FastTrack: Efficient and Precise Dynamic Race Detection

Cormac Flanagan
Computer Science Department
University of California at Santa Cruz
Santa Cruz, CA 95064

Stephen N. Freund
Computer Science Department
Williams College
Williamstown, MA 01267

Abstract

Multithreaded programs are notoriously prone to race conditions. Prior work on dynamic race detectors includes fast but imprecise race detectors that report false alarms, as well as slow but precise race detectors that never report false alarms. The latter typically use expensive vector clock operations that require time linear in the number of program threads.

This paper exploits the insight that the full generality of vector clocks is unnecessary in most cases. That is, we can replace heavyweight vector clocks with an adaptive lightweight representation that, for almost all operations of the target program, requires only constant space and supports constant-time operations. This representation change significantly improves time and space performance, with no loss in precision.

Experimental results on Java benchmarks including the Eclipse development environment show that our FASTTRACK race detector is an order of magnitude faster than a traditional vector-clock race detector, and roughly twice as fast as the high-performance DJIT$^+$ algorithm. FASTTRACK is even comparable in speed to ERASER on our Java benchmarks, while never reporting false alarms.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification–*reliability*; D.2.5 [*Software Engineering*]: Testing and Debugging–*monitors, testing tools*; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Algorithms, Verification

Keywords Race conditions, concurrency, dynamic analysis

1. Introduction

Multithreaded programs are notoriously prone to race conditions and other concurrency errors, such as deadlocks and atomicity violations. The widespread adoption of multi-core processors only exacerbates these problems, both by driving the development of increasingly-multithreaded software and by increasing the interleaving of threads in existing multithreaded systems.

A race condition occurs when a program's execution contains two accesses to the same memory location that are not ordered by the happens-before relation [21], where at least one of the accesses is a write. Race conditions are particularly problematic because they typically cause problems only on certain rare interleavings, making them extremely difficult to detect, reproduce, and eliminate. Consequently, much prior work has focused on static [1, 5, 3, 19, 4, 12, 15, 26, 40] and dynamic [33, 38, 27, 42, 30, 11, 34, 42, 30] analysis tools for detecting race conditions.

In general, dynamic race detectors fall into two categories, depending on whether they report false alarms. *Precise* race detectors never produce false alarms. Instead, they compute a precise representation of the happens-before relation for the observed trace and report an error if and only if the observed trace has a race condition. Typically, the happens-before relation is represented using *vector clocks* (VCs) [23], as in the DJIT$^+$ race detector [29, 30]. Vector clocks are expensive, however, because they record information about each thread in a system. Thus, if the target program has n threads, then each VC requires $O(n)$ storage space and each VC operation requires $O(n)$ time.

Motivated in part by the performance limitations of vector clocks, a variety of alternative *imprecise* race detectors have been developed, which may provide better coverage but can report false alarms on race-free programs. For example, Eraser's LockSet algorithm [33] enforces a lock-based synchronization discipline and reports an error if no lock is consistently held on each access to a particular memory location. Eraser may report false alarms, however, on programs that use alternative synchronization idioms such as fork-join or barrier synchronization. Some LockSet-based race detectors include happens-before reasoning to improve precision in such situations [42, 28]. MultiRace [29, 30] leveraged this combination of techniques to improve performance as well. That analysis, and others [42], also group multiple memory locations into *mini-pages* to improve performance, again at some cost in precision.

A primary limitation of both static race detectors and imprecise dynamic race detectors is the potential for a large number of false alarms. Indeed, it has proven surprisingly difficult and time consuming to identify the real errors among the spurious warnings produced by some tools. Even if a code block looks suspicious, it may still be race-free due to some subtle synchronization discipline that is not (yet) understood by the current programmer or code maintainer. Even worse, additional real bugs (*e.g.*, deadlocks) could be added while attempting to "fix" a spurious warning produced by these tools. Conversely, real race conditions could be ignored because they appear to be false alarms. Precise (albeit incomplete) race detectors avoid these issues, but such detectors are limited by the performance overhead of vector clocks.

This paper exploits the insight that, while vector clocks provide a general mechanism for representing the happens-before relation, their full generality is not actually necessary in most cases. Indeed, the vast majority of data in multithreaded programs is either thread local, lock protected, or read shared. Our FASTTRACK analysis uses an adaptive representation for the happens-before relation to provide constant-time fast paths for these common cases, without any loss of precision or correctness in the general case.

PLDI'09, June 15–20, 2009, Dublin, Ireland.

Figure 1: Multithreaded Program Traces

$$
\begin{array}{rcl}
\alpha \in \mathit{Trace} & = & \mathit{Operation}^* \\
a, b \in \mathit{Operation} & = & rd(t,x) \mid wr(t,x) \\
 & & \mid acq(t,m) \mid rel(t,m) \\
 & & \mid fork(t,u) \mid join(t,u) \\
s, u, t \in \mathit{Tid} & \quad x, y \in \mathit{Var} & \quad m \in \mathit{Lock}
\end{array}
$$

In more detail, a VC-based race detector such as DJIT$^+$ records the clock of the most recent write to each variable x by each thread t. By comparison, FASTTRACK exploits the observation that all writes to x are totally ordered by the happens-before relation (assuming no races detected so far), and so it records information only about the *very last* write to x, specifically, the clock and thread identifier of that write. We refer to this pair of a clock and a thread identifier as an *epoch*.

Read operations on thread-local and lock-protected data are also totally ordered (assuming no races have been detected) and so FASTTRACK records only the epoch of the last read to such data. FASTTRACK adaptively switches from epochs to vector clocks where necessary (for example, when data becomes read-shared) in order to guarantee no loss of precision. It also switches from vector clocks back to lightweight epochs where possible (for example, when read-shared data is subsequently updated).

Using these adaptive representation techniques, FASTTRACK reduces the analysis overhead of almost all monitored operations from $O(n)$-time (where n is the number of threads in the target program) to $O(1)$-time, via lightweight, constant-time fast paths. Note that this performance improvement involves no precision loss.

In addition to improving performance, our epoch representation also reduces space overhead. A VC-based race detector requires an $O(n)$ space overhead for each memory location of the target program and can quickly exhaust memory resources. By comparison, FASTTRACK reduces the space overhead for thread-local and lock-protected data from $O(n)$ to $O(1)$.

For comparison purposes, we have developed implementations of six different dynamic race detectors:

- ERASER [33], a well-known imprecise race detector.
- GOLDILOCKS, a precise race detector based on an extended notion of "LockSets" [14].
- BASICVC, a traditional VC-based race detector.
- DJIT$^+$, a high-performance VC-based race detector [30].
- MULTIRACE, a hybrid LockSet/DJIT$^+$ race detector [30].
- FASTTRACK, the algorithm presented in this paper.

These tools are all implemented on top of the same framework for dynamic analysis of multithreaded Java software, and the VC-based tools use the same optimized vector clock primitives, thus providing a true "apples-to-apples" comparison. Our experimental results on several Java benchmarks including the Eclipse development environment [13] show that FASTTRACK outperforms these other tools. For example, it provides almost a 10x speedup over BASICVC and a 2.3x speedup even over the DJIT$^+$ algorithm. It also provides a substantial increase in precision over ERASER, with no loss in performance.

In summary, the main contributions of FASTTRACK are:

- It provides a significant improvement in precision over earlier, imprecise race detectors such as Eraser [33], while providing comparable performance.
- Despite its efficiency, it is still a comparatively simple algorithm that is straightforward to implement, as illustrated in Figure 5.
- It uses an adaptive lightweight representation for the happens-before relation that reduces both time and space overheads.
- It contains optimized constant-time fast paths that handle upwards of 96% of the operations in benchmark programs.
- It provides a 2.3x performance improvement over the prior DJIT$^+$ algorithm, and typically incurs less than half the memory overhead of DJIT$^+$.
- FASTTRACK also improves the performance of more heavyweight dynamic analysis tools by identifying millions of irrelevant, race-free memory accesses that can be ignored. It provides a 5x speedup for the VELODROME dynamic atomicity checker [17] and an 8x speedup for the SINGLETRACK determinism checker [32].

The presentation of our results proceeds as follows. The following section reviews preliminary concepts and notation, as well as the DJIT$^+$ algorithm. Section 3 presents the FASTTRACK algorithm. Section 4 describes our prototype implementation of this algorithm, and Section 5 presents our experimental results. Section 6 concludes with a discussion of related work.

2. Preliminaries

2.1 Multithreaded Program Traces

We begin by formalizing the notions of execution traces and race conditions. A program consists of a number of concurrently executing threads, each with a thread identifier $t \in \mathit{Tid}$, and these threads manipulate variables $x \in \mathit{Var}$ and locks $m \in \mathit{Lock}$. (See Figure 1.) A *trace* α captures an execution of a multithreaded program by listing the sequence of operations performed by the various threads. The set of operations that a thread t can perform include:

- $rd(t,x)$ and $wr(t,x)$, which read and write a value from x;
- $acq(t,m)$ and $rel(t,m)$, which acquire and release a lock m;
- $fork(t,u)$, which forks a new thread u; and
- $join(t,u)$, which blocks until thread u terminates.

The *happens-before relation* $<_\alpha$ for a trace α is the smallest transitively-closed relation over the operations[1] in α such that the relation $a <_\alpha b$ holds whenever a occurs before b in α and one of the following holds:

- **Program order**: The two operations are performed by the same thread.
- **Locking**: The two operations acquire or release the same lock.
- **Fork-join**: One operation is $fork(t,u)$ or $join(t,u)$ and the other operation is by thread u.

If a happens before b, then it is also the case that b *happens after* a. If two operations in a trace are not related by the happens-before relation, then they are considered *concurrent*. Two memory access *conflict* if they both access (read or write) the same variable, and at least one of the operations is a write. Using this terminology, a trace has a *race condition* if it has two concurrent conflicting accesses.

We restrict our attention to traces that are *feasible* and which respect the usual constraints on forks, joins, and locking operations, *i.e.*, (1) no thread acquires a lock previously acquired but not released by a thread, (2) no thread releases a lock it did not previ-

[1] In theory, a particular operation a could occur multiple times in a trace. We avoid this complication by assuming that each operation includes a unique identifier, but, to avoid clutter, we do not include this unique identifier in the concrete syntax of operations.

ously acquire, (3) there are no instructions of a thread u preceding an instruction $fork(t, u)$ or following an instruction $join(t, u)$, and (4) there is at least one instruction of thread u between $fork(t, u)$ and $join(v, u)$.

2.2 Review: Vector Clocks and the DJIT$^+$ Algorithm

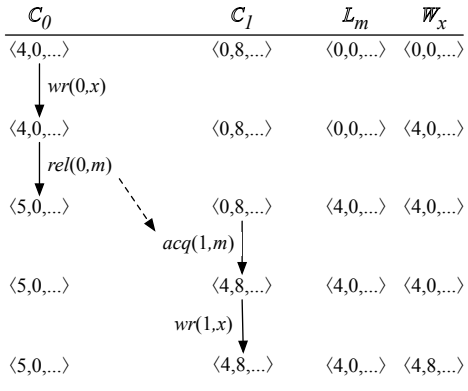
Before presenting the FASTTRACK algorithm, we briefly review the DJIT$^+$ race detection algorithm [30], which is based on vector clocks [23]. A vector clock $VC : Tid \rightarrow Nat$ records a clock for each thread in the system. Vector clocks are partially-ordered ($\sqsubseteq$) in a point-wise manner, with an associated join operation ($\sqcup$) and minimal element ($\bot_V$). In addition, the helper function inc_t increments the t-component of a vector clock:

$$\begin{array}{rcl} V_1 \sqsubseteq V_2 & \text{iff} & \forall t.\ V_1(t) \leq V_2(t) \\ V_1 \sqcup V_2 & = & \lambda t.\ max(V_1(t), V_2(t)) \\ \bot_V & = & \lambda t.\ 0 \\ inc_t(V) & = & \lambda u.\ \texttt{if}\ u = t\ \texttt{then}\ V(u) + 1\ \texttt{else}\ V(u) \end{array}$$

In DJIT$^+$, each thread has its own clock that is incremented at each lock release operation. Each thread t also keeps a vector clock $\mathbb{C}_t$ such that, for any thread u, the clock entry $\mathbb{C}_t(u)$ records the clock for the last operation of thread u that happens before the current operation of thread t. In addition, the algorithm maintains a vector clock $\mathbb{L}_m$ for each lock m. These vector clocks are updated on synchronization operations that impose a happens-before order between operations of different threads. For example, when thread u releases lock m, the DJIT$^+$ algorithm updates $\mathbb{L}_m$ to be $\mathbb{C}_u$. If a thread t subsequently acquires m, the algorithm updates $\mathbb{C}_t$ to be $\mathbb{C}_t \sqcup \mathbb{L}_m$, since subsequent operations of thread t now happen after that release operation.

To identify conflicting accesses, the DJIT$^+$ algorithm keeps two vector clocks, $\mathbb{R}_x$ and $\mathbb{W}_x$, for each variable x. For any thread t, $\mathbb{R}_x(t)$ and $\mathbb{W}_x(t)$ record the clock of the last read and write to x by thread t. A read from x by thread u is race-free provided it happens after the last write of each thread, that is, $\mathbb{W}_x \sqsubseteq \mathbb{C}_u$. A write to x by thread u is race-free provided that the write happens after all previous accesses to that variable, that is, $\mathbb{W}_x \sqsubseteq \mathbb{C}_u$ and $\mathbb{R}_x \sqsubseteq \mathbb{C}_u$.

As an example, consider the following fragment from an execution trace, where we include the relevant portion of the DJIT$^+$ instrumentation state: the vector clocks $\mathbb{C}_0$ and $\mathbb{C}_1$ for threads 0 and 1; and the vector clocks $\mathbb{L}_m$ and $\mathbb{W}_x$ for the last release of lock m and the last write to variable x, respectively. We show two components for each vector clock, but the target program may of course contain additional threads.[2]



[2] For clarity, we present a variant of the DJIT$^+$ algorithm where some clocks are one less than in the original formulation [29]. This revised algorithm has the same performance as the original but is slightly simpler and more directly comparable to FASTTRACK.

At the write $wr(0, x)$, DJIT$^+$ updates $\mathbb{W}_x$ with current clock of thread 0. At the release $rel(0, m)$, $\mathbb{L}_m$ is updated with $\mathbb{C}_0$. At the acquire $acq(1, m)$, $\mathbb{C}_1$ is joined with $\mathbb{L}_m$, thus capturing the dashed release-acquire happens-before edge shown above. At the second write, DJIT$^+$ compares the vector clocks

$$\mathbb{W}_x = \langle 4, 0, \ldots \rangle \sqsubseteq \langle 4, 8, \ldots \rangle = \mathbb{C}_1$$

Since this check passes, the two writes are not concurrent, and no race condition is reported.

3. The FASTTRACK Algorithm

A limitation of VC-based race detectors such as DJIT$^+$ is their performance. If a target program has n threads, then each vector clock requires $O(n)$ storage space and each vector clock operation (copying, comparing, joining, etc) requires $O(n)$ time.

Empirical data gathered from a variety of Java programs indicates that synchronization operations (lock acquires and releases, forks, joins, waits, notifies, etc) account for a very small fraction of the operations that must be monitored by a race detector. Reads and writes to object fields and arrays, on the other hand, account for over 96% of monitored operations. The key insight behind FASTTRACK is that the full generality of vector clocks is not necessary in over 99% of these read and write operations: a more lightweight representation of the happens-before information can be used instead. Only a small fraction of operations performed by the target program necessitate expensive vector clock operations.

We begin by providing an overview of how our analysis catches each type of race condition. Each race condition is either: a *read-write race condition* (where the trace contains a read that is concurrent with a later write to the same variable); a *write-read race condition* (a write concurrent with a later read); or a *write-write race condition* (involving two concurrent writes).

Detecting Write-Write Races. We first consider how to efficiently analyze write operations. At the second write operation in the trace discussed in the previous section, DJIT$^+$ compares the vector clocks $\mathbb{W}_x \sqsubseteq \mathbb{C}_1$ to determine whether there is a race. A careful inspection reveals, however, that it is not necessary to record the *entire* vector clock $\langle 4, 0, \ldots \rangle$ from the first write to x. Assuming no races have been detected on x so far,[3] then all writes to x are totally ordered by the happens-before relation, and so the *only critical information* that needs to be recorded is the clock (4) and identity (thread 0) of the thread performing the last write. This information (clock 4 of thread 0) is then sufficient to determine if a subsequent write to x is in a race with any preceding write.

We refer to a pair of a clock c and a thread t as an *epoch*, denoted $c@t$. Although rather simple, epochs provide the crucial lightweight representation for recording sufficiently-precise aspects of the happens-before relation efficiently. Unlike vector clocks, an epoch requires only constant space, independent of the number of threads in the program, and copying an epoch is a constant-time operation.

An epoch $c@t$ *happens before* a vector clock V ($c@t \preceq V$) if and only if the clock of the epoch is less than or equal to the corresponding clock in the vector.

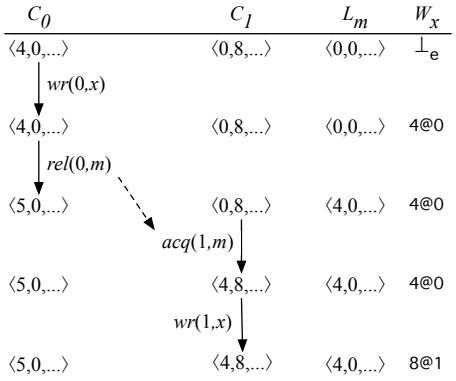
$$c@t \preceq V \quad \text{iff} \quad c \leq V(t)$$

Comparing an epoch to a vector clock ($\preceq$) requires only $O(1)$ time, unlike vector clock comparisons ($\sqsubseteq$), which require $O(n)$ time. We use $\bot_e$ to denote a minimal epoch $0@0$. (This minimal epoch is not unique; for example, another minimal epoch is $0@1$.)

Using this optimized representation, FASTTRACK analyzes the above trace using a compact instrumentation state that records only

[3] FASTTRACK guarantees to detect at least the first race on each variable.

a write epoch W_x for variable x, rather than the entire vector clock $\mathbb{W}_x$, reducing space overhead. (C and L record the same information as $\mathbb{C}$ and $\mathbb{L}$ in DJIT$^+$.)



At the first write to x, FASTTRACK performs an $O(1)$-time epoch write $W_x := 4@0$. FASTTRACK subsequently ensures that the second write is not concurrent with the preceding write via the $O(1)$-time comparison:

$$W_x = 4@0 \preceq \langle 4, 8, ... \rangle = C_1$$

To summarize, epochs reduce the space overhead for detecting write-write conflicts from $O(n)$ to $O(1)$ per allocated memory location, and replaces the $O(n)$-time vector clock comparison "$\sqsubseteq$" with the $O(1)$-time comparison operation "$\preceq$".

Detecting Write-Read Races. Detecting write-read races under the new representation is also straightforward. On each read from x with current vector clock C_t, we check that the read happens after the last write via the same $O(1)$-time comparison $W_x \preceq C_t$.

Detecting Read-Write Races. Detecting read-write race conditions is somewhat more difficult. Unlike write operations, which are totally ordered (assuming no race conditions detected so far), reads are not totally ordered even in race-free programs. Thus, a write to a variable x could potentially conflict with the last read of x performed by *any* other thread, not just the last read in the entire trace seen so far. Hence, we may need to record an entire vector clock R_x, in which $R_x(t)$ records the clock of the last read from x by thread t.

However, we can avoid keeping a complete vector clock in many cases. Our examination of data access patterns across a variety of multithreaded Java programs indicate that variable reads are often totally ordered *in practice*, particularly in the following common situations:

- Thread-local data, where only one thread accesses a variable, and hence these accesses are totally ordered by program-order.
- Lock-protected data, where a protecting lock is held on each access to a variable, and hence all access are totally ordered, either by program order (for accesses by the same thread) or by synchronization order (for accesses by different threads).

Reads are typically unordered only when data is *read-shared*, that is, when the data is first initialized by one thread and then shared between multiple threads in a read-only manner.

FASTTRACK uses an adaptive representation for tracking the read history of each variable that is tuned to optimize the common case of totally-ordered reads, while still retaining the full precision of vector clocks when necessary.

In particular, if the last read to a variable happens after all preceding reads, then FASTTRACK records only the epoch of this last read, which is sufficient to precisely detect whether a subsequent access to that variable conflicts with *any* preceding read in the entire program history. Thus, for thread-local and lock-protected data (which do exhibit totally-ordered reads), FASTTRACK requires only $O(1)$ space for each allocated memory location and only $O(1)$ time per memory access.

In the less common case where reads are not totally ordered, FASTTRACK stores the entire vector clock. However, it still handles read operations in $O(1)$ time, via an epoch-VC comparison ($\preceq$). In addition, since such data is typically read-shared, writes to such variables are rare, and so their analysis overhead is negligible.

Analysis Details. Based on the above intuition, we now describe the FASTTRACK algorithm in detail. Our analysis is an online algorithm that maintains an *analysis state* σ; when the target program performs an operation a, the analysis updates its state via the relation $\sigma \Rightarrow^a \sigma'$. The instrumentation state $\sigma = (C, L, R, W)$ is a tuple of four components, where:

- C_t identifies the current vector clock of thread t.
- L_m identifies the vector clock of the last release of lock m.
- R_x identifies either the epoch of the last read from x, if all other reads happened-before that read, or else records a vector clock that is the join of all reads of x.
- W_x identifies the epoch of the last write to x.

The initial analysis state is:

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

Figure 2 presents the key details of how FASTTRACK (left column) and DJIT$^+$ (right column) handle read and write operations of the target program. Expensive $O(n)$-time operations are highlighted in grey. That table also shows the instruction frequencies observed in our program traces, as well as how frequently each rule was applied. For example, 82.3% of all memory and synchronization operations performed by our benchmarks were reads, and rule [FT READ SAME EPOCH] was used to check 63.4% of those reads.

Read Operations. The first four rules provide various alternatives for analyzing a read operation $rd(t, x)$. Rule [FT READ SAME EPOCH] optimizes the case where x was already read in this epoch. This fast path requires only a single epoch comparison and handles over 60% of all reads. We use $E(t)$ to denote the current epoch $c@t$ of thread t, where $c = C_t(t)$ is t's current clock. DJIT$^+$ incorporates a comparable rule [DJIT$^+$ READ SAME EPOCH].

The remaining three read rules all check for write-read conflicts via the fast epoch-VC comparison $W_x \preceq C_t$, and then update R_x appropriately. Here, R is a function, R_x abbreviates the function application $R(x)$, and $R[x := V]$ denotes the function that is identical to R except that it maps x to V. Changes to the instrumentation state are expressed as functional updates for clarity in the transition rules, but they are implemented as constant-time in-place updates in our implementation.

If R_x is already a vector clock, then [FT READ SHARED] simply updates the appropriate component of that vector. Note that multiple reads of read-shared data from the same epoch are all covered by this rule. We could extend rule [FT READ SAME EPOCH] to handle same-epoch reads of read-shared data by matching the case that $R_x \in VC$ and $R_x(t) = C_t(t)$. The extended rule would cover 78% of all reads (the same as [DJIT$^+$ READ SAME EPOCH]) but does not improve performance of our prototype perceptibly.

If the current read happens after the previous read epoch (where that previous read may be either by the same thread or by a different thread, presumably with interleaved synchronization), [FT READ EXCLUSIVE] simply updates R_x with the accessing threads current epoch. For the more general situation where the current read may be concurrent with the previous read epoch,

Figure 2: FASTTRACK **Race Detection Algorithm and its Comparison to DJIT$^+$.**

FASTTRACK **State:**

$$\begin{array}{rcl} C & : & Tid \to VC \\ L & : & Lock \to VC \\ W & : & Var \to Epoch \\ R & : & Var \to (Epoch \cup VC) \end{array}$$

DJIT$^+$ State:

$$\begin{array}{rcl} \mathbb{C} & : & Tid \to VC \\ \mathbb{L} & : & Lock \to VC \\ \mathbb{W} & : & Var \to VC \\ \mathbb{R} & : & Var \to VC \end{array}$$

Reads:  **82.3% of all Operations**

[FT READ SAME EPOCH] 63.4% of reads

$$\frac{R_x = E(t)}{(C,L,R,W) \Rightarrow^{rd(t,x)} (C,L,R,W)}$$

[FT READ SHARED] 20.8% of reads

$$\frac{R_x \in VC \quad W_x \preceq C_t \quad R' = R[x := R_x[t := C_t(t)]]}{(C,L,R,W) \Rightarrow^{rd(t,x)} (C,L,R',W)}$$

[FT READ EXCLUSIVE] 15.7% of reads

$$\frac{R_x \in Epoch \quad R_x \preceq C_t \quad W_x \preceq C_t \quad R' = R[x := E(t)]}{(C,L,R,W) \Rightarrow^{rd(t,x)} (C,L,R',W)}$$

[FT READ SHARE] 0.1% of reads

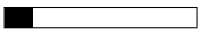
$$\frac{R_x = c@u \quad W_x \preceq C_t \quad V = \bot_V[t := C_t(t), u := c] \quad R' = R[x := V]}{(C,L,R,W) \Rightarrow^{rd(t,x)} (C,L,R',W)}$$

[DJIT$^+$ READ SAME EPOCH] 78.0% of reads

$$\frac{\mathbb{R}_x(t) = \mathbb{C}_t(t)}{(\mathbb{C},\mathbb{L},\mathbb{R},\mathbb{W}) \Rightarrow^{rd(t,x)} (\mathbb{C},\mathbb{L},\mathbb{R},\mathbb{W})}$$

[DJIT$^+$ READ] 22.0% of reads

$$\frac{\mathbb{W}_x \sqsubseteq \mathbb{C}_t \quad \mathbb{R}' = \mathbb{R}[x := \mathbb{R}_x[t := \mathbb{C}_t(t)]]}{(\mathbb{C},\mathbb{L},\mathbb{R},\mathbb{W}) \Rightarrow^{rd(t,x)} (\mathbb{C},\mathbb{L},\mathbb{R}',\mathbb{W})}$$

Writes:  **14.5% of all Operations**

[FT WRITE SAME EPOCH] 71.0% of writes

$$\frac{W_x = E(t)}{(C,L,R,W) \Rightarrow^{wr(t,x)} (C,L,R,W)}$$

[FT WRITE EXCLUSIVE] 28.9% of writes

$$\frac{R_x \in Epoch \quad R_x \preceq C_t \quad W_x \preceq C_t \quad W' = W[x := E(t)]}{(C,L,R,W) \Rightarrow^{wr(t,x)} (C,L,R,W')}$$

[FT WRITE SHARED] 0.1% of writes

$$\frac{R_x \in VC \quad R_x \sqsubseteq C_t \quad W_x \preceq C_t \quad W' = W[x := E(t)] \quad R' = R[x := \bot_e]}{(C,L,R,W) \Rightarrow^{wr(t,x)} (C,L,R',W')}$$

[DJIT$^+$ WRITE SAME EPOCH] 71.0% of writes

$$\frac{\mathbb{W}_x(t) = \mathbb{C}_t(t)}{(\mathbb{C},\mathbb{L},\mathbb{R},\mathbb{W}) \Rightarrow^{wr(t,x)} (\mathbb{C},\mathbb{L},\mathbb{R},\mathbb{W})}$$

[DJIT$^+$ WRITE] 29.0% of writes

$$\frac{\mathbb{W}_x \sqsubseteq \mathbb{C}_t \quad \mathbb{R}_x \sqsubseteq \mathbb{C}_t \quad \mathbb{W}' = \mathbb{W}[x := \mathbb{W}_x[t := \mathbb{C}_t(t)]]}{(\mathbb{C},\mathbb{L},\mathbb{R},\mathbb{W}) \Rightarrow^{wr(t,x)} (\mathbb{C},\mathbb{L},\mathbb{R},\mathbb{W}')}$$

[FT READ SHARE] allocates a vector clock to record the epochs of *both* reads, since either read could subsequently participate in a read-write race.

Of these three rules, the last rule is the most expensive but is rarely needed (0.1% of reads) and the first three rules provide commonly-executed, constant-time fast paths. In contrast, the corresponding DJIT$^+$ rule [DJIT$^+$ READ] *always* executes an $O(n)$-time vector clock comparison for these cases.

Write Operations. The next three FASTTRACK rules handle a write operation $wr(t, x)$. Rule [FT WRITE SAME EPOCH] optimizes the case where x was already written in this epoch, which applies to 71.0% of write operations, and DJIT$^+$ incorporates a comparable rule. [FT WRITE EXCLUSIVE] provides a fast path for the 28.9% of writes for which R_x is an epoch, and this rule checks that the write happens after all previous accesses. In the case where R_x is a vector clock, [FT WRITE SHARED] requires a full (slow) VC comparison, but this rule applies only to a tiny fraction (0.1%) of writes. In contrast, the corresponding DJIT$^+$ rule [DJIT$^+$ WRITE] requires slow VC comparisons on 29.0% of writes.

Other Operations. Figure 3 shows how FASTTRACK handles all other operations (acquire, release, fork, and join). These operations are rare, so the traditional analysis for these operations in terms of expensive VC operations is perfectly adequate. Thus, these FASTTRACK rules are similar to those of DJIT$^+$ and other VC-based analyses.

Example. The execution trace in Figure 4 illustrates how FASTTRACK dynamically adapts the representation for the read history R_x of a variable x. Initially, R_x is $\bot_e$, indicating that x has not yet been read. After the first read operation $rd(1, x)$, R_x becomes the epoch $1@1$ recording both the clock and the thread identifier of that read. The second read $rd(0, x)$ at clock 8 is concurrent with the first read, and so FASTTRACK switches to the vector clock representation $\langle 8, 1, \ldots \rangle$ for R_x, recording the clocks of the last reads from x by both threads 0 and 1. After the two threads join, the write operation $wr(0, x)$ happens after all reads. Hence, any later operation cannot be in a race with either read without also being in a race on that write operation, and so the rule [FT WRITE SHARED] discards the read history of x by resetting R_x to $\bot_e$, which also switches x back into epoch mode and so optimizes later accesses to x. The last read in the trace then sets R_x to a non-minimal epoch.

Correctness. FASTTRACK is precise and reports data races if and only if the observed trace contains concurrent conflicting accesses, as characterized by the following theorem.

THEOREM 1 (Correctness). *Suppose α is a feasible trace. Then α is race-free if and only if $\exists \sigma$ such that $\sigma_0 \Rightarrow^{\alpha} \sigma$.*

PROOF The two directions of this theorem follow from Theorems 2 and 3, respectively. See Appendix. □

4. Implementation

ROADRUNNER. We have developed a prototype implementation of FASTTRACK as a component of ROADRUNNER, a framework we have designed for developing dynamic analyses for multithreaded software. ROADRUNNER is written entirely in Java and runs on any JVM. ROADRUNNER inserts instrumentation code into the target bytecode program at load time. This instrumentation code generates a stream of events for lock acquires and releases, field and array accesses, method entries and exits, etc. Back-end tools, such as FASTTRACK, process this event stream as it is generated. Re-entrant lock acquires and releases (which are redundant) are filtered out by ROADRUNNER to simplify these analyses.

Figure 3: Synchronization and Threading Operations

Other: 3.3% of all Operations

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

[FT RELEASE]
$$\frac{L' = L[m := C_t] \quad C' = C[t := inc_t(C_t)]}{(C, L, R, W) \Rightarrow^{rel(t,m)} (C', L', R, W)}$$

[FT FORK]
$$\frac{C' = C[u := C_u \sqcup C_t, t := inc_t(C_t)]}{(C, L, R, W) \Rightarrow^{fork(t,u)} (C', L, R, W)}$$

[FT JOIN]
$$\frac{C' = C[t := C_t \sqcup C_u, u := inc_u(C_u)]}{(C, L, R, W) \Rightarrow^{join(t,u)} (C', L, R, W)}$$

ROADRUNNER enables back-end tools to attach instrumentation state to each thread, lock object, and data memory location used by the target program. Tool-specific event handlers update the instrumentation state for each operation in the observed trace and report errors when appropriate.

The ROADRUNNER framework provides several benefits. By working exclusively at the bytecode level, ROADRUNNER tools can check any Java program regardless of whether source code is available. Moreover, tools only need to reason about the relatively simple bytecode language. In addition, ROADRUNNER's component architecture facilitates reliable comparisons between different back-end checking tools.

FastTrack Instrumentation State and Code. FASTTRACK represents an epoch $c@t$ as a 32-bit integer, where the top eight bits store the thread identifier t and the bottom twenty-four bits store the clock c. Two epochs for the same thread can be directly compared as integers, since the thread identifier bits in each integer are identical. FASTTRACK represents a vector clock as an array of epochs, even though the thread identifier bits in these epochs are redundant, since this representation optimizes the epoch-VC comparison operation ($\preceq$). We use the function `TID(`e`)` to extract the thread identifier of an epoch.

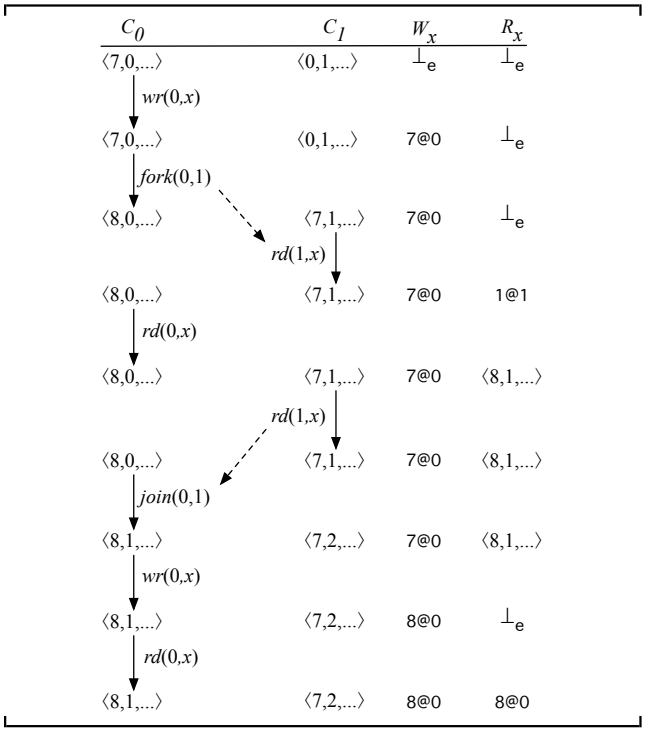
While 32-bit epochs has been sufficient for all programs tested, switching to 64-bit epochs would enable the FASTTRACK to handle large thread identifiers or clock values. In addition, existing techniques to reduce the size of vector clocks [10] could also be employed to save space.

FASTTRACK associates with each thread a `ThreadState` object (see Figure 5) containing a unique thread identifier `tid` and a vector clock `C`. The current epoch for thread t can be expressed as t`.C[`t`.tid]`, and we cache that value in the `epoch` field.

Each memory location (object field or array element) has an associated `VarState` object containing epochs `W` and `R` and a vector clock `Rvc`. These represent the W and R components of the analysis state. Setting `R` to the special epoch `READ_SHARED` indicates that the location is in read-shared mode, and hence `Rvc` is in use. FASTTRACK also maintains a `LockState` containing a vector clock for each object used as a lock.

Figure 5 shows FASTTRACK event handling pseudocode for read and write operations. The code includes the name of the corresponding FASTTRACK analysis rule for each code branch, as well

Figure 4: Example Trace.



as the percentage of time that the branch is taken. Note that the two slow operations in these event handlers (vector clock allocation and comparison) are executed extremely rarely, on 0.1% of reads and 0.1% of writes. Despite its performance, the FASTTRACK algorithm is surprisingly straightforward to implement. Event handlers for other operations are also simple but less interesting. Our actual implementation introduces some additional features, such as more precise error reporting, and it replaces the $\not\preceq$ check "`x.W > t.C[TID(x.W)]`" with the equivalent but slightly faster condition "`TID(x.W) != t.tid && x.W > t.C[TID(x.W)]`", eliminates obvious common subexpressions, etc.

Granularity. ROADRUNNER supports two levels of granularity for analyzing memory locations. The default *fine-grain* analysis treats each field of an object as a distinct entity that has its own `VarState`. The *coarse-grain* analysis treats all fields of an object as a single entity with a single `VarState`. The coarse-grain analysis significantly reduces the memory footprint for `VarState`s, but may produce false alarms if, for examples, two fields of an object are protected by different locks. However, as others have observed [38, 42], object-oriented programs most often use the same synchronization discipline for all fields of an object. ROADRUNNER also supports fine and coarse grain analyses for arrays.

Extensions. The FASTTRACK implementation extends our analysis in several straightforward ways. Most importantly, it supports additional synchronization primitives, including `wait` and `notify`, `volatile` variables, and barriers.

FASTTRACK models a `wait` operation on lock m in terms of the underlying release and subsequent acquisition of m. Thus, no additional analysis rules are necessary. A `notify` operation can be ignored by FASTTRACK. It affects scheduling of threads but does not induce any happens-before edges between them.

The Java Memory Model [22] guarantees that a write to a volatile variable $vx \in VolatileVar$ happens before every subsequent read of vx. To handle volatiles, FASTTRACK extends the L component of the analysis state to map volatile variables to the vector clock of the last write:

$$L \ : \ (Lock \cup VolatileVar) \rightarrow VC$$

Volatile reads and writes then modify the instrumentation state in much the same way as lock acquire and release.

[FT READ VOLATILE]
$$\frac{C' \ = \ C[t := C_t \sqcup L_{vx}]}{(C, L, R, W) \Rightarrow^{vol\_rd(t,vx)} (C', L, R, W)}$$

[FT WRITE VOLATILE]
$$\frac{L' \ = \ L[vx := (C_t \sqcup L_{vx})] \quad C' \ = \ C[t := inc_t(C_t)]}{(C, L, R, W) \Rightarrow^{vol\_rd(t,vx)} (C', L', R, W)}$$

We also add a new type of event to indicate when threads are released from a barrier. While strictly not necessary since FASTTRACK can precisely handle the synchronization primitives upon which barriers are built, the barrier implementations used in many

Figure 5: FastTrack Instrumentation State and Code

```
class ThreadState {
  int tid;
  int C[];
  int epoch;  // invariant: epoch == C[tid]
}

class VarState {
  int W, R;
  int Rvc[]; // used iff R == READ_SHARED
}

class LockState {
  int L[];
}

void read(VarState x, ThreadState t)
  if (x.R == t.epoch) return;                 // Same Epoch 63.4%

  // write-read race?
  if (x.W > t.C[TID(x.W)]) error;

  // update read state
  if (x.R == READ_SHARED) {                   // Shared     20.8%
    x.Rvc[t.tid] = t.epoch;
  } else {
    if (x.R <= t.C[TID(x.R)]) {               // Exclusive  15.7%
      x.R = t.epoch;
    } else {                                  // Share       0.1%
      if (x.Rvc == null)
        x.Rvc = newClockVector();                   // (SLOW PATH)
      x.Rvc[TID(x.R)] = x.R;
      x.Rvc[t.tid] = t.epoch;
      x.R = READ_SHARED;
    }
  }
}

void write(VarState x, ThreadState t)
  if (x.W == t.epoch) return;                 // Same Epoch 71.0%

  // write-write race?
  if (x.W > t.C[TID(x.W)]) error;

  // read-write race?
  if (x.R != READ_SHARED) {                   // Shared     28.9%
    if (x.R > t.C[TID(x.R)]) error;
  } else {                                    // Exclusive   0.1%
    if (x.Rvc[u] > t.C[u] for any u) error;       // (SLOW PATH)
  }
  x.W = t.epoch; // update write state
}
```

benchmarks and libraries contain benign race conditions that would cause spurious warnings if not handled specially. The operation $barrier\_rel(T)$ indicates that the set of threads T are simultaneously released from a barrier. (We do not need to track entries to barriers.)

[FT BARRIER RELEASE]

$$\frac{C' = \lambda t..\ \texttt{if}\ t \in T\ \texttt{then}\ inc_t(\bigsqcup_{u \in T} C_u)\ \texttt{else}\ C(t)}{(C, L, R, W) \Rightarrow^{barrier\_rel(T)} (C', L, R, W)}$$

Thus, the first post-barrier step by any thread $t \in T$ happens after all pre-barrier steps by threads in T and is unordered with respect to the next steps taken by other threads in T. Configuration options enable FASTTRACK to use this rule for any barrier implementation identified by the user.

Although FASTTRACK does not currently support the full set of concurrency primitives available in the Java concurrency library [37], we believe their effects on a program's happens-before graph can all be modeled in our representation.

5. Evaluation

We validate the effectiveness of FASTTRACK with three sets of experiments. We first compare FASTTRACK's performance and precision to other dynamic race detectors when checking a variety of benchmarks. Second, we demonstrate how to use FASTTRACK to improve the performance of checkers for more complex concurrency properties. Finally, we describe our experience of using FASTTRACK to check the Eclipse development environment [13].

5.1 Precision and Performance

This section compares the precision and performance of seven dynamic analyses: EMPTY (which performs no analysis and is used to measure the overhead of ROADRUNNER); FASTTRACK; ERASER [33], extended to handle barrier synchronization [29]; DJIT$^+$ [30]; MULTIRACE [30]; GOLDILOCKS [14]; and BASICVC. BASICVC is a simple VC-based race detector that maintains a read and a write VC for each memory location and performs at least one VC comparison on every memory access. To ensure reliable comparisons, all tools were implemented on top of ROADRUNNER as similarly as possible. For example, BASICVC, DJIT$^+$, MULTIRACE, and FASTTRACK use the same vector clock implementation, and all checkers were profiled and tuned to eliminate unnecessary bottlenecks.

We note that several additional techniques could be used to improve the performance of all these checkers. For example, we could (1) include a separate static analysis to reduce the need for run-time checks; (2) include a separate dynamic escape analysis; (3) add unsound optimizations; (4) tune ROADRUNNER to better support one particular kind of analysis; (5) implement these checkers directly on a JVM rather than on top of ROADRUNNER; (6) implement the checkers inside the JVM itself (sacrificing portability, maintainability, etc); or (7) at the extreme, implement the checkers directly in hardware. These techniques would improve performance in ways that are orthogonal to the central contributions of this paper, and at the cost of additional complexity. In order to present our results n the most consistent manner, we do not report on these complementary optimizations.

Benchmark Configuration. We performed experiments on 16 benchmarks: `elevator`, a discrete event simulator for elevators [39]; `hedc`, a tool to access astrophysics data from Web sources [39]; `tsp`, a Traveling Salesman Problem solver [39]; `mtrt`, a multithreaded ray-tracing program from the SPEC JVM98 benchmark suite [35]; `jbb`, the SPEC JBB2000 business object simulator [35]; `crypt`, `lufact`, `sparse`, `series`, `sor`, `moldyn`, `montecarlo`, and `raytracer` from the Java Grande benchmark suite [20]; the `colt` scientific computing library [6]; the `raja` ray tracer [18]; and `philo`, a dining philosophers simulation [14]. The Java Grande benchmarks were configured to use four worker threads and the largest data set provided (except `crypt`, for which we used the smallest data set because BASICVC, DJIT$^+$, and MULTIRACE ran out of memory on the larger sizes).

All experiments were performed on an Apple Mac Pro with dual 3GHz quad-core Pentium Xeon processors and 12GB of memory, running OS X 10.5.6 and Sun's Java HotSpot 64-bit Server VM version 1.6.0. All classes loaded by the benchmark programs were instrumented, except those from the standard Java libraries. The timing measurements include the time to load, instrument, and execute the target program, but it excludes JVM startup time to reduce noise. The tools report at most one race for each field of each class, and at most one race for each array access in the program source code.

Summary of Results. Table 1 lists the size, number of threads, and uninstrumented running times for each program examined. All timing measurements are the average of 10 test runs. Variability across consecutive runs was typically less than 10%. The four programs marked with '*' are not compute-bound, and we exclude these programs when computing average slowdowns.

The "Instrumented Time" columns show the running times of each program under each of the tools, reported as the ratio to the uninstrumented running time. Thus, target programs ran 4.1 times slower, on average, under the EMPTY tool. Most of this overhead is due to communicating all target program operations to the back-end checker. For GOLDILOCKS, we include both measurements for our own implementation on top of ROADRUNNER and performance estimates for the original implementation [14], as discussed below.

The variations in slowdowns for different programs that we observed in our experiments are not uncommon for dynamic race condition checkers. Different programs exhibit different memory access and synchronization patterns, some of which will impact analysis performance more than others. In addition, instrumentation can impact cache performance, class loading time, and other low-level JVM operations. These differences can sometimes even make an instrumented program run slightly faster than the uninstrumented (as in colt).

The last six columns show the number of warnings produced by each checker using the fine grain analysis. All eight warnings from FASTTRACK reflect real race conditions. As reported previously [16, 28, 38, 39], some of these are benign (as in `tsp`, `mtrt`, and `jbb`) but others can impact program behavior (as on the `checksum` field in `raytracer` and races on several fields related to a thread pool in `hedc`).

ERASER ***Comparison.*** The performance results show that our re-implementation of ERASER incurs an overhead of 8.7x, which is competitive with similar Eraser implementations built on top of unmodified JVMs, such as [28]. Surprisingly, FASTTRACK is slightly faster than ERASER on some programs, even though it performs a precise analysis that traditionally has been considered more expensive.

More significantly, ERASER reported many spurious warnings that do not correspond to actual races.[4] Augmenting our ERASER implementation to reason about additional synchronization constructs [30, 42] would eliminate some of these spurious warnings, but not all. On `hedc`, ERASER reported a spurious warning and also missed two of the real race conditions reported by FASTTRACK,

[4] The total number of warnings is about three times higher if ERASER does not reason about barriers.

Program	Size (loc)	Thread Count	Base Time (sec)	Instrumented Time (slowdown)								Warnings					
				EMPTY	ERASER	MULTIRACE	GOLDILOCKS RR	GOLDILOCKS KAFFE	BASICVC	DJIT$^+$	FASTTRACK	ERASER	MULTIRACE	GOLDILOCKS	BASICVC	DJIT$^+$	FASTTRACK
colt	111,421	11	16.1	0.9	0.9	0.9	1.8	2	0.9	0.9	0.9	3	0	0	0	0	0
crypt	1,241	7	0.2	7.6	14.7	54.8	77.4	–	84.4	54.0	14.3	0	0	0	0	0	0
lufact	1,627	4	4.5	2.6	8.1	42.5	–	8.5	95.1	36.3	13.5	4	0	–	0	0	0
moldyn	1,402	4	8.5	5.6	9.1	45.0	17.5	28.5	111.7	39.6	10.6	0	0	0	0	0	0
montecarlo	3,669	4	5.0	4.2	8.5	32.8	6.3	2.2	49.4	30.5	6.4	0	0	0	0	0	0
mtrt	11,317	5	0.5	5.7	6.5	7.1	6.7	–	8.3	7.1	6.0	1	1	1	1	1	1
raja	12,028	2	0.7	2.8	3.0	3.2	2.7	–	3.5	3.4	2.8	0	0	0	0	0	0
raytracer	1,970	4	6.8	4.6	6.7	17.9	32.8	146.8	250.2	18.1	13.1	1	1	1	1	1	1
sparse	868	4	8.5	5.4	11.3	29.8	64.1	–	57.5	27.8	14.8	0	0	0	0	0	0
series	967	4	175.1	1.0	1.0	1.0	1.0	1.1	1.0	1.0	1.0	1	0	0	0	0	0
sor	1,005	4	0.2	4.4	9.1	16.9	63.2	1.4	24.6	15.8	9.3	3	0	0	0	0	0
tsp	706	5	0.4	4.4	24.9	8.5	74.2	2.9	390.7	8.2	8.9	9	1	1	1	1	1
elevator*	1,447	5	5.0	1.1	1.1	1.1	1.1	–	1.1	1.1	1.1	0	0	0	0	0	0
philo*	86	6	7.4	1.1	1.0	1.1	7.2	1	1.1	1.1	1.1	0	0	0	0	0	0
hedc*	24,937	6	5.9	1.1	0.9	1.1	1.1	3.3	1.1	1.1	1.1	2	1	0	3	3	3
jbb*	30,491	5	72.9	1.3	1.5	1.6	2.1	–	1.6	1.6	1.4	3	1	–	2	2	2
Average				4.1	8.6	21.7	31.6	24.2	89.8	20.2	8.5	27	5	3	8	8	8

Table 1: Benchmark Results. Programs marked with '*' are not compute-bound and are excluded when computing average slowdowns.

Program	Vector Clocks Allocated		Vector Clock Operations	
	DJIT$^+$	FAST TRACK	DJIT$^+$	FAST TRACK
colt	849,765	76,209	5,792,894	1,266,599
crypt	17,332,725	119	28,198,821	18
lufact	8,024,779	2,715,630	3,849,393,222	3,721,749
moldyn	849,397	26,787	69,519,902	1,320,613
montecarlo	457,647,007	25	519,064,435	25
mtrt	2,763,373	40	2,735,380	402
raja	1,498,557	3	760,008	1
raytracer	160,035,820	14	212,451,330	36
sparse	31,957,471	456,779	56,553,011	15
series	3,997,307	13	3,999,080	16
sor	2,002,115	5,975	26,331,880	54,907
tsp	311,273	397	829,091	1,210
elevator	1,678	207	14,209	5,662
philo	56	12	472	120
hedc	886	82	1,982	365
jbb	109,544,709	1,859,828	327,947,241	64,912,863
Total	796,816,918	5,142,120	5,103,592,958	71,284,601

Table 2: Vector Clock Allocation and Usage.

due to an (intentional) unsoundness in how the Eraser algorithm reasons about thread-local and read-shared data [33].

BASICVC ***and*** **DJIT$^+$** ***Comparison.*** DJIT$^+$ and BASICVC reported exactly the same race conditions as FASTTRACK. That is, the three checkers all yield identical precision. In terms of performance, however, the results show that FASTTRACK significantly outperforms the other checkers. In particular, it is roughly 10x faster than BASICVC and 2.3x faster than DJIT$^+$. These performance improvements are due primarily to the reduction in the allocation and use of vector clocks, as shown in Table 2. Over all the benchmarks, DJIT$^+$ allocated more over 790 million vector clocks, whereas FASTTRACK allocated only 5.1 million. DJIT$^+$ performed over 5.1 billion $O(n)$-time vector clock operations, while FASTTRACK performed only 17 million. The memory overhead for storing the extra vector clocks leads to significant cache performance degradation in some programs, particularly those that perform random accesses to large arrays.

MULTIRACE ***Comparison.*** MULTIRACE maintains DJIT$^+$'s instrumentation state, as well as a lock set for each memory location [29]. The checker updates the lock set for a location on the first access in an epoch, and full vector clock comparisons are performed after this lock set becomes empty. This synthesis substantially reduces the number of vector clock operations, but introduces the overhead of storing and updating lock sets. In addition, the use of ERASER's unsound state machine for thread-local and read-shared data leads to imprecision. In combination with a coarse-grain analysis, this approach produced substantial performance improvement [29].

Our re-implementation of the MULTIRACE algorithm in ROADRUNNER used fine-grain analysis and exhibited performance comparable to DJIT$^+$. Interestingly, over all benchmarks MULTIRACE performed less than half the number of VC operations as FASTTRACK. However, this did not lead to speed-ups over DJIT$^+$, because the memory footprint was even larger than DJIT$^+$, leading to substantial cache performance degradation. Additionally, on average roughly 10% of all operations required an ERASER operation that also imposed some additional overhead.

GOLDILOCKS ***Comparison.*** GOLDILOCKS [14] is a precise race detector that does not use vector clocks to capture the happens-before relation. Instead, it maintains, for each memory location, a set of "synchronization devices" and threads. A thread in that set can safely access the memory location, and a thread can add itself to the set (and possibly remove others) by performing any of the operations described by the synchronization devices in the set.

GOLDILOCKS is a complicated algorithm that required 1,900 lines of code to implement in ROADRUNNER, as compared to fewer than 1,000 lines for each other tool. GOLDILOCKS also ideally requires tight integration with the underlying virtual machine and, in particular, with the garbage collector, which is not possible under ROADRUNNER. Both of these factors cause GOLDILOCKS to incur a high slowdown. As shown in Table 1, GOLDILOCKS implemented in ROADRUNNER incurred a slowdown of 31.6x across our benchmarks (but ran out of memory on lufact), even when utilizing an unsound extension to handle thread-local data efficiently. (This extension caused it to miss the three races in hedc found by other tools.) We believe some performance improvements are possible, for both GOLDILOCKS and the other tools, by integration into the virtual machine.

As another data point, the original GOLDILOCKS study reported its slowdown for the compute-intensive benchmarks in common

Program	Memory (MB)	Memory Overhead				Slowdown			
		Fine		Coarse		Fine		Coarse	
		DJIT+	FASTTRACK	DJIT+	FASTTRACK	DJIT+	FASTTRACK	DJIT+	FASTTRACK
colt	36	4.3	2.4	2.0	1.8	0.9	0.9	0.9	0.8
crypta	41	44.3	10.5	1.2	1.2	54.0	14.3	6.6	6.6
lufactc	80	9.8	4.1	1.1	1.1	36.3	13.5	5.4	6.6
moldynb	37	3.3	1.7	1.3	1.2	39.6	10.6	11.9	8.3
montecarlob	595	6.1	2.1	1.1	1.1	30.5	6.4	3.4	2.8
mtrt	51	3.9	2.2	2.6	1.9	7.1	6.0	8.3	7.0
raja	35	1.3	1.3	1.2	1.3	3.4	2.8	3.1	2.7
raytracerb	36	6.2	1.9	1.4	1.2	18.1	13.1	14.5	10.6
sparsec	131	23.3	6.1	1.0	1.0	27.8	14.8	3.9	4.1
seriesc	51	8.5	3.1	1.1	1.1	1.0	1.0	1.0	1.0
sorc	40	5.3	2.1	1.1	1.1	15.8	9.3	5.8	6.3
tsp	33	1.7	1.3	1.2	1.2	8.2	8.9	7.6	7.3
elevator*	32	1.2	1.2	1.2	1.2	1.1	1.1	1.1	1.1
philo*	32	1.2	1.2	1.2	1.2	1.1	1.1	1.1	1.1
hedc*	33	1.4	1.4	1.3	1.3	1.1	1.1	0.9	0.9
jbb*	236	4.1	2.4	2.3	1.9	1.6	1.4	1.3	1.3
Average		7.9	2.8	1.4	1.3	20.2	8.5	6.0	5.3

Table 3: Comparison of Fine and Coarse Granularities.

with this paper to be roughly 4.5x [14]. However, those results are for a GOLDILOCKS implementation written in *compiled* C code inside the Kaffe Virtual Machine, and target programs were *interpreted* by this modified JVM.[5] If GOLDILOCKS were implemented inside a JIT, then target programs would run significantly faster, and so the GOLDILOCKS slowdown would appear larger but would be more directly comparable to ROADRUNNER tools (in which both the target program and checker are optimized by the JIT). To compensate for this effect, we estimated the corresponding GOLDILOCKS-JIT slowdown from the experimental results of [14] as:

$$\frac{(\text{Goldilocks-Time} - \text{Interpreted-Time}) + \text{JIT-Time}}{\text{JIT-Time}}$$

That is, we compute the cost of the Goldilocks analysis based on the running times of the unmodified Kaffe interpreter and the Goldilocks-aware Kaffe interpreter, and then estimate the slowdown that this overhead would have on the running time of the target program in the Kaffe JIT. We include these estimates in Table 1 in the "Kaffe" column. They suggest the GOLDILOCKS slowdown on a JIT could be as high as 25x, which is close to our implementation.

In summary, GOLDILOCKS is an interesting algorithm but its complexity and JVM-integration issues make it difficult to implement efficiently, and so GOLDILOCKS may not provide significant performance benefits over FASTTRACK.

Analysis Granularity. Next, we investigate the effect of analysis granularity on space and time overheads. Table 3 shows the memory requirements for each uninstrumented program, and the memory overhead factor and slowdown factor for DJIT+ and FASTTRACK, using both fine and course grain analyses. Memory overhead is reported as the ratio of the maximum heap space used during analysis to the maximum heap space used under uninstrumented execution. The overall increase in heap usage can be smaller than the total size of allocated vector clocks because the garbage collector collects vector clocks belonging to reclaimed objects.

The memory overhead for the fine-grain analysis is substantial (columns 3 and 4), since every field and array element requires a its own "`VarState`" object. However, FASTTRACK's memory requirements are substantially better than DJIT+'s because many fewer vector clocks are allocated. The coarse-grain analysis (column 5 and 6) reduces the memory overhead by roughly half for both checkers, and results in a roughly 50% speedup.

The coarse-grain analysis does cause FASTTRACK and the other analyses to report spurious warnings on most of the benchmarks. We could adaptively refine the granularity of the analysis for those objects for which the coarse analysis generates warnings, either by iteratively running the checker under different levels of granularity [30] or by performing on-line adaptation [42] with some loss of precision. Our estimates suggest that performing on-line adaptation in ROADRUNNER would yield performance close to the coarse-grain analysis, but with some improvement in precision.

5.2 Analysis Composition

Precise race condition information can also significantly improve the performance of other dynamic analyses. For example, atomicity checkers, such as ATOMIZER [16] and VELODROME [17], and determinism checkers, such as SINGLETRACK [32], can ignore race-free memory accesses.

To compose independent analyses, the ROADRUNNER command line option: "`-tool FastTrack:Velodrome`" configures ROADRUNNER to feed the event stream from the target program to FASTTRACK, which filters out race-free memory accesses from the event stream and passes all other events on to VELODROME.[6]

The following table illustrates the improvement of ATOMIZER, VELODROME and SINGLETRACK under five different filters: NONE, which shows the slowdown of these tools over the original, uninstrumented benchmark programs; TL, which filters out only accesses to thread-local data; ERASER; DJIT+; and FASTTRACK. FASTTRACK significantly improves the performance of these tools, because their rather complex analyses can avoid analyzing potentially millions of uninteresting, race-free data accesses. The slowdowns reported are the average slowdown for our compute-bound benchmarks.

Checker	Slowdown for Prefilters				
	NONE	TL	ERASER	DJIT+	FASTTRACK
ATOMIZER	57.2	16.8	—[7]	17.5	12.6
VELODROME	57.9	27.1	14.9	19.6	11.3
SINGLETRACK	104.1	55.4	32.7	19.7	11.7

[5] Some of the JavaGrande benchmark programs were run on smaller data sets than we used, which could also impact relative performance.

[6] Note that FASTTRACK (and other tools) may filter out a memory access that is later determined to be involved in a race condition; thus this optimization may involve some small reduction in coverage.

5.3 Checking Eclipse for Race Conditions

To validate FASTTRACK in a more realistic setting, we applied it to the Eclipse development environment, version 3.4.0 [13]. We modified two lines of source code to report the time to perform each user-initiated operation but made no other modifications. We used Sun's Java 1.5.0 HotSpot Client VM because our test platform must run Eclipse as a 32-bit application with a maximum heap size of 2GB.

We performed experiments with the following five Eclipse operations:

Startup: Launch Eclipse and load a workspace containing four projects with 65,000 lines of code.

Import: Import and perform an initial build on a project containing 23,000 lines of code.

Clean Small: Rebuild a workspace with four projects containing a total of 65,000 lines of code.

Clean Large: Rebuild a workspace with one project containing 290,000 lines of code.

Debug: Launch the debugger and run a small program that immediately throws an exception.

Operation	Base Time (sec)	Instrumented Time (Slowdown)			
		EMPTY	ERASER	DJIT$^+$	FAST TRACK
Startup	6.0	13.0	16.0	17.3	16.0
Import	2.5	7.6	14.9	17.1	13.1
Clean Small	2.7	14.1	16.7	24.4	15.2
Clean Large	6.5	17.1	17.9	38.5	15.4
Debug	1.1	1.6	1.7	1.7	1.6

For these tests, FASTTRACK loaded and instrumented roughly 6,000 classes corresponding to over 140,000 lines of source code. Several classes using reflection were not instrumented to avoid current limitations of ROADRUNNER, but these classes did not involve interesting concurrency. Native methods were also not instrumented. Eclipse used up to 24 concurrent threads during these tests.

Eclipse startup time was heavily impacted by the bytecode instrumentation process, which accounted for about 75% of the observed slowdowns. While ROADRUNNER does not currently support off-line bytecode instrumentation, that feature would remove a substantial part of startup time. FASTTRACK performed quite well on the three most compute-intensive tests (Import, Clean Small, and Clean Large), exhibiting performance better than DJIT$^+$ and comparable to ERASER. Monitoring the Debug test incurred less overhead for all tools because that test spent much of its time starting up the target VM and switching views on the screen.

ERASER reported potential races on 960 distinct field and array accesses for these five tests, largely because Eclipse uses many synchronization idioms, such as wait/notify, semaphores, readers-writer locks, etc. that ERASER cannot handle. Additional extensions [42] could handle some of these cases.

FASTTRACK reported 30 distinct warnings. While we have not been able to fully verify the correctness of all code involved in these races, none caused major failures or data corruption during our tests. They include:

- Races on an array of nodes in a tree data structure implementation. We believe these races can cause null pointer exceptions under Java's non-sequentially consistent memory model.
- Races on fields related to progress meters that may cause incorrect information to be displayed briefly.
- An instance of double-checked locking in the code to read files for compilation units from disk that, while benign, adds significant complexity to the code.
- Benign races on array entries used to communicate values from helper threads back to their parents, and in the initialization code in the debugger for monitoring input and output streams.

DJIT$^+$ reported 28 warnings. These overlapped heavily with those reported by FASTTRACK, but scheduling differences led to several being missed and several new (benign) races being identified. The items listed above were reported by both tools.

Although our exploration of Eclipse is far from complete, these preliminary observations are quite promising. FASTTRACK is able to scale to precisely check large applications with lower run-time and memory overheads than existing tools.

[7] Since ATOMIZER already uses ERASER to identify potential races internally, it would not be meaningful to add it as a prefilter.

6. Related Work

Much prior work has focused on dynamic analyses to detect race conditions. In addition to the dynamic race detectors discussed earlier, a variety of alternative approaches have been explored. Eraser's LockSet algorithm [33] has been refined to eliminate false positives, reduce overhead, and handle additional synchronization idioms, as in [38, 27].

Some race detectors have combined Eraser's LockSet algorithm with happens-before reasoning (*e.g.*, for analyzing barriers and fork-join synchronization), with good results. RaceTrack [42], for example, uses happens-before information to approximate the set of threads concurrently accessing memory locations. An empty lock set is only considered to reflect a potential race if the happens-before analysis indicates that the corresponding location is accessed concurrently by multiple threads. MultiRace [30], as described above, also falls into this category. While these analyses reduce the number of false alarms, they cannot eliminate them completely. Other approaches have combined dynamic analysis with a global static analysis to improve precision and performance [8, 14, 39].

TRaDE [11] is a precise race detector based on a dynamic escape analysis and accordion clocks [10], a technique that reduces the space overhead of vector clocks for applications with many short-lived threads. Unlike FASTTRACK, TRaDE is implemented inside the HotSpot virtual machine interpreter, and so the target program is interpreted while the instrumentation code is compiled, making it difficult to compare results directly. However, for the two benchmarks (`colt` and `raja`) common to both papers, FASTTRACK is several times faster. We expect the TRaDE innovations (dynamic escape analysis and accordion clocks) to further improve FASTTRACK's performance.

In addition to *on-the-fly* analyses that monitor and report races as a program runs, other techniques record program events for *post-mortem* race identification (see, for example, [9, 2, 31]). These approaches, however, might be difficult to use for long-running programs. Dynamic race detectors have also been developed for other settings, including for nested fork-join parallelism [24].

Many static analysis techniques for identifying races have also been explored. While static race detection provides the potential to detect all race conditions over all program paths, decidability limitations imply that, for all realistic programming languages, any sound static race detector is incomplete and may produce false alarms. Warlock [36] was an early static race detector system for ANSI C programs for reasoning about lock-based synchronization. Type systems for identifying race conditions have been developed for various languages, including Java [1, 5, 3] and Cyclone, a statically safe variant of C [19]. Aiken and Gay [4] investigate

static race detection in the context of SPMD programs. A variety of other approaches have also been developed, including model checking [7, 41, 25] and dataflow analysis [12, 15], as well as scalable whole-program analyses [26, 40].

7. Conclusions

Race conditions are notoriously difficult to debug. Precise race detectors avoid the programmer-overhead of identifying and eliminating spurious warnings, which are particularly problematic on large programs with complex synchronization. FASTTRACK is a new precise race detection algorithm that achieves better performance than existing algorithms by tracking less information and dynamically adapting its representation of the happens-before relation based on memory access patterns. The FASTTRACK algorithm and adaptive epoch representation is also straightforward to implement, and may be useful in other dynamic analyses for multithreaded software.

Acknowledgments

This work was supported in part by NSF Grants 0341179, 0341387, 0644130, and 0707885. We thank Ben Wood for comments on a draft of this paper. We also thank Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran for helping us to understand the performance of Goldilocks. The proof structure in the Appendix was adapted from techniques developed by Caitlin Sadowski for the SINGLETRACK determinism checker [32].

References

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *TOPLAS*, 28(2):207–255, 2006.

[2] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting data races on weak memory systems. In *ISCA*, pages 234–243, 1991.

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

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

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

[6] CERN. Colt 1.2.0. Available at `http://dsd.lbl.gov/~hoschek/colt/`, 2007.

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

[8] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridhara. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, pages 258–269, 2002.

[9] J.-D. Choi, B. P. Miller, and R. H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *TOPLAS*, 13(4):491–530, 1991.

[10] M. Christiaens and K. D. Bosschere. Accordion clocks: Logical clocks for data race detection. In *Euro-Par*, pages 494–503, 2001.

[11] M. Christiaens and K. D. Bosschere. TRaDe: Data Race Detection for Java. In *International Conference on Computational Science*, pages 761–770, 2001.

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

[13] The Eclipse programming environment, version 3.4.0. Available at `http://www.eclipse.org`, 2009.

[14] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A race and transaction-aware Java runtime. In *PLDI*, pages 245–255, 2007.

[15] D. R. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP*, pages 237–252, 2003.

[16] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. *Sci. Comput. Program.*, 71(2):89–109, 2008.

[17] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI*, pages 293–303, 2008.

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

[19] D. Grossman. Type-safe multithreading in Cyclone. In *TLDI*, pages 13–25, 2003.

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

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

[22] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL*, pages 378–391, 2005.

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

[24] J. M. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Supercomputing*, pages 24–33, 1991.

[25] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.

[26] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI*, pages 308–319, 2006.

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

[28] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *PPOPP*, pages 167–178, 2003.

[29] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*, pages 179–190, 2003.

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

[31] M. Ronsse and K. D. Bosschere. RecPlay: A fully integrated practical record/replay system. *TCS*, 17(2):133–152, 1999.

[32] C. Sadowski, S. N. Freund, and C. Flanagan. SingleTrack: A dynamic determinism checker for multithreaded programs. In *ESOP*, 2009.

[33] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *TOCS*, 15(4):391–411, 1997.

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

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

[36] N. Sterling. Warlock: A static data race analysis tool. In *USENIX Winter Technical Conference*, pages 97–106, 1993.

[37] Sun Microsystems. The `java.util.concurrent` package. Available at `http://java.sun.com/javase/6/docs/api/`, 2008.

[38] C. von Praun and T. Gross. Object race detection. In *OOPSLA*, pages 70–82, 2001.

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

[40] J. W. Voung, R. Jhala, and S. Lerner. Relay: static race detection on millions of lines of code. In *FSE*, pages 205–214, 2007.

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

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

A. FastTrack Correctness Proofs

For a transition $\sigma_a \Rightarrow^a \sigma'_a$, by convention the elements in state σ_a are named C^a, L^a, R^a, and W^a, and the elements in state σ'_a are named $\acute{C}^a$, $\acute{L}^a$, $\acute{R}^a$, and $\acute{W}^a$. We interpret each epoch as a function from threads to clocks:

$$c@t \;\simeq\; (\lambda u.\ \text{if } t = u \text{ then } c \text{ else } 0)$$

This interpretation allows us to overload the function application operator to also apply to the W and R components of the state.

DEFINITION 1 (Well-Formed States). *A state $\sigma = \langle C, L, R, W\rangle$ is* well-formed *if*

1. *for all $u, t \in Tid$, if $t \neq u$ then $C_u(t) < C_t(t)$,*
2. *for all $m \in Lock, t \in Tid$, $L_m(t) < C_t(t)$,*
3. *for all $x \in Var, t \in Tid$, $R_x(t) \leq C_t(t)$,*
4. *for all $x \in Var, t \in Tid$, $W_x(t) \leq C_t(t)$.*

LEMMA 1. *σ_0 is well-formed.*

LEMMA 2 (Preservation of Well-Formedness). *If σ is well-formed and $\sigma \Rightarrow^a \sigma'$ then σ' is well-formed.*

LEMMA 3 (Clocks Imply Happens-Before). *Suppose σ_a is well-formed and $\sigma_a \Rightarrow^{a.\alpha} \sigma_b \Rightarrow^b \sigma'_b$. Let $t = tid(a)$ and $u = tid(b)$. If $C^a_t(t) \leq C^b_u(t)$ then $a <_{a.\alpha.b} b$.*

PROOF If $t = u$ then $a <_{a.\alpha.b} b$ by program order. Otherwise assume $t \neq u$, and the proof proceeds by induction on the length of α. Since σ_a is well-formed,

$$C^a_u(t) < C^a_t(t) \leq C^b_u(t)$$

(Here, C^a_u denotes the clock vector for the thread u in the pre-state of operation a.) Hence there must be some operation $d \in a.\alpha$ that increases the $C_u(t)$ component of the analysis state. Thus:

$$C^a_u(t) \leq C^d_u(t) < C^a_t(t) \leq \acute{C}^d_u(t) \leq C^b_u(t)$$

Since d increases the $C_u(t)$ component of the analysis state, it must be one of $fork(v, u)$, $join(u, v)$, or $acq(u, m)$. We illustrate the case for $d = join(u, v)$. In this case:

$$\begin{array}{ll} & C^a_t(t) \\ \leq & \acute{C}^d_u(t) & \text{from above} \\ = & max(C^d_u(t), C^d_v(t)) & \text{by [FT JOIN]} \\ = & C^d_v(t) & \text{as } C^d_u(t) < C^a_t(t) \end{array}$$

If e is the last operation by v that appears before d in $a.\alpha$, then $C^d_v(t) = C^e_v(t)$, and so $a <_{a.\alpha.b} e <_{a.\alpha.b} d <_{a.\alpha.b} b$ by induction, fork-join order, and program order, respectively.

Note that if there is no v-operation before d in $a.\alpha$ then no operation increases the $C_v(t)$ component of the state, and so we obtain the contradiction that $C^a_v(t) \not< C^a_t(t)$.

□

THEOREM 2 (Soundness). *If $\sigma_0 \Rightarrow^\alpha \sigma'$ then α is race-free.*

PROOF Suppose α has a race condition, and thus contains an operation a followed by a later operation b where a conflicts with b and $a \not<_\alpha b$. The proof proceeds by lexicographic induction on the length of α, and on the number of intervening operations between a and b. Without loss of generality, we assume the prefix of α before b is race-free, and that b is not in a race with any operation in α before a. Let $t = tid(a)$ and $u = tid(b)$. Clearly $t \neq u$. By Lemma 3, $C^a_t(t) > C^b_u(t)$.

We illustrate the argument for $a = wr(t, x)$ and $b = rd(u, x)$.

- If the rule for b is [FT READ SAME EPOCH], then there must have been a preceding read $d = rd(u, x)$.

  If d is after a, then by induction $a <_\alpha d <_\alpha b$.

  If d is before a, then since d and b are in the same epoch, there can be no intervening fork or release operations by thread u (as those increase $C_u(u)$ and so change the epoch) and hence $d \not<_\alpha a$, and so there is an earlier race condition in this trace.

- Otherwise, if the rule for b is not [FT READ SAME EPOCH], then

  $$\begin{array}{rcll} (C^a_t(t))@t & = & \acute{W}^a & \text{by [FT WRITE } \ldots \text{]} \\ & = & W^b & \text{as no intervening writes to } x \\ & \preceq & C^b & \text{by [FT READ } \ldots \text{]} \end{array}$$

  Hence $C^a_t(t) \leq C^b_u(t)$, and so we have a contradiction.

□

We introduce the abbreviation:

$$K^a \;=\; \begin{cases} \acute{C}^a & \text{if } a \text{ a } join \text{ or } acquire \text{ operation} \\ C^a & \text{otherwise} \end{cases}$$

LEMMA 4. *Suppose σ is well-formed and $\sigma \Rightarrow^\alpha \sigma'$ and $a, b \in \alpha$. Let $t = tid(a)$ and $u = tid(b)$. If $a <_\alpha b$ then $K^a(t) \sqsubseteq K^b(u)$.*

PROOF By induction on the derivation of $a <_\alpha b$. We choose to use a happens-before derivation that applies program-order in preference to other rules wherever possible. □

THEOREM 3 (Completeness). *If α is race-free then $\sigma_0 \Rightarrow^\alpha \sigma$.*

PROOF Suppose $\alpha = \beta.a.\gamma$ such that operation a is *stuck*, that is:

$$\sigma_0 \Rightarrow^\beta \sigma' \not\Rightarrow^a \cdots$$

We consider all possible operations for a that could get stuck, and illustrate the argument for $a = rd(t, x)$. If this read is stuck, then $W^a_x \not\preceq C^a_t$. We consider the last write $b = wr(u, x)$ preceding a. In all cases,

$$(C^b_u(u))@u = \acute{W}^b_x = W^a_x \not\preceq C^a_t$$

Hence

$$K^b_u(u) = C^b_u(u) \not\leq C^a_t(u) = K^a_t(u)$$

By Lemma 4, $b \not<_\alpha a$, and so α has a write-read race condition.

□