SLIMFAST: Reducing Metadata Redundancy in Sound and Complete Dynamic Data Race Detection

Abstract
Data races are one of the main culprits behind the complexity of multithreaded programming. Existing data race detectors require large amounts of metadata for each program variable to perform their analyses. The SLIMFAST system exploits the insight that there is a large amount of redundancy in this metadata: many program variables often have identical metadata state. By sharing metadata across variables, a large reduction in space usage can be achieved compared to the state-of-the-art FASTTrack algorithm. SLIMFAST uses immutable metadata to safely support metadata sharing across threads while also accelerating concurrency control. SLIMFAST’s lossless metadata compression achieves these benefits while preserving soundness and completeness.

Across a range of Java benchmarks from the Java Grande, DaCapo and NAS Parallel Benchmark suites, SLIMFAST is able to reduce memory consumption by 1.76x on average, and up to 4.47x for some benchmarks, compared to FASTTrack. By improving cache locality and simplifying concurrency control, SLIMFAST also accelerates data race detection by 1.24x on average, and up to 8.33x for some benchmarks, compared to FASTTrack.

1. Introduction
With the increasing ubiquity of multicore processors in everything from servers to wearables, multithreaded programming has become ever more critical to efficiently utilizing modern processor hardware. Unfortunately, multithreaded programming suffers from a host of well-known programmability challenges. One of the key challenges is the potential for programs to contain data races. Data races can introduce non-sequentially-consistent [37] and even undefined [5] behavior into programs, making programs hard to understand. Data races complicate recording and replaying multithreaded execution [57], verifying atomicity [21, 26] and determinism [8] specifications, enforcing deterministic execution [13, 49] and performing model checking of current programs [7, 45].

Our focus in this work is on sound (reporting all races) and complete (reporting no false races) dynamic data race detection, e.g., tools like FASTTrack [22, 23] and DJIT+ [51]. These race detectors maintain extensive per-variable metadata – recording the most recent reads and writes to a variable by each thread – to preserve soundness and completeness. We build upon FASTTrack [22] because it is the current state-of-the-art in dynamic race detection.

The key insight behind SLIMFAST is that there exists a large amount of cross-variable redundancy in race detection metadata, typically to the extent of several orders of magnitude. Program variables may have the same metadata state when they are accessed together without an intervening lock release operation. Figure 1 shows a trace of operations from a single thread and the metadata state for each variable after each operation. The first highlighted blocks shows that, after the writes to Y and Z, the metadata for X, Y and Z is identical. Similarly, after the read of Z the metadata for Y and Z is identical. There is no need, from a correctness standpoint, to maintain separate copies of metadata for two variables whose metadata is identical. SLIMFAST replaces such redundant per-variable copies with references to a single, shared instance. This instance is immutable for simpler concurrency (see Section 4). We identify several novel invariants of the FASTTrack algorithm to allow checking for metadata redundancy in O(1) time.

Ultimately, SLIMFAST maintains metadata equivalent to that of conventional race detectors [22, 51] but in a more space-efficient way, thus preserving soundness and completeness. Our evaluation shows that SLIMFAST reduces the memory overhead of FASTTrack by 1.76x on average, and
up to 4.47x for some programs. This enables race detection to scale to larger programs and inputs. Due to improved cache locality and simpler concurrency control, SLIMFAST runs 1.24x faster than FASTTrack on average, improving performance by up to 8.33x in some cases.

The FASTTrack detector we build upon employs two metadata formats: a smaller format containing an EpochPair (see Section 3) and a larger EpochPlusVC format containing a write epoch and a vector clock. The EpochPair format is of constant size and is highly amenable to compression via SLIMFAST. In Section 4.5 we show how SLIMFAST achieves optimal redundancy elimination for EpochPair metadata, ensuring there are never two copies of metadata with the same value. The cost of identifying unique vector clocks, however, is much higher, taking $O(n)$ time where $n$ is the number of threads in the program and the size of the vector clocks. SLIMFAST leverages several invariants of vector clock updates to reduce nearly all vector clock redundancy in just $O(1)$ time.

This paper makes the following contributions:

- We present the SLIMFAST algorithm and prove its correctness
- We show that SLIMFAST provides optimal redundancy elimination for the common-case EpochPair metadata, and very good redundancy elimination in practice even for larger EpochPlusVC metadata
- A SLIMFAST implementation that provides an average space reduction of 1.76x and an average speedup of 1.24x over the state-of-the-art FASTTrack system
- We show that the SLIMFAST idea generalizes beyond FASTTrack, saving over 3x space on average in the SPD3 race detector [53] as well

The remainder of this paper is organized as follows. We measure the metadata redundancy in current race detectors in Section 2, describe how the FASTTrack detector works in Section 3, outline the high-level operation of SLIMFAST in Section 4, demonstrate the thread-safety of our analysis in Section 5, and present other implementation details in Section 6. We evaluate SLIMFAST in Section 7.1 and Section 7, and describe related work in Section 8.

2. Redundancy in Race Detection Metadata

Race detection algorithms rely on per-location metadata which tracks the most recent reads and writes to each location. Using this metadata, a race detector can identify conflicting accesses that indicate a data race. FASTTrack [22] improved upon previous race detection algorithms by shrinking the size of the per-location metadata whenever possible without losing precision. However, FASTTrack does not address the redundancy that can arise across locations. Redundancy arises whenever two distinct locations $x$ and $y$ have identical metadata. Metadata redundancy can arise via several access patterns, e.g., whenever $x$ and $y$ are written by the same thread without an intervening release synchronization operation, these two writes occur with the same logical timestamp. Thus, after the second write, the metadata for $x$ and $y$ will have the same value.

We measured the metadata redundancy in the FASTTrack race detector [22], modifying its implementation to store a copy of all of its metadata in a global set. If two metadata instances with the same value are inserted into this unique set, only one copy will be retained. We then measured the size of the unique set along with FASTTrack’s original number of metadata objects. In Figure 2, we compare the average ratio of the original number of metadata objects to the size of the unique set, across the execution of each of our benchmark programs. There are typically several orders of magnitude of metadata redundancy in these programs. Removing this redundancy can yield substantial space and time savings for dynamic race detection without sacrificing precision.

3. Background

In this section we review a formal description of the FASTTrack race detector in Section 3.1, and then describe the modifications needed for SLIMFAST in Section 4.

3.1 FASTTrack Overview

For ease of comparison with prior work, we adopt the notation from the FASTTrack [22] paper. We present an abbreviated version of the FASTTrack operational semantics here and refer readers to [22] for more detail.

In FASTTrack, a program execution is modeled as a sequence of operations performed by a set of threads, called a trace. Each operation in a trace is one of:

- $rd(t, x)$ or $wr(t, x)$, in which thread $t$ reads from or writes to a location $x$;
- $acq(t, m)$ or $rel(t, m)$, in which $t$ acquires or releases a lock $m$; or
- $fork(t, u)$ or $join(t, u)$, in which $t$ forks or synchronizes with another thread $u$.

While operations appear in some order in a trace, that order does not imply that the effects of operations performed by various threads can be linearized. Instead, we derive a partial
happens-before order from a trace α, written \( <_\alpha \), such that 
\( a <_\alpha b \) when \( a \) must occur before \( b \). The happens-before order is the transitive closure of the smallest relation such that 
\( a <_\alpha b \) when \( a \) occurs before \( b \) in \( \alpha \) and either:

- \( a \) and \( b \) are both performed by thread \( t \) (program order);
- \( a \) and \( b \) are operations on the same lock \( m \) (locking); or
- one of \( a \) and \( b \) is fork \((t, u)\) or join \((t, u)\) and the other is 
an operation by thread \( u \).

Given this definition, a race condition occurs when two operations \( a \) and \( b \) both access the same location, at least one of them is a write, and neither \( a <_\alpha b \) nor \( b <_\alpha a \) holds (so that the operations are seen as concurrent in the trace).

**FASTTrack** checks for race conditions by maintaining a tuple of metadata based on vector clocks. A vector clock \( V \) records a timestamp for each thread \( t \) in a system, written as \( V(t) \), and we say that \( V \subseteq V' \) when \( V(t) \leq V'(t) \) for every \( t \). To avoid wasting space, sometimes epochs are used in place of vector clocks; an epoch \( c \otimes t \) is a reduced vector clock that holds a timestamp for just one thread, and is treated as a vector clock that is \( c \) for \( t \) and 0 for every thread other than \( t \). A **FASTTrack** analysis state is a tuple \((C, L, M)\), where \( C_t \) is a vector clock for the thread \( t \), \( L_m \) is a vector clock for the lock \( m \), and \( M_x = (M^R_x, M^W_x) \) is a read vector clock/epoch and a write epoch for the location \( x \). Intuitively, \( C_u(t) \) records the last timestamp at which the thread \( t \) synchronized with \( u \), and \( L_m(t) \), \( M^R_x(t) \), and \( M^W_x(t) \) record the last timestamps at which \( t \) accessed the lock \( m \), read from the location \( x \), and wrote to the location \( x \), respectively. The write metadata for a location is always an epoch; the read metadata may be either an epoch or a full vector clock, depending on whether the location is being read concurrently by multiple threads. We write **EpochPair** for the set of location metadata pairs \((R, W)\) in which both \( R \) and \( W \) are epochs, and **EpochPlusVC** for the set of pairs in which \( R \) is a vector clock and \( W \) is an epoch. The **FASTTrack** operational semantics are presented in the left columns of Figure 3.

We have made three small departures from the **FASTTrack** semantics as originally presented. First, we represent metadata for a location \( x \) as a pair of read and write metadata \( M_x \), instead of tracking read metadata and write metadata in separate pieces of analysis state, to more closely mirror our implementation. We also extend the **FASTTrack** **WRITEEXCLUSIVE** rule (Figure 3) to reset the read epoch \( M^R_x := \bot_e \). This extension is correct because \( M^W_x := E(t) \) and \( M^R_x \leq E(t) \). Any subsequent race-free access to \( x \) must be well-ordered with respect to \( M^W_x \) and thus must also be well-ordered with \( M^R_x \). Any subsequent racing access to \( x \) that is concurrent with \( M^R_x \) will be concurrent with \( M^W_x \) as well, so the read epoch is extraneous and can be cleared. Finally, we have added a **READSHAREDSAMEEPOCH** rule which is used when a thread \( t \) reads **EpochPlusVC** metadata multiple times within a given epoch. This lets us distinguish cases where **EpochPlusVC** updates are necessary, which have different semantics with **SLIMFAST**.

### 4. **SLIMFAST Overview**

With **SLIMFAST**, race detection metadata is made immutable so that it can be shared safely across threads. This requires additional work to perform metadata updates. Figure 4 shows the rules requiring metadata updates, whose semantics differ between **FASTTrack** and **SLIMFAST**. These semantics present many performance optimizations, which we discuss in detail below.

#### 4.1 Optimizing Writes

**SLIMFAST** introduces a new piece of state to optimize write operations: an **EpochPair** \( \forall t \), for each thread \( t \) consisting of the tuple \((\bot_e, E(t))\), i.e., an empty read epoch and \( t \)'s current write epoch. \( \forall t \) is updated whenever \( t \)'s current epoch is updated, i.e., on a Release or Fork operation. Whenever \( t \) performs a write operation on a location \( x \) that updates \( x \)'s metadata (**FASTTrack** rules **WRITEEXCLUSIVE** or **WRITESHARED**), \( x \)'s metadata can be set directly to \( \forall t \). \( \forall t \) is just a single epoch because all of \( t \)'s writes within a given epoch use the same metadata value of \( \forall t \) (see Figure 5).

#### 4.2 Optimizing **EpochPair** Reads

To optimize **EpochPair** reads, **SLIMFAST** introduces an additional piece of state: a set of **EpochPairs** \( S \) maintained for each thread. A thread \( t \)'s set is written as \( S_t \). Consider the case of \( t \) performing a read that triggers an update of **EpochPair** metadata (the **FASTTrack** **READEXCLUSIVE** rule). In **SLIMFAST**, before \( t \) performs its metadata update, it consults \( S_t \) to determine whether the new metadata value already exists and can be reused from \( S_t \) (**READEXCLEUSE** or whether new metadata needs to be allocated and added to \( S_t \) (**READEXCALLOC**).

#### 4.2.1 Reducing the Size of \( S_t \)

To avoid a situation where \( S_t \) grows too large, we need to be able to remove elements as well as add them. We observe that with **READEXCLEUSE** and **READEXCALLOC** – the only rules that access \( S_t \) – lookups in \( S_t \) only ever use the read epoch equal to \( t \)'s current epoch \( E(t) \). Thus, retaining **EpochPairs** with previous read epochs is unnecessary as these previous **EpochPairs** will never be used in future metadata updates. As shown in Figure 6, no metadata with read epoch \( t \otimes 2 \) will be used for metadata updates once epoch 3 begins.

We thus clear \( S_t \) whenever a new epoch begins, i.e., on **RELEASE** and **FORK** operations. Note that clearing \( S_t \) does not affect the metadata objects referenced by \( S_t \), as they may still be referenced by various program locations and may be needed for future race detection checks.
4.2.2 Optimizing $S_i$ Lookups

All the $S_i$ lookups a thread $t$ performs within a given epoch use the same value for the read epoch. Combined with the fact that $S_i$ only contains EpochPairs inserted during the current epoch (see Section 4.2.1 above), $S_i$ will only ever contain EpochPairs with a read epoch of $E(t)$. Thus, it is unnecessary to use the read epoch as part of the key for $S_i$; instead, $S_i$ can be indexed solely by the write epoch $M^W_x$. Figure 6 illustrates this case: the shaded boxes show that the read epochs used in metadata updates are constant within an epoch. The first access in Epoch 1 does not have a shaded read epoch because writes do not access $S_i$ and instead are set to $\emptyset$.

Removing the read epoch from $S_i$ lookups accelerates the $S_i$ hash function. For clarity we elide this optimization in the SlimFast semantics (Figure 4) but it is used in the implementation.

4.3 Optimizing EpochPlusVC Accesses

Removing redundancy from EpochPlusVC metadata is challenging because of their size: redundancy checks require $O(n)$ time where $n$ is the number of threads. Fortunately, the structure of EpochPlusVC updates can be exploited to eliminate most redundancy in $O(1)$ time.

In SlimFast, EpochPlusVC metadata $M_x$ is extended with an array $M^\text{Next}_x$ of EpochPlusVC references (Figure 7). While the $M^W_x$ and $M^R_x$ fields (and their contents) are immutable, the $M^\text{Next}_x$ array elements are mutable. The use of the $M^\text{Next}_x$ array is explained below.

4.3.1 Optimizing EpochPair to EpochPlusVC Inflations

In SlimFast, each thread maintains a set of EpochPlusVC metadata called $Q_t$. This set is used to remove redundancy for inflations: operations that transition a location’s metadata from EpochPair to EpochPlusVC format. Consider the FastTrack READINFLATE rule, which is used when a concurrent read occurs to a location with EpochPair metadata and the metadata must be inflated to an EpochPlusVC. With SlimFast, $t$ checks $Q_t$ to determine whether existing metadata can be reused (READINFLREUSE) or needs to be allocated and added to $Q_t$ for subsequent reuse (READINFLALLOC). The EpochPlusVC metadata in $Q_t$ upholds several invariants. There are only ever two non-empty entries in the EpochPlusVC vector clock (as can be seen from READINFLALLOC), and one of the non-empty entries is always the current clock for the current thread $t$ because $Q_t$ is cleared on every RELEASE and FORK. Thus, $Q_t$ can be indexed using just the remote read $M^R_x$ and the last write $M^W_x$, allowing for $O(1)$ lookups.

4.3.2 Optimizing EpochPlusVC Updates

If a location’s metadata is already in EpochPlusVC form, SlimFast uses the $M^\text{Next}_x$ array to provide $O(1)$ redundancy checks. We exploit the insight that, for any given EpochPlusVC read that triggers an update (FastTrack’s READSHARED rule), a thread $t$ only updates its own entry of the vector clock. In SlimFast, the EpochPlusVC metadata $M_x$ for a location $x$ is extended with an EpochPlusVC array $M^\text{Next}_x$ that maps from $M_x$ to an updated EpochPlusVC $M'_x$. Each element $M^\text{Next}_x(t)$ is a mapping for thread $t$, where the

\[
\begin{align*}
M^R_x &= E(t) \\
& (C, L, M) \Rightarrow \text{ev}(t, x) (C, L, M) \\
M_x \in \text{EpochPlusVC} \\
M^W_x \leq C_t \\
M^R_x[t] &= C(t) \\
M' &= M[x := (M^R_x[t := C(t)], M^W_x)] \\
& (C, L, M) \Rightarrow \text{ev}(t, x) (C, L, M') \\
M_x \in \text{EpochPair} \\
M^R_x \leq C_t \\
M^W_x \leq C_t \\
M' &= M[x := (M^R_x, M^W_x)] \\
& (C, L, M) \Rightarrow \text{ev}(t, x) (C, L, M') \\
\text{READINFLATE} & \Rightarrow (C, L, M) \\
M_x \in \text{EpochPair} \\
M^R_x \leq C_t \\
M^W_x \leq C_t \\
M' &= M[x := (E(t), M^W_x)] \\
& (C, L, M) \Rightarrow \text{ev}(t, x) (C, L, M') \\
\text{READEXCLUSIVE} & \Rightarrow (C, L, M) \\
M_x \in \text{EpochPair} \\
M^R_x \leq C_t \\
M^W_x \leq C_t \\
M' &= M[x := (E(t), M^W_x)] \\
\text{WRITESAMEEpoch} & \Rightarrow (C, L, M) \\
M^W_x = E(t) \\
& (C, L, M) \Rightarrow \text{ew}(t, x) (C, L, M) \\
M_x \in \text{EpochPair} \\
M^R_x \leq C_t \\
M^W_x \leq C_t \\
M' &= M[x := (\langle L, E(t) \rangle)] \\
& (C, L, M) \Rightarrow \text{ew}(t, x) (C, L, M') \\
M_x \in \text{EpochPlusVC} \\
M^R_x \subseteq C_t \\
M^W_x \leq C_t \\
M' &= M[x := (\langle L, E(t) \rangle)] \\
& (C, L, M) \Rightarrow \text{ew}(t, x) (C, L, M') \\
\text{WRITEEXCLUSIVE} & \Rightarrow (C, L, M) \\
M_x \in \text{EpochPair} \\
M^R_x \leq C_t \\
M^W_x \leq C_t \\
M' &= M[x := (\langle L, E(t) \rangle)] \\
& (C, L, M) \Rightarrow \text{ew}(t, x) (C, L, M') \\
\text{WRITESHARED} & \Rightarrow (C, L, M) \\
\text{FORK} & \Rightarrow (C, L, M) \\
\text{JOIN} & \Rightarrow (C, L, M) \\
\text{ACQUIRE} & \Rightarrow (C, L, M) \\
\text{RELEASE} & \Rightarrow (C, L, M) \\
\end{align*}
\]
Figure 4: SlimFast operational semantics. Shading indicates differences from corresponding FastTrack rule.

only difference between $M_x$ and $M_x^{\text{Next}}(t)$ is that the latter has an updated vector clock entry for $t$.

When a thread $t$ reads from a location $x$ with EpochPlusVC metadata that requires updating, SLIMFAST checks whether the required EpochPlusVC is already referenced from $M_x^{\text{Next}}(t)$. In our semantics, $M_x^{\text{Next}}(t)(t)$ extracts the clock value for $t$ from $M_x^{\text{Next}}(t)$. If this is equal to $t$’s current clock (READSHAREUSE), then $M_x^{\text{Next}}(t)$ provides the desired EpochPlusVC value directly. Otherwise (READSHAREALLOC), a new EpochPlusVC is allocated and written into $M_x^{\text{Next}}$ to facilitate future reuse.

The price of $O(1)$ redundancy checking is sub-optimal redundancy elimination, as Figure 8 shows. The initial reads of $x$ and $y$ result in EpochPair metadata. Each thread’s second access triggers inflation, but each thread’s $Q$ set is empty, so two pieces of EpochPlusVC metadata are allocated with identical values (READINFLALLOC). Detecting such redundancy would require some kind of global, synchronized map where the costs of access would quickly outweigh the benefits of redundancy elimination. To’s read of $z$ reuses the EpochPair metadata (READEXLUSE). $T1$’s read of $z$ is able to reuse a previously-allocated EpochPlusVC from $Q_{T1}$ (READINFLUSE). At the end of this program, some EpochPlusVC redundancy has been eliminated: all three locations have identical metadata but only two EpochPlusVCs are needed to represent this.

<table>
<thead>
<tr>
<th>T1</th>
<th>operation</th>
<th>per-location</th>
<th>EpochPair</th>
</tr>
</thead>
<tbody>
<tr>
<td>acquire L</td>
<td>[ - , t1@1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write X</td>
<td>[ - , t1@1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write Y</td>
<td>[ - , t1@1]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T1</th>
<th>operation</th>
<th>per-location</th>
<th>EpochPair</th>
</tr>
</thead>
<tbody>
<tr>
<td>acquire L</td>
<td>[ - , t1@2]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write X</td>
<td>[ - , t1@2]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write Y</td>
<td>[ - , t1@2]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Lemma 1. Given a trace \( \alpha \), \( \sigma_0 \xrightarrow{\alpha} (C, L, M) \) in FASTTrack if and only if \( \sigma_0 \xrightarrow{\alpha} (C, L, M, S, Q, W) \) in SLIMFast for some \( S, Q \) and \( W \).

Proof. By induction on \( \alpha \). For each operation, a FASTTrack rule can be applied if and only if a SLIMFast rule can be applied that makes exactly the same changes to \( C, L, M \), maintaining the invariant that \( \mathbb{W}_t = (\bot_e, E(t)) \) for each thread \( t \).

Theorem 2. Suppose \( \alpha \) is a feasible trace. Then \( \alpha \) is race-free if and only if \( \exists \sigma \) such that \( \sigma_0 \xrightarrow{\alpha} \sigma \) in SLIMFast.

Proof. This follows directly from Lemma 1 and the correctness theorem of FASTTrack.

4.5 Per-thread Epoch Sets are Optimal

Maintaining per-thread EpochPair sets \( S_t \) may appear at first to be sub-optimal in terms of eliminating redundant metadata, as two threads \( t \) and \( u \) may insert metadata with the same value into \( S_t \) and \( S_u \), respectively, creating redundancy across different epoch sets. Fortunately, it can be shown that any two sets \( S_t \) and \( S_u \) are always disjoint and thus incur no additional space overhead over a global epoch set.

Theorem 3 (5 Redundancy Elimination). \( S_t \cap S_u = \emptyset \), for any two threads \( t \) and \( u \), at every point in their execution.

Proof. By contradiction: assume that \( S_t \cap S_u \neq \emptyset \) at some point in the execution. Thus there is some EpochPair \( e \) in the intersection. Thus, two metadata instances equivalent to \( e \) must have been constructed by two threads \( t \) and \( u \).

\( e \) must have been allocated by the READEXCLALLOC, and \( e \) contains some read epoch \( c \in v \). However, all EpochPairs allocated by this rule have a read epoch equal to \( E(v) \), which contains a thread identifier for the allocating thread \( v \). A thread \( u \) will never allocate an EpochPair containing another thread's read epoch. Thus, the epoch \( e \) can only have been allocated by thread \( v \) (where \( v \) must be equal to \( t \) or \( u \)), and \( e \) can only live in \( S_v \).

Thus, \( S_t \cap S_u = \emptyset \) always holds and there is no redundancy lost due to per-thread sets.

Similar reasoning shows that for any two threads \( t \) and \( u \), \( \mathbb{W}_t \neq \mathbb{W}_u \) as each thread only ever constructs its \( \mathbb{W} \) using its own thread id (in RELEASE or FORK). Thus there is no redundancy among the \( \mathbb{W} \) EpochPairs, either.

5. Enforcing Rule Atomicity

The semantics presented in Section 4 and Figure 4 implicitly rely on the fact that each rule executes atomically. Because SLIMFast metadata is immutable, the SLIMFast implementation relies on atomic operations instead of locking, similar in spirit to Linux’s RCU mechanism [41], where updates are performed on local state and then made visible via an atomic operation. In this section we describe an algorithm to provide atomicity for SLIMFast rules using atomic operations and prove that this algorithm is correct.

5.1 Metadata State Transitions

Figure 9 shows the pseudocode for the SLIMFast metadata update algorithm. We discuss the correctness of the cases...
START: Metadata e = M_x

// Case 1: no update
if Read Same Epoch or
  Write Same Epoch or
  Read Shared Same Epoch:
  return

// Case 2: update
else:
  Metadata e′ = updateMetadata(e)
  if !CAS(&M_x, e => e′): goto START
  return

Figure 9: SlimFast’s metadata update algorithm.

in SlimFast’s metadata update algorithm below, and provide a formal argument in Section 5.2. We observe that the mutable elements of $M^\text{Next}_x$ are thread-private, as $M^\text{Next}_x[t]$ is only ever accessed by thread $t$ (in the READSHARED* rules). Thus, operations on $M^\text{Next}_x$ are trivially atomic.

Case 1: No updates Rules that do not update metadata (READSAMEÉPOCH, WRITESAMEÉPOCH, READSHAREDSAMEÉPOCH) do not need synchronization because metadata is immutable.

Case 2: Update For the remaining rules that do update metadata, the updateMetadata function encapsulates the logic of the corresponding rule. After the updated metadata is calculated, the CAS operation on line 12 is used to update the metadata for $x$ via the pointer &M_x. The CAS ensures that the rule executes atomically by verifying that the metadata has not changed during the duration of the rule’s execution. If the CAS operation fails, then there may have been concurrent updates that have changed the metadata state and so the update restarts from line 1.

As with other CAS-based algorithms, we must take care to avoid ABA issues [42] that can lead to an inadvertently successful CAS operation. There are two classes of ABA issues to consider: memory reuse and direct metadata reuse. Memory reuse issues do not arise in SlimFast as it is implemented in a garbage-collected language (Java) and the local read of $M_x$ on line 1 prevents $M_x$ from becoming garbage and getting reallocated as another object with the same address. Direct metadata reuse is subtler, following from the monotonicity of metadata updates (see Lemma 4).

5.2 Correctness of Atomicity

In this section, we outline the proof that the algorithm of Figure 9 guarantees atomicity of metadata updates. First, we prove that the ABA issues described in Section 5.1 do not arise, by showing that once a metadata is modified it never returns to its original value.

Lemma 4 (Monotonicity). Let $M$ be the metadata for a location $x$, and suppose $M$ is updated to some $N \neq M$.

Then there is no sequence of updates that can be applied to $N$ that will result in the original state $M$.

Proof is available in the supplemental material. Using Lemma 4, we can prove that the update algorithm guarantees atomicity. A CAS failure causes an update to restart if another update has occurred simultaneously.

Theorem 5 (Atomicity). Let $a$ be an update that changes the metadata for a location $x$. Then $a$ either completes before any other changes are made to $x$’s metadata, or restarts (thus waiting until after other changes).

Proof. The update $a$ writes only at the CAS operation at line 12, which fails if the metadata being updated is anything other than the original $M$. If $x$’s metadata is $M$ at the time of the CAS, then either the metadata has not been changed (as desired) or else two or more operations have written to the metadata, changing it to some other state and then back to $M$. However, Lemma 4 rules out the latter case. Thus, if $a$ successfully executes line 12, it must have completed before any other update wrote to $x$’s metadata.

6. Implementation

SlimFast is implemented using the RoadRunner framework [24], on which FastTrack is also implemented. The RoadRunner framework provides the shadow-memory implementation that SlimFast uses to store its metadata.

6.1 EpochPair and EpochPlusVC

SlimFast uses the EpochPair and EpochPlusVC classes to store its metadata. In our implementation, EpochPlusVC extends EpochPair with a vector clock, an array of references to EpochPlusVCs (Figure 7), and the methods to manipulate it. On each variable access, SlimFast uses dynamic dispatch to call the correct method to check and update the metadata for that variable. This implementation wastes some space in EpochPlusVC due to an unnecessary read epoch field that is inherited from EpochPair. However, this implementation was slightly faster in practice than the alternative, space-efficient implementation in which EpochPair and EpochPlusVC inherit from a common base class.

6.2 Storing and Retrieving Immutable Metadata

As described in Section 4, SlimFast maintains per-thread sets $S_i$ of immutable EpochPairs and $Q_i$ of EpochPlusVCs. We noticed that in practice these sets are very small (see Section 7.4), so we adopt a simple array-based implementation for these sets with a 20-element fixed-size array. With a fixed-size array, overflow is possible. In such cases, we wrap around and overwrite the initial array entries. This overwriting does not affect correctness, but it does compromise Theorem 3 (redundancy elimination for EpochPairs) because redundancy cannot be detected for the overwritten EpochPair. In practice, overflow never occurred in any of the 24 benchmarks we ran.
7. Evaluation

The main goal of our evaluation is to measure how well SLIMFAST achieves its goal of reducing the space and runtime overheads of dynamic data race detection, and to examine how SLIMFAST scales as the thread counts increase.

7.1 Experimental Setup

We use the DaCapo 2009 [3], Java Grande [12, 40], and NAS Parallel Benchmark 3.0.3 [27] suites to evaluate SLIMFAST. ROADRUNNER is incompatible with the h2, eclipse, daytrader, tradebeans in the DaCapo suite so we exclude them from our evaluation. We use the largest provided inputs for the DaCapo and Grande programs, and the A inputs for NPB.CG, NPB.MG, NPB.ET, NPB.IS, and the W inputs for NPB.BT, NPB.LU, NPB.SP, from the NAS benchmarks. We used ROADRUNNER version v0.3 with the fast-path optimization enabled.

We use the lock-based version of FASTTRACK from ROADRUNNER v0.3. While a CAS-based implementation of FASTTRACK also exists, it supports only 24-bit clock values which suffer from rollover on some benchmarks [2]. We found the CAS-based FASTTRACK provided equivalent performance to the lock-based version on our benchmarks, and was in some cases slower because CAS operations are not particularly efficient on our experimental machine. SLIMFAST uses 32-bit clocks to avoid rollover, and we modify FASTTRACK likewise.

All experiments were performed on a quad-socket machine consisting of four 2.0 GHz Intel Xeon E7-4820 (Westmere) chips (8 cores/16 threads each) with 128GB of RAM, running 64-bit Linux 3.11.10. All code was compiled with JDK 1.7.0.25 and run on HotSpot 64-Bit Server VM 23.25-b01, with a heap size of 64GB and the default Parallel Scavenger collector. Our results are for running each benchmark with 16 threads, unless specified otherwise or the benchmark only supports a fixed number of threads (avrora, batik, jpy, luindex, pmd, and tomcat).

7.2 Memory Savings

We measured the total memory usage of SLIMFAST, FASTTRACK, and ROADRUNNER by forcing a garbage collection every 100ms and recording the heap usage after each collection. This measures the entire heap, including both application data and race detector state. Figure 10 shows the average space reduction of SLIMFAST over FASTTRACK. Among all 24 benchmarks we ran, SLIMFAST consumes 1.76x less memory on average than FASTTRACK, with up to 4.47x savings on NPB.IS. Overall, SLIMFAST’s space reduction is highly correlated with the metadata redundancy ratio of each benchmark (Figure 2).

SLIMFAST consumes more space than FASTTRACK for avrora and NPB.BT because these benchmarks have relatively low redundancy ratios: avrora has the lowest among all our benchmarks. For avrora and NPB.BT, SLIMFAST’s EpochPlusVC space optimizations (principally the $M_{next}$ array inside each EpochPlusVC object) backfire and increase space usage mildly. When using a version of SLIMFAST with mutable EpochPlusVCs, SLIMFAST has less heap usage than FASTTRACK on all benchmarks.

7.3 Performance

To compare the performance of SLIMFAST and FASTTRACK, we ran each configuration 15 times and recorded the average runtime of each configuration. We used a large heap (64GB) so that no garbage collection, except the one ROADRUNNER always triggers upon start-up, occurred during the experiments.

The orange bars in Figure 11 illustrate the average speedup of SLIMFAST over FASTTRACK. For 12 programs, SLIMFAST outperforms FASTTRACK, with speedups of up to 8.33x (sparsemat); for 5 programs, SLIMFAST runs slower than FASTTRACK, with the overhead ranging from 6.2% (batik) to 27.7% (tomcat); for the rest of the programs, the difference in runtime is negligible between SLIMFAST and FASTTRACK. Overall, the geometric mean of speedups of SLIMFAST over FASTTRACK is 1.24x.

According to additional experiments we have done, the speedups SLIMFAST gains can mainly be attributed to 2 reasons. First, SLIMFAST’s space reduction shrinks the working set of the instrumented programs, bringing better cache behavior. Second, SLIMFAST’s immutable EpochPlusVCs reduce contention over metadata, especially on sunflow and sparsemat which have many EpochPlusVC accesses.

Why SLIMFAST is sometimes slower than FASTTRACK is also an interesting question. For avrora, lusearch, and xalan, the brown bars in Figure 11 show the speedup of a version of SLIMFAST with mutable EpochPlusVCs. These bars show that the costs of immutable EpochPlusVCs sometimes outweigh their benefits, due to the relatively low redundancy ratios (Figure 2) and low percentages of reuse in EpochPlusVCs objects (Table 1) in these benchmarks. For tomcat, because its worker threads are highly independent of each other with little sharing, SLIMFAST’s positive effects on synchronization and working set does not translate into speedup.

To measure how the performance of SLIMFAST scales as the number of threads increase, we ran benchmarks supporting a configurable number of threads with 2, 4, 8, 16 threads, with SLIMFAST and FASTTRACK. Figure 12 shows the speedup that SLIMFAST and FASTTRACK demonstrate with more threads (all SLIMFAST bars are normalized to SLIMFAST with 2 threads, and all FASTTRACK bars to FASTTRACK with 2 threads). On almost all benchmarks, SLIMFAST shows better scalability than FASTTRACK; the most drastic difference appears on sparsemat and sunflow where FASTTRACK does not scale at all while the performance of SLIMFAST keeps improving with more threads. This is a result of SLIMFAST’s more efficient metadata syn-
chronization which copes well with contention on accesses to (mainly *EpochPlusVC*) metadata.

### 7.4 Additional Characterization

To better understand **SLIMFAST**’s performance, we performed a series of characterization experiments; Table 1 shows the results. Column 2 shows the slowdown of **ROADRUNNER**, **FASTTRACK**, and **SLIMFAST**, normalized to native JVM execution. The slowdown of **ROADRUNNER** ranges from 1.3x to 30.1x, which accounts for a large part of the slowdown of **FASTTRACK** and **SLIMFAST**. A more efficient framework for dynamic analysis would likely reduce this overhead substantially. Columns 3 & 4 show the percentage of accesses to **EpochPlusVCs**, among all accesses, and the percentage of **EpochPlusVCs** allocations, among all metadata allocations. These two indicators are highly correlated with the speedup of **SLIMFAST** over **FASTTRACK**: for instance, fop and jython have no **EpochPlusVCs** (they’re mostly single-threaded), and the speedup of **SLIMFAST** on them is small. In contrast, most of the metadata objects accessed in sunflow and sparsemat are **EpochPlusVCs** where **SLIMFAST** shows significant speedups.

Columns 5 & 6 show the average occupancy of the \( S_t \) and \( Q_t \) sets: for all the benchmarks, these sets have \( \leq 11 \) elements on average. This motivates our implementation of \( S_t \) and \( Q_t \) with fixed-size arrays. Columns 7 & 8 give the percentages of reuse of **EpochPlusVCs** and **EpochPairs** among the total lookups to \( S_t \) and \( Q_t \), respectively. Higher percentages mean that fewer **EpochPairs** and **EpochPlusVCs** need to be allocated, saving space. All 24 benchmarks have high percentages of **EpochPairs** reuse (the lowest, luindex, is 82.8%), showing that **EpochPairs** are highly redundant. In
contrast, the percentage of EpochPlusVC's reuse is low on some benchmarks, showing that the redundancy among vector clocks is less significant.

Column 9 in Table 1 shows the maximum heap usage with SLIMFAST. Note that SLIMFAST both saves space and reduces runtime more effectively on programs with larger heaps. Thus, as program working sets continue to grow we expect SLIMFAST's benefits to become even more pronounced.

The final column in Table 1 shows the memory overhead of SLIMFAST with respect to ROADRUNNER: most programs incur less than 51% overhead, except NPB_LU (124.03%) and sparsemat (79.25%). The relatively small spatial overhead of SLIMFAST may make precise dynamic data race detection more feasible in memory-constrained systems, such as mobile devices.

Although SLIMFAST guarantees optimal redundancy elimination for EpochPairs (Theorem 3), such a guarantee does not extend to EpochPlusVCs. We measured the redundancy ratio on EpochPlusVCs in SLIMFAST, and the results are shown in Figure 13. Somewhat surprisingly, on most benchmarks, essentially no redundancy exists on EpochPlusVCs, which means SLIMFAST does in fact gain near-optimal space reduction on these programs. On lufact and montecarlo, about \( \frac{2}{3} \) of EpochPlusVCs are redundant. This is a relatively low ratio, and EpochPlusVCs represent only about 60% of the allocated metadata for these workloads, further shrinking the opportunity for meaningful space reduction. Set against the \( O(n) \) cost of doing optimal redundancy elimination, SLIMFAST strikes a good balance of finding most of the redundancy in just \( O(1) \) time.

Finally, we explore how applicable the SLIMFAST idea is to other data race detection algorithms. We have implemented in ROADRUNNER a prototype of the SPD3 race detector for fork-join programs [53], and SLIMFAST-SPD3, a tool that applies the SLIMFAST idea to SPD3. Since SPD3 supports only limited forms of synchronization, we run it only on the Grande programs.

Figure 14 shows the average space reduction of SLIMFAST-SPD3 over SPD3. The number atop each bar shows the redundancy ratio of metadata in SPD3. Among all the benchmarks measured, SLIMFAST-SPD3 is able to gain more than 2x space reduction, with a max reduction of 4.27x and a geometric mean reduction of 3.07x. This result shows that SLIMFAST is applicable to multiple race detection algorithms and, perhaps, to a broad swathe of other concurrency analyses as well.

### 8. Related Work

While many dynamic data race detection algorithms have been proposed [17, 51, 60], the most closely related work to SLIMFAST is the FASTTrack system [22, 23], which represents the state-of-the-art in dynamic data race detection. We provide an extensive comparison of our approach with the FASTTrack algorithm in Section 4, and a memory usage and performance comparison in Section 7.

The SlimState system [63] seeks to reduce metadata re-
The RedCard system [25] provides two static analyses of race checks in complementary manner to SFLAG's inherently more precise tracking, its performance results are roughly comparable with SlimState. SlimState is faster on crypt, montecarlo and lusearch while SLIMFAST is faster on lufact and (by 8x) on sparsematmult.

The RedCard system [25] provides two static analyses to accelerate dynamic race detection. The first analysis removes redundant race checks to a single location within a release-free span of code, targeting the computational cost of race checks in complementary manner to SLIMFAST's focus on metadata space usage. RedCard's second analysis merges the metadata for two memory locations that are always accessed together in a release-free span. This analysis targets metadata space usage, as SLIMFAST does, but is subject to the general conservatisms of static analysis. For the Java Grande benchmarks used in both the RedCard evaluation and ours, SLIMFAST provides substantially increased space savings – RedCard saves no space on lufact, montecarlo, series, sor, or sparsematmult while SLIMFAST reduces the heap by at least 2x on each. RedCard reduces metadata alone by 1.5x on moldyn and 7x on crypt, while SLIMFAST reduces metadata by 300,000x and 12x, respectively.

Accordion clocks [10] reduce metadata space usage by removing vector clock entries for terminated threads. FAST-TRACK makes the use of vector clocks infrequent and thus dilutes the benefit of accordion clocks, though they remain complementary to SLIMFAST as SLIMFAST does not reduce the space consumption of vector clocks.

Several forms of sampling-based dynamic race detection have been proposed. Such schemes trade off soundness [6, 16, 19, 28, 38] for reduced performance overheads. In contrast, SLIMFAST reduces the memory and performance overheads of data race detection without sacrificing soundness or completeness.

**Lockset-based race detection** [15, 58], an alternative to the happens-before data race detection algorithm, reports false races on some common programming idioms like privatization but can also detect with a single execution some races that would require multiple executions with happens-before. Other work has generalized happens-before race detection to detect more races from a single execution [9, 59, 62] at the cost of decreased performance.

Several systems exist for detecting data races in structured parallel programs such as fork-join programs [20, 32, 36, 43], async-finish programs [54] or programs with asynchronous callbacks [30, 50, 55]. Structured parallelism admits more time- and space-efficient data race detection algorithms than the general multithreaded programs we target. However, none of these algorithms exploit metadata redundancy and thus a SLIMFAST-like approach may be beneficial in structured parallelism contexts as well.

There have been several proposals for static race detection [18, 48], and such systems can serve as a complement to SLIMFAST by pruning race detection instrumentation and metadata at compile time. Others have proposed type systems [1, 4] and implicitly-parallel languages [29, 56] that eliminate data races by construction, though these systems sacrifice expressiveness to obtain race-freedom guarantees.

There have also been several proposals of hardware support for data race detection. These systems typically sacrifice soundness and completeness in favor of simpler hardware [31, 44, 47, 52], though sound and complete hardware race detection has also been proposed [14].

Finally, there is a rich literature on detecting concurrency bugs other than races such as atomicity violations [11, 21, 26, 33], sequential consistency violations [35, 39, 46], ordering violations [34, 61] and determinacy violations [8]. Several of these analyses use data race detection as a pre-processing step, and thus can directly benefit from the improvement provided by SLIMFAST.

**9. Conclusion**

Data races are a key contributor to the complexity of multi-threaded programming. Existing dynamic race detectors incur unnecessarily high space overheads by storing many redundant copies of metadata. The SLIMFAST race detector eliminates the vast majority of this redundancy by maintaining only a single, immutable instance of each metadata value and sharing references to this instance. We leverage several novel invariants of dynamic race detection that make identifying redundant metadata efficient. As a result, SLIMFAST offers reduced memory usage and runtime compared to the current state of the art.
References


[51] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In Proceed-


