) \Rightarrow^{rel(t,m)} (C', L', F, P, T)}$$

$\boxed{\tau \Rightarrow \tau'}$

[SS COMMIT]
$$\frac{\begin{array}{rcl} p & \in & P \\ p & \subseteq & F_t \\ F' & = & F[t := F_t \setminus p] \\ T_p & \preceq & C_t \\ T' & = & T[p := E_t] \end{array}}{(C, L, F, P, T) \Rightarrow (C, L, F', P, T')}$$

[SS REFINE]
$$\frac{\begin{array}{c} P \text{ and } P' \text{ are partitions of } \textit{Index} \\ \forall i \in \textit{Index}.\ T_{\varphi(P,i)} = T'_{\varphi(P',i)} \end{array}}{(C, L, F, P, T) \Rightarrow (C, L, F, P', T')}$$

$\boxed{\tau \Rightarrow^\omega \tau'}$

[SS CHANGE 1]
$$\frac{\tau \Rightarrow \tau' \Rightarrow^\omega \tau''}{\tau \Rightarrow^\omega \tau''}$$

[SS CHANGE 2]
$$\frac{\tau \Rightarrow^\omega \tau' \Rightarrow \tau''}{\tau \Rightarrow^\omega \tau''}$$

Fig. 3. FASTTRACK and SLIMSTATE analysis states and transition rules, where $E_t = t@C_t(t)$.

IV. SLIMSTATE ANALYSIS DETAILS

We now formalize the SLIMSTATE algorithm and prove that its optimizations do not compromise precision.

A. Multithreaded Program Traces

An execution trace ω captures the execution of a multithreaded program by listing the sequence of operations performed by its threads. Each operation of a thread $t \in \textit{Tid}$ can acquire or release a lock $m \in \textit{Lock}$ or access index $i \in \textit{Index} = \{0, \ldots, n-1\}$ of a single global array:

$$\begin{array}{rcl} \omega \in \textit{Trace} & = & \textit{Operation}^* \\ b \in \textit{Operation} & = & acc(t,i) \mid acq(t,m) \mid rel(t,m) \end{array}$$

The simplicity of our execution model is for presentation clarity only. The actual implementation, as described in Section V, is significantly more complex because it must handle the full Java bytecode language and support arbitrarily many arrays and objects, other synchronization mechanisms, and non-conflicting concurrent reads. In this section, we elide these nonessential details in order to clearly present the key ideas of precise dynamic array shadow compression.

B. FASTTRACK Algorithm

Figure 3 summarizes the FASTTRACK algorithm adapted for our idealized trace language. Each FASTTRACK analysis state $\sigma = (C, L, S)$ includes the current vector clock C_t of each thread t; the vector clock L_m for the last release of each lock m; and the epoch S_i for the last access to index i of the global array. The initial analysis state is

$$\sigma_0 = (\lambda t.inc_t(\bot_V),\ \lambda m.\bot_V,\ \lambda i.\bot_e)$$

where $\bot_e$ refers to a minimal epoch $0@0$, $\bot_V$ is the minimal VC $(\lambda t.\ 0)$, and

$$inc_t(V) \quad = \quad \lambda u.\ \texttt{if}\ u = t\ \texttt{then}\ V(u) + 1\ \texttt{else}\ V(u)$$

The rule [FT ACCESS] for $acc(t, i)$ compares the epoch S_i with thread t's vector clock C_t; if this check fails, the analysis get stuck, reflecting a detected race condition. Otherwise this rule updates S_i to the epoch E_t of thread t, where $E_t = t@C_t(t)$. We use $S[i := E_t]$ to denote the function that is identical to S, except that it maps i to E_t. The rule [FT ACQUIRE] for $acq(t, m)$ simply joins L_m, the vector clock of the last release of lock m, with the current thread's clock C_t. Conversely, rule [FT RELEASE] for $rel(t, i)$ updates L_m with C_t and increments the t-component of C_t.

A trace $\omega = b_1 \ldots b_j$ is *race free under* FASTTRACK if it can be successfully analyzed by these rules without getting stuck, *i.e.* there exist states $\sigma_1, \ldots, \sigma_j$ such that $\sigma_0 \to^{b_1} \sigma_1 \to^{b_2} \cdots \to^{b_j} \sigma_j$, which we abbreviate $\sigma_0 \to^\omega \sigma_j$.

C. SLIMSTATE Algorithm

The SLIMSTATE algorithm in Figure 3 represents the shadow array S in a more space efficient manner as the map $T : P \to \textit{Epoch}$. Here, P is some *partition* of $\textit{Index}$, which means P is a set of *parts*, where each part $p \in P$ is a subset of $\textit{Index}$ and each index $i \in \textit{Index}$ is in exactly one part in P. Thus, T contains an epoch T_p for each part $p \in P$.

In the ideal case, as in the first two critical sections in Figure 2, the partition COARSE $= \{\textit{Index}\}$ contains a single part $\textit{Index}$, and thus T contains a single epoch $T_{\textit{Index}}$ that applies to all indices in the array. At the other extreme, P is the trivial partition FINE $= \{\{i\} \mid i \in \textit{Index}\}$ and T contains a separate epoch $T_{\{i\}}$ for each index i, meaning $|\textit{dom}(T)| = |\textit{dom}(S)| = |\textit{Index}|$. Various other partitions are of course possible. For example, partition $P = \{\{0, 2, 4, \ldots\}, \{1, 3, 5, \ldots\}\}$ divides T into two epochs for the even and odd indices in the global array, and $\{\{0, 1\}, \{2, 3\}\}$ characterizes the refined partition at the end of Figure 2.

SLIMSTATE also maintains a footprint $F_t \subseteq \textit{Index}$ recording all indices accessed during thread t's current SFR. Each

array access extends this footprint via rule [SS ACCESS]. A thread's footprint must be committed into T before it performs any synchronization operation, as captured via antecedent $F_t = \emptyset$ in rules [SS ACQUIRE] and [SS RELEASE].

SLIMSTATE's analysis state is represented as a five-tuple (C, L, F, P, T), as shown in Figure 3, and the relation $\Rightarrow^b$ describes how the SLIMSTATE state is updated for each observed event b of the target program.

Footprint commits are handled via the relation $\Rrightarrow$ defined via [SS REFINE] and [SS COMMIT]. Rule [SS REFINE] changes the partition P while still preserving the epoch for each index. Here, $\varphi(P, i)$ denotes the part $p \in P$ that contains i. This rule enables the partition P to be refined so that a footprint F_t is exactly the union of some parts $p_1, \ldots, p_j$ in P. In this case, we say that P is *sufficiently precise* for the footprint F_t.

Once the partition is sufficiently precise for a footprint F_t, rule [SS COMMIT] can commit each part $p \subseteq F_t$. The rule checks that accesses to p are race-free ($T_p \preceq C_t$), updates each T_p for these accesses, and removes p from F_t. If the check $T_p \preceq C_t$ fails, the analysis gets stuck, reflecting a race.

Finally, the rules [SS CHANGE 1] and [SS CHANGE 2] enable multiple $\Rrightarrow$ steps to be performed either before or after each $\Rightarrow^b$ step. The SLIMSTATE algorithm is nondeterministic about when $\Rrightarrow$ steps occur; as described in Section V, our implementation determinizes this algorithm by deferring commits as long as possible to optimize representation efficiency based on the largest footprints possible.

SLIMSTATE does not check for a race on an access to an index i immediately. Instead i is added to the accessing thread's footprint, and the race is detected later when that footprint is committed. We say that state $\tau = (C, L, F, P, T)$ has a *latent race* on i if i is in two footprints simultaneously or if i is in a footprint that, when committed, will reveal a race; that is, if:

1) $\exists t, u \in Tid.\ t \neq u \ \wedge\ i \in F_t \cap F_u$, or
2) $\exists t \in Tid.\ i \in F_t \ \wedge\ T_{\varphi(P,i)} \not\preceq C_t$.

The initial SLIMSTATE state is

$$\tau_0 = (\lambda t.inc_t(\bot_V),\ \lambda m.\bot_V,\ \lambda t.\emptyset, \textsc{Coarse}, \bot_T)$$

where $\bot_T$ maps *Index* to $\bot_e$. A trace ω is *race free under* SLIMSTATE if there exists a state τ with no latent races such that $\tau_0 \Rightarrow^\omega \tau$.

D. SLIMSTATE *Precision*

We now prove that SLIMSTATE is a precise race detector by showing that it behaves the same as FASTTRACK, which previous work has shown to be precise [7]. We begin by introducing the following functions, α and γ, which map FASTTRACK states to SLIMSTATE states, and vice versa:

$$\begin{array}{rcl} \alpha & : & \textit{FT-State} \to \textit{SS-State} \\ \alpha(C, L, S) & = & (C, L, \lambda t.\ \emptyset, \textsc{Fine}, T) \quad \text{where } T_{\{i\}} = S_i \end{array}$$

$$\begin{array}{rcl} \gamma & : & \textit{SS-State} \to \textit{FT-State} \\ \gamma(C, L, F, P, V) & = & (C, L, S) \quad \text{where} \end{array}$$

$$S_i = \begin{cases} E_t & \text{if } i \in F_t \text{ for some } t \\ T_{\varphi(P,i)} & \text{if } \forall t.\ i \notin F_t \end{cases}$$

Note that γ is well defined only for states with no latent races. We now show that each FASTTRACK transition corresponds to a SLIMSTATE transition. Below, we assume trace ω is non-empty, since the empty trace is trivially race free.

Lemma 1. *If $\sigma \to^b \sigma'$ then $\alpha(\sigma) \Rightarrow^b \alpha(\sigma')$.*

Proof is by case analysis on $\sigma \to^b \sigma'$.

Theorem 1. *If a trace ω is race free under* FASTTRACK*, then ω is race free under* SLIMSTATE*.*

Proof: We have that $\sigma_0 \to^\omega \sigma$ for some σ. By Lemma 1 and induction on ω, $\alpha(\sigma_0) \Rightarrow^\omega \alpha(\sigma)$. Moreover, $\tau_0 \Rrightarrow \alpha(\sigma_0)$. This transition can be merged into the first step of $\alpha(\sigma_0) \Rightarrow^\omega \alpha(\sigma)$, yielding $\tau_0 \Rightarrow^\omega \alpha(\sigma)$. By the definition of α, $\alpha(\sigma)$ has no latent races, and thus ω is race free under SLIMSTATE. ■

We now show the converse also holds. Each SLIMSTATE transition corresponds to a FASTTRACK transition provided that the SLIMSTATE states have no latent races.

Lemma 2. *If $\tau \Rightarrow^b \tau'$, and τ and τ' have no latent races, then $\gamma(\tau) \to^b \gamma(\tau')$.*

Proof is by case analysis on $\tau \Rightarrow^b \tau'$.

Theorem 2. *If a trace ω is race free under* SLIMSTATE*, then ω is race free under* FASTTRACK*.*

Proof: We have that $\tau_0 \Rightarrow^\omega \tau$ for some τ with no latent races. Hence every intermediate state in this sequence has no latent race, and thus $\sigma_0 = \gamma(\tau_0) \to^\omega \gamma(\tau)$ by Lemma 2. Thus ω is race free under FASTTRACK. ■

Since the SLIMSTATE relation $\Rightarrow$ is non-deterministic, determining that a trace $\omega = b_1 \ldots b_j$ is racy under SLIMSTATE in theory involves a search to show that there are no possible states $\tau_1, \ldots, \tau_j$ such that $\tau_0 \Rightarrow^{b_1} \tau_1 \Rightarrow^{b_2} \ldots \Rightarrow^{b_j} \tau_j$ where τ_j has no latent races. However, this search is not necessary; the following theorem shows that any reachable state with a latent race is evidence of a racy trace.

Theorem 3. *Suppose $\tau_0 \Rightarrow^\omega \tau$ where τ has a latent race. Then ω is not race free under* SLIMSTATE*.*

Proof: In the analysis sequence $\tau_0 \Rightarrow^\omega \tau$, consider the first transition into a state τ_2 with a latent race, so that $\tau_0 \Rightarrow^{\omega_1} \tau_1 \Rightarrow^b \tau_2 \Rightarrow^{\omega_2} \tau$, where $\omega = \omega_1.b.\omega_2$. By Lemma 2, $\sigma_0 = \gamma(\tau_0) \to^{\omega_1} \gamma(\tau_1)$. By Lemma 3, there does not exist a σ' such that $\gamma(\tau) \to^b \sigma'$. Hence ω is not race free under FASTTRACK, and by Theorem 2, ω is not race free under SLIMSTATE. ■

Lemma 3. *Suppose $\tau \Rightarrow^b \tau'$ where τ' has a latent race but τ does not. Then there is no σ' such that $\gamma(\tau) \to^b \sigma'$.*

Proof is by case analysis on the type of latent race in τ'.

V. SLIMSTATE IMPLEMENTATION

We have implemented SLIMSTATE for Java and describe the most salient aspects of this implementation below.

A. Representation of Footprints and Partitions

The idealized SLIMSTATE algorithm of the previous section manipulates arbitrary partitions and footprints. An im-

plementation could support that full generality by, for example, encoding those structures with bit sets, but a prototype demonstrated that this is prohibitively expensive. Instead, the SLIMSTATE implementation restricts the general algorithm to footprints and partitions that have efficient representations and that reflect common array usage patterns.

Footprints. SLIMSTATE represents footprints as *strided ranges* of the form $\langle b{:}e{:}k\rangle$, with the following meaning:

$$\langle b{:}e{:}k\rangle \equiv \{b+ik \mid b \le b+ik \le e\}$$

Thus, $\langle 0:3:1\rangle$ encodes $\{0,1,2,3\}$, $\langle 1:99:2\rangle$ encodes $\{1,3,\ldots,99\}$, and EMPTY $= \langle 1:0:1\rangle$ encodes the empty set. For all non-empty strided ranges $\langle b{:}e{:}k\rangle$, we require that $0 \le b \le e$, $k \ge 1$, and k divides $e-b$. Strided ranges can represent many footprints observed in practice. The following algebraic rules describe how to extend a strided range with another index (which occurs in the [SS ACCESS] rule).

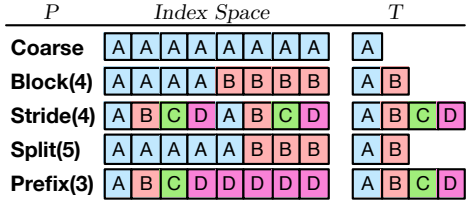
$$\begin{array}{rcl}
\langle b{:}e{:}k\rangle \cup \{i\} & = & \langle b{:}e{:}k\rangle \ \text{ if } i \in \langle b{:}e{:}k\rangle \\
\text{EMPTY} \cup \{i\} & = & \langle i{:}i{:}1\rangle \\
\langle b{:}b{:}1\rangle \cup \{b+k\} & = & \langle b{:}b+k{:}k\rangle \\
\langle b{:}b{:}1\rangle \cup \{b-k\} & = & \langle b-k{:}b{:}k\rangle \\
\langle b{:}e{:}k\rangle \cup \{e+k\} & = & \langle b{:}e+k{:}k\rangle \\
\langle b{:}e{:}k\rangle \cup \{b-k\} & = & \langle b-k{:}e{:}k\rangle
\end{array}$$

If a strided range footprint cannot be extended with a new index, SLIMSTATE commits the current footprint, and then adds the index to the new empty footprint.

Partitions. The SLIMSTATE implementation supports the following *partition modes*, where n denotes the length of the corresponding program array, and d and s divide n:

$$\begin{array}{rcl}
\text{COARSE} & \equiv & \{\{0,\ldots,n{-}1\}\} \\
\text{FINE} & \equiv & \{\{0\},\ldots,\{n{-}1\}\} \\
\text{BLOCK}(d) & \equiv & \{\{0,\ldots,d{-}1\},\{d,\ldots,2d{-}1\},\ldots, \\
 & & \ \{n{-}d,\ldots,n{-}1\}\} \\
\text{STRIDE}(s) & \equiv & \{\{0,s,2s,\ldots,n{-}s\}, \\
 & & \ \{1,s{+}1,2s{+}1,\ldots,n{-}s{+}1\},\ldots, \\
 & & \ \{s{-}1,2s{-}1,3s{-}1,\ldots,n{-}1\}\} \\
\text{SPLIT}(i) & \equiv & \{\{0,\ldots,i{-}1\},\{i,\ldots,n{-}1\}\} \\
\text{PREFIX}(i) & \equiv & \{\{0\},\{1\},\ldots\{i{-}1\},\{i,i{+}1,\ldots,n{-}1\}\}
\end{array}$$

Below are several examples of how these modes divide the Index space for an 8 element array into disjoint parts (labeled $A, B, \ldots$) that each correspond to a single entry in the compressed array shadow T.



Partition BLOCK(d) applies when SFRs accessing the array always access one or more complete blocks of d elements. Partition STRIDE(s) applies when SFRs follow a strided access pattern with steps of size s. The COARSE and FINE partitions described previously can be represented either as blocks or strides:

$$\text{COARSE} \equiv \text{BLOCK}(n) \equiv \text{STRIDE}(1) \equiv \{\{0,\ldots,n{-}1\}\}$$
$$\text{FINE} \equiv \text{BLOCK}(1) \equiv \text{STRIDE}(n) \equiv \{\{0\},\ldots,\{n{-}1\}\}$$

Partition SPLIT(i) applies when elements $0,\ldots,i-1$ are accessed together, and similarly for $i,\ldots,n-1$. Partition PREFIX(i) maintains separate shadow locations for the first i elements. This is particularly useful when a program creates a large buffer but only ever touches the first few elements.

TABLE I. PARTITION REFINEMENT RULES, WHERE $f \neq$ EMPTY.

P	$f = \langle b{:}e{:}k\rangle$	$\mathit{refine}(P,f)$
COARSE	$f = \langle 0{:}n{-}1{:}1\rangle$	COARSE
COARSE	$f = \langle 0{:}e{:}1\rangle$	SPLIT$(e{+}1)$
COARSE	$f = \langle b{:}n{-}1{:}1\rangle$	SPLIT(b)
COARSE	$d' = \gcd(n,b,e{+}1),\ \ 1{<}d',\ \ k{=}1$	BLOCK(d')
COARSE	$b{<}k,\ \ n{-}1{<}e{+}k,\ \ 1{<}k,\ \ k{\mid}n$	STRIDE(k)
COARSE	$e{<}n/4$	PREFIX$(e{+}1)$
BLOCK(d)	$d' = \gcd(d,b,e{+}1),\ 1{<}d',\ k{=}1$	BLOCK(d')
SPLIT(i)	$f = \langle 0{:}i{-}1{:}1\rangle$	SPLIT(i)
SPLIT(i)	$f = \langle i{:}n{-}1{:}1\rangle$	SPLIT(i)
SPLIT(i)	$f = \langle 0{:}n{-}1{:}1\rangle$	SPLIT(i)
SPLIT(i)	$d' = \gcd(n,i,b,e{+}1),\ 1{<}d',\ k{=}1$	BLOCK(d')
SPLIT(i)	$d = \max(i,e),\ d{<}n/4$	PREFIX$(d{+}1)$
STRIDE(s)	$\begin{pmatrix} s' = \mathrm{lcm}(k,s),\ \ s'{\mid}n,\ \ 1{<}s' \\ b{<}k,\ \ n{-}1{<}e{+}k \end{pmatrix}$	STRIDE(s')
PREFIX(i)	$f = \langle i{:}n{-}1{:}1\rangle$	PREFIX(i)
PREFIX(i)	$e{<}i$	PREFIX(i)
PREFIX(i)	$2e{<}n{-}1$	PREFIX$(2e)$
Any	*Any*	FINE

Each partition mode provides an efficient mapping from an array index $i \in \mathit{Index}$ to a corresponding index in the shadow array T. Eg, for BLOCK(d), this mapping is simply $(i \text{ div } d)$.

Table I describes how a partition P is refined to a new partition $\mathit{refine}(P,f)$ that is sufficiently precise for the footprint f. (Recall P is sufficiently precise for f if f is exactly the union of some parts $p_1,\ldots,p_j$ in P.) Note that $\mathit{refine}(P,f)$ is a refinement of P: each part in P becomes the union of some parts in $\mathit{refine}(P,f)$. The given rules are applied in the order listed, and they attempt to minimize the number of parts in the resulting partition. In particular, if P is already sufficiently precise for f, then $P = \mathit{refine}(P,f)$.

COARSE Rules. The first three rules either preserve the COARSE partition or convert it to SPLIT(d), if possible. The fourth rule converts the partition to BLOCK(d'), if f exactly covers one or more blocks in BLOCK(d'). (We require $1 < d'$ to avoid converting to FINE $=$ BLOCK(1) before exploring other rules.) The fifth rule converts COARSE to STRIDE(k) when given an appropriate footprint. The last rule converts to COARSE to prefix mode if the largest index in the footprint is in the first quartile of the array, a threshold heuristically chosen to balance various implementation tradeoffs.

BLOCK(d) Rules. We refine this partition to have a smaller block size d' if we can find a d' that evenly divides both the original size d and the range end points of the footprint f. We choose the largest block size satisfying this requirement.

SPLIT(i) Rules. We remain in SPLIT(i) if the footprint matches either or both parts. We may transition to BLOCK(d') if we can find a d' that evenly divides the array size, the index i, and the endpoints of the footprint f. We may also transition

to prefix mode if the original split and the endpoints of f are all within the first quartile of the array.

STRIDE(s) Rules. We refine STRIDE(s) to STRIDE(s') if s' a multiple of step size s and f is an appropriate strided range.

PREFIX(i) Rules. A prefix partition remains the same if the footprint f is $\{i, \ldots, n-1\}$, or if f only contains indices in the finely tracked prefix. Otherwise, we (at least) double the size of the finely tracked prefix to appropriately amortize the cost of initializing a new, larger shadow array.

If none of the above rules apply, the partition becomes FINE.

B. Java Implementation Details

We built our prototype in the ROADRUNNER analysis framework [21]. ROADRUNNER takes as input a compiled Java target program and inserts instrumentation code into the target to generate an event stream of memory and synchronization operations. Back-end checking tools process these events as the target executes. Standard Java library classes are not instrumented by ROADRUNNER, and so they are not checked for races and are assumed to not perform synchronization relevant to the target program. ROADRUNNER contains special handling for `Object` and `Thread` methods providing basic synchronization operations, such as `wait()` and `notify()`, as well as `volatile` variables. To facilitate comparisons, SLIMSTATE reuses the FASTTRACK reference implementation wherever possible. Indeed, the only substantive difference is in the treatment of arrays. Objects are tracked exactly as in FASTTRACK. Both tools prevent races on their shadow state via non-blocking optimistic concurrency control mechanisms.

SLIMSTATE correctly handles concurrent reads by maintaining separate 32-bit read and write epochs, R and W, in each shadow location and imposing ordering requirements on pairs of accesses only when at least one is a write. When reads for a location are not totally ordered, the single epoch R is replaced with a vector clock, as in FASTTRACK [7].

Array Shadows. For each array, SLIMSTATE creates an `ArrayShadow` object encapsulating the corresponding partition P and shadow array T, where T is stored as an array of adaptive epoch pairs (R, W) described in the previous paragraph. When P is not sufficiently precise for a footprint being committed, SLIMSTATE refines that P, as described in the previous section, and similarly refines the shadow array T as necessary. `ArrayShadow` objects support concurrent refinements (which typically leave P unchanged) and commits via optimistic concurrency control. Each `ArrayShadow` also maintains separate read and write footprints as strided ranges for each thread that has touched its corresponding array.

Commit Buffers. For each thread t, SLIMSTATE maintains a *commit buffer* recording all `ArrayShadows` with non-empty read or write footprints for t. SLIMSTATE flushes that buffer and commits the corresponding footprints whenever t performs a synchronization operation.

As mentioned earlier, if thread t performs an array read (or write) that cannot be merged into the `ArrayShadow`'s read (or write) footprint, the footprint is committed immediately, and the access is recorded in the new empty footprint.

We limit the size of commit buffers to 2048 entries and flush them if they become full. This may lead to sub-optimal commits for some arrays, but our experience indicates that this does not occur often and that larger buffers degrade performance by preventing the shadow state for an array from being garbage collected even if the array itself has been.

FINE *and* PREFIX *Modes.* Once the partition for an array is in FINE or PREFIX(i) mode, SLIMSTATE stops building footprints for that array and instead commits accesses to the shadow array as they occur.

Small Arrays. The overhead of managing footprints and partitions for small arrays can be higher than tracking those arrays with FINE partitions from the start. SLIMSTATE treats all arrays below a threshold size of 16 as having a FINE partition.

VI. EVALUATION

We evaluate the effectiveness of SLIMSTATE on the benchmarks listed in Figure 4. Each is labeled with its source: (1) the Java Grande Benchmarks [22]; (2) the DaCapo Benchmarks [23]; (3) `jbb` and `mtrt` [24]; (4) `colt` [25]; and (5) `raja` [26]. The Java Grande benchmarks were configured with 4 worker threads and their largest data sets. ROADRUNNER contains special provisions to compensate for bugs in the barrier implementations in some of those programs [27]. We separated the DaCapo programs from the DaCapo harness and excluded the tradebeans benchmark to avoid limitations of ROADRUNNER's instrumentation loading support. The code in several specific class files in other programs was excluded from analysis for similar reasons. These programs were configured to use the default benchmark parameters. The `jbb` benchmark was modified to terminate after a fixed number of transactions rather than after a fixed time.

All experiments were performed on an Apple Mac Pro with a 2.7GHz 12-core Pentium Xeon processor with hyper-threading and 64GB of memory. We used the Oracle HotSpot Server VM 1.7 with the default parallel garbage collector. Both FASTTRACK and SLIMSTATE are precise and report no spurious warnings or missed races. We manually verified that both tools report the same races modulo variations due to observed interleavings.

Array Shadow Compression. Figure 4 provides an overview of the shadow compression achieved by SLIMSTATE's various partition modes. These benchmarks are ordered according to how well SLIMSTATE compresses array shadow state. The left side shows the aggregate array sizes for all arrays matching each partition mode, relative to the aggregate size of all allocated arrays. With no compression, the shadow state and arrays will be equal in size. An array may go through multiple refinements over its lifetime, and this figure reflects only its final partition mode. The right half of Figure 4 shows the compressed size of the shadow arrays for each partition mode. For programs like `sor` and `sparse`, the number of array shadows is identical to the number of array elements, since FINE partitions provide no compression. For `crypt` and `raytracer`, on the other hand, the number of shadow locations is well below 1% of the array sizes.

Table II provides a numeric view of this data. The second column shows, for each benchmark, the *shadow fraction*,

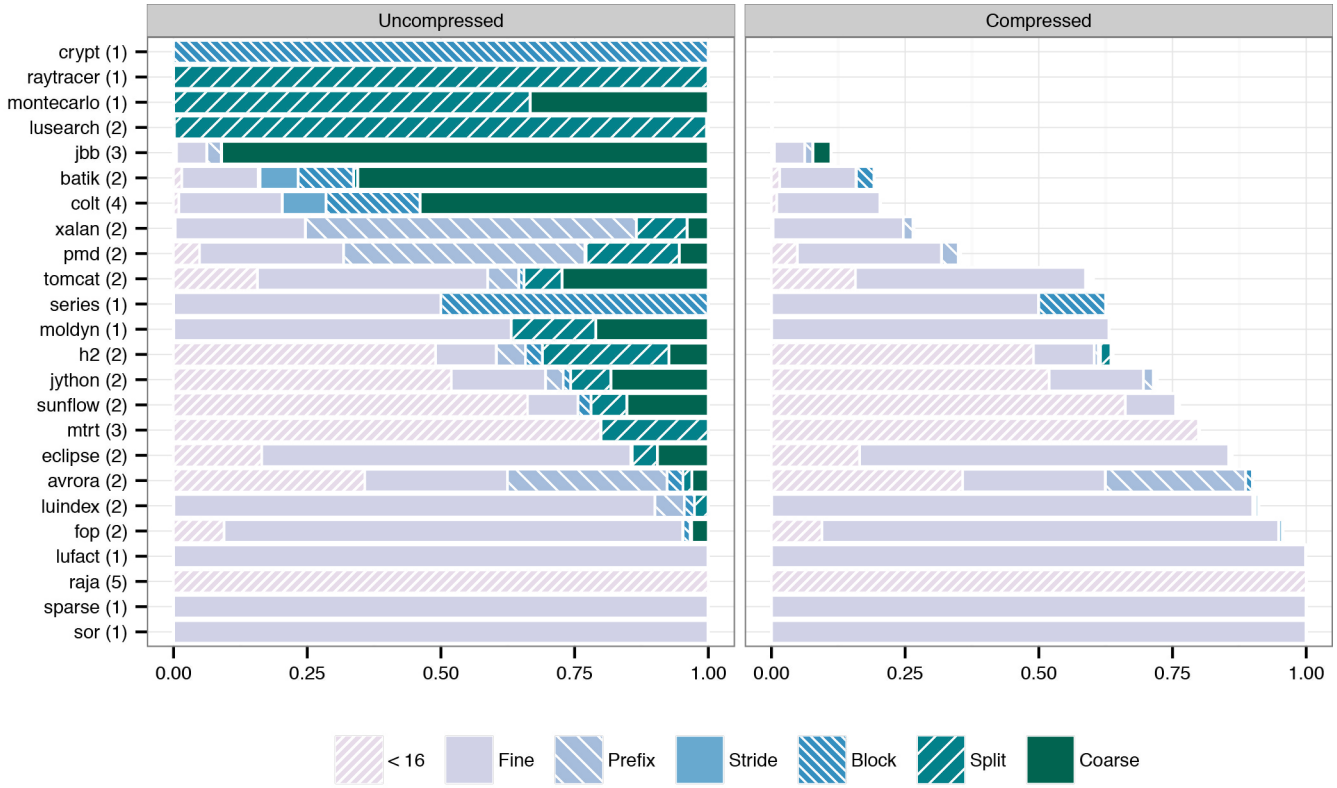


Fig. 4. **Left:** Fraction of shadow array locations in each partitioning mode at the end of their life times. **Right:** How many shadow array locations are needed for each partition mode as a fraction of the original number of locations.

TABLE II. ARRAY SHADOW FRACTIONS.

Program	SLIM STATE	Alternative Bit Set Representations		
		Footprint	Partition	Both
crypt	0.00000057	0.00000023	0.00000021	0.00000017
raytracer	0.000057	0.000057	0.000034	0.000034
montecarlo	0.0018	0.0018	0.0017	0.0017
lusearch	0.0019	0.0019	0.00064	0.00061
jbb	0.11	0.11	0.092	0.089
batik	0.19	0.11	0.11	0.028
colt	0.20	0.11	0.012	0.0099
xalan	0.27	0.25	0.043	0.041
pmd	0.35	0.31	0.12	0.042
tomcat	0.60	0.54	0.094	0.045
moldyn	0.63	0.63	0.63	0.0021
series	0.63	0.63	0.000014	0.000014
h2	0.64	0.61	0.26	0.21
jython	0.72	0.71	0.31	0.28
sunflow	0.76	0.68	0.42	0.35
mtrt	0.80	0.80	0.27	0.25
eclipse	0.86	0.62	–	0.054
avrora	0.90	0.89	0.38	0.37
luindex	0.91	0.88	0.68	0.45
fop	0.96	0.92	0.21	0.047
lufact	1.00	1.00	0.49	0.41
sor	1.00	1.00	1.00	0.004
sparse	1.00	0.53	–	0.0000041
raja	1.00	1.00	0.54	0.54
Average	0.56	0.51	0.26	0.13

which is the number of array shadow locations as a fraction of the number of array elements. SLIMSTATE achieves a shadow fraction of 0.56, in comparison to 1 for FASTTRACK. SLIMSTATE also performs 30% fewer array race checks than FASTTRACK for these programs.

Bit Set Representations. Columns 3–5 investigate how the shadow fraction is influenced by SLIMSTATE's representations for footprints and partitions. Column 3 shows that using bit sets to represent arbitrary footprints reduces the average shadow fraction to 0.51; Column 4 shows that using bit sets to represent partitions reduces it to 0.26 (our prototype ran out of memory on the entries marked '–'.); and Column 5 shows that using bit sets to represent both footprints and partitions reduces it to 0.13. These bit set implementations are prohibitively expensive for many programs, but their shadow fractions suggest that more expressive representations, particularly for partitions, may further improve SLIMSTATE's space savings.

Reduced Race Detection Overhead. Table III shows the total number of array elements allocated in each program, as well as that count as a percentage of all locations — array elements plus object fields — tracked during race detection. The table also shows the running time and minimum heap size for each benchmark under three configurations: Base (no race detection), FASTTRACK (FT), and SLIMSTATE (SS).

Since SLIMSTATE is designed to improve the performance of race detection on arrays, we focus our attention on the 15 *array-intensive* programs for which at least half of the checked memory locations are array elements. For those programs, the running time under SLIMSTATE when memory is

TABLE III. ANALYSIS OVERHEAD FOR BENCHMARK PROGRAMS, SORTED BY THE PERCENT OF MEMORY LOCATIONS (ARRAY ELEMENTS + OBJECT FIELDS) THAT ARE ARRAY ELEMENTS. ARRAY-INTENSIVE PROGRAMS HAVE A PERCENTAGE OF ARRAY ELEMENTS OF AT LEAST 50%.

Program	Array Elems		Shadow	Minimum Heap Space				Time with No Space Limits			
	Count (x10^6)	% of Total	Fraction	Base (MB)	$\frac{\text{FT}}{\text{Base}}$	$\frac{\text{SS}}{\text{Base}}$	$\left(\frac{\text{SS}}{\text{FT}}\right)$	Base (sec)	$\frac{\text{FT}}{\text{Base}}$	$\frac{\text{SS}}{\text{Base}}$	$\left(\frac{\text{SS}}{\text{FT}}\right)$
Array-Intesive Programs											
crypt	150.0	100	0.00000057	193.6	27.67	1.00	(0.04)	0.62	87.71	15.06	(0.17)
lufact	4.0	100	1.00	32.1	7.00	7.00	(1.00)	0.81	40.55	36.70	(0.91)
series	2.0	100	0.63	22.2	3.86	3.04	(0.79)	218.51	1.03	1.03	(1.00)
sor	4.0	100	1.0	33.5	4.76	4.76	(1.00)	0.47	19.13	20.53	(1.07)
sparse	16.0	100	1.00	100.0	5.53	3.17	(0.57)	1.49	40.95	39.27	(0.96)
montecarlo	180.0	98	0.0018	619.1	3.55	1.09	(0.31)	3.90	5.92	3.09	(0.52)
lusearch	2,445.8	98	0.0019	1.5	32.25	21.50	(0.67)	2.26	13.09	7.14	(0.55)
xalan	102.1	90	0.27	11.3	10.17	8.50	(0.84)	3.00	6.06	6.56	(1.08)
eclipse	771.1	87	0.86	87.5	8.29	9.14	(1.10)	29.83	9.83	10.72	(1.09)
luindex	2.6	86	0.91	1.3	63.00	61.00	(0.97)	1.24	10.56	11.18	(1.06)
batik	6.3	80	0.19	22.7	5.09	4.14	(0.81)	6.44	2.15	2.23	(1.04)
tomcat	21.3	76	0.60	15.0	11.20	12.10	(1.08)	14.88	2.08	2.08	(1.00)
colt	0.4	76	0.20	0.6	73.05	48.03	(0.66)	15.98	1.10	1.13	(1.02)
jbb	35.0	72	0.11	96.0	3.97	3.09	(0.78)	1.92	11.63	11.48	(0.99)
avrora	0.7	55	0.90	1.3	23.00	21.43	(0.93)	4.28	3.69	3.71	(1.01)
Geo Mean (Array-Intensive)					11.1	7.1	**(0.65)**		7.9	6.5	**(0.83)**
Object-Intesive Programs											
h2	72.8	37	0.64	272.1	4.46	4.79	(1.07)	12.44	12.58	14.07	(1.12)
fop	2.6	32	0.96	28.9	6.25	6.20	(0.99)	2.59	4.72	4.91	(1.04)
moldyn	0.2	30	0.63	1.4	22.78	21.44	(0.94)	3.68	15.94	18.54	(1.16)
mtrt	0.8	30	0.80	16.0	7.61	8.11	(1.07)	0.44	8.25	9.00	(1.09)
pmd	6.3	30	0.35	45.3	5.31	5.44	(1.02)	3.75	4.79	4.55	(0.95)
jython	27.3	20	0.72	25.6	16.11	11.11	(0.69)	12.02	6.62	7.06	(1.07)
sunflow	4.7	1	0.76	15.0	8.07	7.80	(0.97)	1.65	16.39	17.97	(1.10)
raytracer	1.0	0	0.000057	1.4	19.00	10.27	(0.54)	3.06	18.67	19.12	(1.02)
raja	0.0	0	1.00	1.3	13.60	13.60	(1.00)	0.25	9.76	10.04	(1.03)
Geo Mean (Object-Intensive)					9.8	8.9	(0.90)		9.7	10.0	(1.06)
Overall Geo Mean					10.6	7.7	(0.73)		8.5	7.8	(0.91)

unconstrained is 17%-109% the running time of FASTTRACK, with a geometric mean of 83%. When memory is constrained, SLIMSTATE is able to successfully check those programs in heap spaces that are 4%-108% of the minimum heap required to check them with FASTTRACK, with a geometric mean of 65%. The most significant improvements are achieved for array-intensive programs with patterns matching our modes.

While our focus is primarily array-intensive programs, we also show the object-intensive programs in Table III. These programs, which use fewer arrays and typically exhibit large array shadow fractions, offer less opportunity for compression. Two notable exceptions are `raytracer` and `jython`, which use a small number of arrays heavily and in a compressible way. When all 24 programs are considered, the running time under SLIMSTATE when memory is unconstrained is reduced to 91% of the running time under FASTTRACK. When memory is constrained, SLIMSTATE is able to check those programs in heap spaces that are on average only 73% of the minimum heap size required to check them with FASTTRACK.

These timing measurements are the average of 20 runs when the JVM's maximum heap size is set to the machine's physical memory size of 64GB, which is roughly an order of magnitude larger than the maximum space used by any benchmark under any checker. Minimum heap space is measured by iteratively reducing the JVM's maximum permitted heap size until execution fails to terminate within five times the running time of FASTTRACK under no memory constraints. (Increasing that time limit led to no discernible changes in the results.)

Some programs where all arrays ultimately have FINE partitions, such as `sparse`, still exhibit space and time savings because show arrays are still compressed for parts of their lifetimes, including periods when memory pressure is greatest. SLIMSTATE space savings drop from 35% to about 20% when we do not by default use FINE partitions for arrays with fewer than 16 elements, due to the additional bookkeeping.

Time vs. Space Graphs. Since Java is a garbage collected language, the VM can reduce overall running time at the cost of increased space usage for the heap, and vice-versa. Figure 5 provides a more complete view of this time/space tradeoff for the checkers on representative array-intensive programs. (Graphs for all programs appear in our extended report [28].)

In these graphs, JVM heap size is normalized to Base Minimum Heap Size from Table III, and run time is normalized to Base Time with No Space Limits. We also include FASTTRACKOBJ, a version of FASTTRACK that only checks for races on objects, as a proxy for "ideal" behavior if *all* array checking time and space overhead were eliminated. The graphs for `crypt`, and `lusearch` show sizable improvements in both time and space. Other programs, such as `sparse` show modest

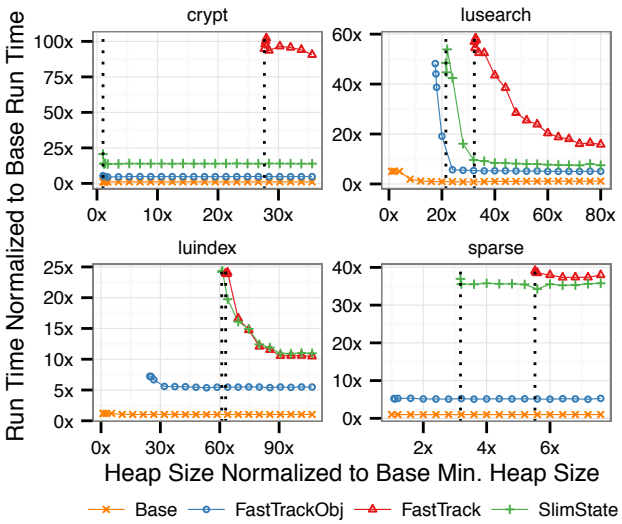


Fig. 5. Run times for different heap sizes.

gains in speed but a sizable drop in the minimum heap size. For `luindex`, limited opportunities for compression combined with additional bookkeeping lead SLIMSTATE to perform no better (or slightly worse) than FASTTRACK. That pattern is repeated in several other programs. The high variability under different workloads indicates that SLIMSTATE may be most effective when used in conjunction with adaptive feedback to target compression where it will be most effective.

VII. Related Work

A common approach for shadow state compression in earlier tools is using a single shadow state for each array and object [10], [29], [11], [6], [7], [30]. Octet uses a similar coarse-grained mechanism to enable efficient tracking of cross-thread dependences inside a JVM [30]. To determine if a reported race is real, one approach is to re-run the program with a more fine grained shadow state for the offending array, as in MultiRace [6]. Another is to switch to a fine-grained representation on-the-fly and hope that a second race manifests later if the first had been a real race, as in RaceTrack [11]. Other recent work [12] uses a single shadow state for contiguous memory locations if those locations are accessed within the same critical sections. However, only the first two critical sections accessing those locations are considered, and the shadow state is not refined if later accesses are not correlated, resulting in potential false alarms.

In contrast, SLIMSTATE's adaptive compression technique precisely tracks the happens-before relation with no false alarms or missed races.

Several race checkers defer the processing of accesses until later in the execution. RecPlay [31], for example, records all memory locations accessed within each SFR and then uses this information to verify that concurrent regions access disjoint memory during replay. DRD [32] and ThreadSanitizer [33] similarly buffer accesses but do not infer patterns or compress the shadow state. Similar buffering is also common in transactional memory systems [34].

Many other dynamic analyses improve space and time performance by sacrificing precision guarantees in various ways. For example, Eraser's lockset algorithm reasons about lock-based synchronization, augmented with specialized handling of thread-local and read-shared data [35]. Other approaches extend that algorithm to be less prone to false alarms [29], [11], [6], [36], [37]. Sampling techniques have also been explored, with some loss of precision [38], [39], [40]. RADISH [41] checks race-freedom of most accesses in hardware at access time, but defers some race checks into a queue processed by another core asynchronously. The hardware waits for the queue to empty at synchronization points, in the same way that SLIMSTATE empties its access buffers at those points.

It is also possible to design efficient, specialized, race detectors for specific programming models such as structured parallelism in Cilk or X10 [17], [42], partitioned global address space programs [43], or GPUs [44].

A number of static analyses reason about access patterns in the context of race detection. DPJ, for example, uses source-level type annotations to enforce access patterns guaranteed to be race free [45]. REDCARD uses a global analysis to infer whether simple access patterns for arrays and objects are followed within all synchronization-free code blocks on all executions [46]. A dynamic analysis can partition shadow state based on that information, but REDCARD uses a very expensive and somewhat brittle whole program analysis. SLIMSTATE instead infers access patterns that are only required to hold for the observed program trace, enabling compression based on patterns that static analysis fails to identify or that are only violated on rare or exceptional control flow paths. REDCARD misses such opportunities. On the benchmarks in common with that work, REDCARD reduces the number of shadow locations allocated for both arrays and objects by 27%. SLIMSTATE reduces that number by 80%, despite not addressing objects.

VIII. Summary

Dynamic race detectors incur significant space overhead for recording analysis shadow state, particularly for arrays. Prior work compressed shadow state by compromising precision. We show that, by recording footprints and adaptively refining the compression strategy, SLIMSTATE achieves significant compression of array shadow state with no loss of precision. Dynamic analyses for richer concurrency properties, such as atomicity or determinism, often must first reason about race conditions in the observed trace, and the contributions of this work also serves to potentially improve those analyses as well.

SLIMSTATE builds footprints dynamically, which involves some run time overhead. A promising direction for future work is to compute these footprints statically. Adapting SLIMSTATE to access patterns and shadow compression for objects is another avenue of future work, although it is likely that other techniques will be required to achieve good performance, given the size of most objects.

Acknowledgment

This work was supported, in part, by NSF Grants 1337278, 1421051, 1421016, and 1439042.

REFERENCES

[1] S. V. Adve and H.-J. Boehm, "Memory models: a case for rethinking parallel languages and hardware," *Commun. ACM*, vol. 53, no. 8, pp. 90–101, 2010.

[2] D. R. Engler and K. Ashcraft, "RacerX: Effective, static detection of race conditions and deadlocks," in *SOSP*, 2003.

[3] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for Java," in *PLDI*, 2006, pp. 308–319.

[4] M. Abadi, C. Flanagan, and S. N. Freund, "Types for safe locking: Static race detection for Java," *TOPLAS*, vol. 28, no. 2, pp. 207–255, 2006.

[5] F. Mattern, "Virtual time and global states of distributed systems," in *Workshop on Parallel and Distributed Algorithms*, 1988.

[6] E. Pozniansky and A. Schuster, "MultiRace: Efficient on-the-fly data race detection in multithreaded C++ programs," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 3, pp. 327–340, 2007.

[7] C. Flanagan and S. N. Freund, "FastTrack: Efficient and precise dynamic race detection," in *PLDI*, 2009, pp. 121–133.

[8] M. Christiaens and K. D. Bosschere, "TRaDe: Data Race Detection for Java," in *International Conference on Computational Science*, 2001, pp. 761–770.

[9] T. Elmas, S. Qadeer, and S. Tasiran, "Goldilocks: A race and transaction-aware Java runtime," in *PLDI*, 2007, pp. 245–255.

[10] C. von Praun and T. Gross, "Object race detection," in *OOPSLA*, 2001, pp. 70–82.

[11] Y. Yu, T. Rodeheffer, and W. Chen, "RaceTrack: Efficient detection of data race conditions via adaptive tracking," in *SOSP*, 2005, pp. 221–234.

[12] Y. W. Song and Y. Lee, "Efficient data race detection for C/C++ programs using dynamic granularity," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 679–688.

[13] C. Flanagan and S. N. Freund, "Atomizer: A dynamic atomicity checker for multithreaded programs," in *POPL*, 2004, pp. 256–267.

[14] L. Wang and S. D. Stoller, "Runtime analysis of atomicity for multithreaded programs," *IEEE Trans. Software Eng.*, vol. 32, no. 2, pp. 93–110, 2006.

[15] C. Flanagan, S. N. Freund, and J. Yi, "Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs," in *PLDI*, 2008, pp. 293–303.

[16] S. Biswas, J. Huang, A. Sengupta, and M. D. Bond, "DoubleChecker: efficient sound and precise atomicity checking," in *PLDI*, 2014.

[17] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark, "Detecting data races in Cilk programs that use locks," in *Proceedings of the 10th Symposium on Parallel Algorithms and Architectures*, 1998, pp. 298–309.

[18] C. Sadowski, S. N. Freund, and C. Flanagan, "SingleTrack: A dynamic determinism checker for multithreaded programs," in *ESOP*, 2009, pp. 394–409.

[19] J. Yi, C. Sadowski, and C. Flanagan, "Cooperative reasoning for preemptive execution," in *PPoPP*, 2011.

[20] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[21] C. Flanagan and S. N. Freund, "The RoadRunner dynamic analysis framework for concurrent programs," in *PASTE*, 2010, pp. 1–8.

[22] Java Grande Forum, "Java Grande Forum benchmark suite," Available from `http://www.javagrande.org/`, 2013.

[23] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. B. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA*, 2006, pp. 169–190.

[24] Standard Performance Evaluation Corporation, "SPEC benchmarks," `http://www.spec.org/`, 2003.

[25] CERN, "Colt 1.2.0," `http://dsd.lbl.gov/~hoschek/colt/`, 2004.

[26] E. Fleury and G. Sutre, "Raja, version 0.4.0-pre4," Available at `http://raja.sourceforge.net/`, 2007.

[27] C. Flanagan and S. N. Freund, "Adversarial memory for detecting destructive races," in *PLDI*, 2010, pp. 244–254.

[28] J. Wilcox, P. Finch, C. Flanagan, and S. N. Freund, "SlimState: Array shadow state compression for precise dynamic race detection (extended report)," http://www.cs.williams.edu/~freund/papers/slimstate-extended.pdf, 2015.

[29] R. O'Callahan and J.-D. Choi, "Hybrid dynamic data race detection," in *PPoPP*, 2003, pp. 167–178.

[30] M. D. Bond, M. Kulkarni, M. Cao, M. Zhang, M. F. Salmi, S. Biswas, A. Sengupta, and J. Huang, "OCTET: capturing and controlling cross-thread dependences efficiently," in *OOPSLA*, 2013, pp. 693–712.

[31] M. Ronsse and K. D. Bosschere, "RecPlay: A fully integrated practical record/replay system," *TOCS*, vol. 17, no. 2, pp. 133–152, 1999.

[32] "DRD: a thread error detector," http://valgrind.org/docs/manual/drd-manual.html, 2014.

[33] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: Data race detection in practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, 2009, pp. 62–71.

[34] N. Shavit and D. Touitou, "Software transactional memory," in *ACM Symposium on Principles of Distributed Computing*, 1995, pp. 204–213.

[35] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson, "Eraser: A dynamic data race detector for multi-threaded programs," *TOCS*, vol. 15, no. 4, pp. 391–411, 1997.

[36] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov, "Dynamic race detection with LLVM compiler - compile-time instrumentation for ThreadSanitizer," in *RV*, 2011, pp. 110–114.

[37] X. Xie and J. Xue, "Acculock: Accurate and efficient detection of data races," in *CGO*, 2011, pp. 201–212.

[38] M. D. Bond, K. E. Coons, and K. S. McKinley, "PACER: proportional detection of data races," in *PLDI*, 2010, pp. 255–268.

[39] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk, "Effective data-race detection for the kernel," in *OSDI*, 2010, pp. 151–162.

[40] L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H.-J. Boehm, "IFRit: interference-free regions for dynamic data-race detection," in *OOPSLA*, 2012, pp. 467–484.

[41] J. Devietti, B. P. Wood, K. Strauss, L. Ceze, D. Grossman, and S. Qadeer, "Radish: Always-on sound and complete race detection in software and hardware," in *ISCA*, 2012, pp. 201–212.

[42] R. Raman, J. Zhao, V. Sarkar, M. T. Vechev, and E. Yahav, "Efficient data race detection for async-finish parallelism," in *RV*, 2010, pp. 368–383.

[43] C.-S. Park, K. Sen, and C. Iancu, "Scalable data race detection for partitioned global address space programs," in *PPoPP*, 2013, pp. 305–306.

[44] M. Zheng, V. T. Ravi, F. Qin, and G. Agrawal, "GRace: a low-overhead mechanism for detecting data races in GPU programs," in *PPoPP*, 2011, pp. 135–146.

[45] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian, "A type and effect system for deterministic parallel Java," in *OOPSLA*, 2009, pp. 97–116.

[46] C. Flanagan and S. N. Freund, "RedCard: Redundant check elimination for dynamic race detectors," in *ECOOP*, 2013, pp. 255–280.