

FastTrack: Efficient and Precise Dynamic Race Detection

By Cormac Flanagan and Stephen N. Freund

Abstract

Multithreaded programs are notoriously prone to race conditions. Prior work developed precise dynamic race detectors that never report false alarms. However, these checkers employ expensive data structures, such as vector clocks (VCs), that result in significant performance overhead.

This paper exploits the insight that the full generality of VCs is not necessary in most cases. That is, we can replace VCs with an adaptive lightweight representation that, for almost all operations of the target program, requires constant space and supports constant-time operations. Experimental results show that the resulting race detection algorithm is over twice as fast as prior precise race detectors, with no loss of precision.

1. INTRODUCTION

Multithreaded programs are prone to race conditions and other concurrency errors such as deadlocks and violations of expected atomicity or determinism properties. The broad adoption of multicore processors is exacerbating these problems, both by driving the development of multithreaded software and by increasing the interleaving of threads in existing multithreaded systems.

A race condition occurs when two threads concurrently perform memory accesses that *conflict*. Accesses conflict when they read or write the same memory location and at least one of them is a write. In this situation, the order in which the two conflicting accesses are performed can affect the program's subsequent state and behavior, likely with unintended and erroneous consequences.

Race conditions are notoriously problematic because they typically cause problems only on rare interleavings, making them difficult to detect, reproduce, and eliminate. Consequently, much prior work has focused on static and dynamic analysis tools for detecting race conditions.

To maximize test coverage, race detectors use a very broad notion of when two conflicting accesses are considered concurrent. The accesses need not be performed at exactly the same time. Instead, the central requirement is that there is no “synchronization dependence” between the two accesses, such as the dependence between a lock release by one thread and a subsequent lock acquire by a different thread. These various kinds of synchronization dependencies form a partial order over the instructions in the execution trace called the *happens-before relation*.¹³ Two memory accesses are then considered to be *concurrent* if they are not ordered by this happens-before relation.

In this paper, we focus on online dynamic race detectors, which generally 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. Note that there are typically many possible traces for a particular program, depending on test inputs and scheduling choices. Precise dynamic race detectors do not reason about all possible traces, however, and may not identify races that occur only when other code paths are taken. While full coverage is desirable, it comes at the cost of potential false alarms because of the undecidability of the halting problem. To avoid these false alarms, precise race detectors focus on detecting only race conditions that occur on the observed trace.

Typically, precise detectors represent the happens-before relation with *vector clocks* (VCs),¹⁴ as in the DJIT⁺ race detector.¹⁶ Vector clocks are expensive to maintain, however, because a VC encodes information about each thread in a system. Thus, if the target program has n threads, each VC requires $O(n)$ storage space and VC operations (such as comparison) require $O(n)$ time. Since a VC must be maintained for each memory location and modified on each access to that location, this $O(n)$ time and space overhead precludes the use of VC-based race detectors in many settings.

A variety of alternative *imprecise* race detectors have been developed, which may provide improved performance (and sometimes better coverage), but which report false alarms on some race-free programs. For example, Eraser's LockSet algorithm¹⁸ 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 limited happens-before reasoning to improve precision in such situations.^{15, 16, 22}

Other optimizations include using static analyses or dynamic escape analyses^{3, 21} or using “accordion” VCs that reduce space overheads for programs with shortlived threads.⁵ Alternative approaches record program events for *post-mortem* race identification.^{1, 4, 17}

Although these imprecise tools successfully detect race conditions, their potential to generate many false alarms limits their effectiveness. Indeed, it has proven surprisingly difficult and time consuming to identify the real errors among

The original version of this paper was published in the *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2009.

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 code maintainer. Even worse, additional real bugs (e.g., deadlocks or performance problems) 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.

This paper exploits the insight that, while VCs provide a general mechanism for representing the happens-before relation, their full generality is not actually necessary in most cases. 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 and constant-space overhead for these common cases, without any loss of precision or correctness.

In more detail, a VC-based race detector such as DJIT⁺ records the time 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 on x have been detected so far, and records information only about the *very last* write to x . Specifically, FASTTRACK records 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 on x have been detected, and FASTTRACK records only the epoch of the last read from such data. FASTTRACK adaptively switches from epochs to VCs when necessary (e.g., when data becomes read-shared) in order to guarantee no loss of precision. It also switches from VCs back to lightweight epochs when possible (e.g., when read-shared data is subsequently updated).

Using these 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.

In addition to improving performance, the epoch representation also reduces space overhead. A VC-based race detector requires $O(n)$ space 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 implemented six different dynamic race detectors: FASTTRACK plus five other race detectors described in the literature. Experimental results on Java benchmarks, including the Eclipse programming environment, show that FASTTRACK outperforms the other tools. For example, it provides almost a 10x speedup over a traditional VC-based race detector and a 2.3x speedup over the DJIT⁺ algorithm. It also provides a substantial increase in precision over ERASER, with no loss in performance.

2. PRELIMINARIES

2.1. Multithreaded program traces

We begin with some terminology and definitions regarding multithreaded execution traces. A program consists of

a number of concurrently executing threads, each with a thread identifier $t \in Tid$. These threads manipulate variables $x \in Var$ and locks $m \in Lock$. A *trace* α captures an execution of a multithreaded program by listing the sequence of *operations* performed by the various threads. The operations that a thread t can perform include:

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

The *happens-before relation* ($<_{\alpha}$) for a trace α is a partial order over the operations in α that captures control and synchronization dependencies. In particular, the relation $a <_{\alpha} b$ holds whenever operation a occurs before operation b in α and one of the following conditions applies:

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

In addition, the happens-before relation is transitively closed: that is, if $a <_{\alpha} b$ and $b <_{\alpha} c$ then $a <_{\alpha} c$.

If a happens before b , then we also say 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.

2.2. Vector clocks and the DJIT⁺ algorithm

Before presenting the FASTTRACK algorithm, we briefly review the DJIT⁺ online race detection algorithm,¹⁶ which is based on VCs.¹⁴ A VC

$$V : Tid \rightarrow Nat$$

records a clock for each thread in the system. Vector clocks are partially-ordered ($\sqsubseteq$) in a pointwise manner, with an associated join operation ($\sqcup$) and minimal element ($\perp_v$). In addition, the helper function inc_t increments the t -component of a VC:

$$V_1 \sqsubseteq V_2 \text{ iff } \forall t. V_1(t) \leq V_2(t)$$

$$V_1 \sqcup V_2 = \lambda t. \max(V_1(t), V_2(t))$$

$$\perp_v = \lambda t. 0$$

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

In DJIT⁺, each thread has its own clock that is incremented at each lock release operation. Each thread t also keeps a VC $\mathbb{C}_t$ 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 VC $\mathbb{L}_m$ for each lock m . These VCs 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 VCs, $\mathbb{R}_x$ 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, $\mathbb{W}_x \subseteq \mathbb{C}_u$. A write to x by thread u is race-free provided that the write happens after all previous accesses to that variable, that is, $\mathbb{W}_x \subseteq \mathbb{C}_u$ and $\mathbb{R}_x \subseteq \mathbb{C}_u$.

As an example, consider the execution trace fragment shown in Figure 1, where we include the relevant portion of the DJIT⁺ instrumentation state: the VCs $\mathbb{C}_0$ and $\mathbb{C}_1$ for threads 0 and 1; and the VCs $\mathbb{L}_m$ 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 VC, but the target program may of course contain additional threads.^a

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 VCs:

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

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, since each VC requires $O(n)$ space and each VC operation (copying, comparing, joining, etc.) requires $O(n)$ time.

Empirical benchmark data indicates that reads and writes operations account for the vast majority (over 96%) of monitored operations. The key insight behind FASTTRACK is that the full generality of VCs 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 VC operations.

We begin by providing an overview of how our analysis catches each type of race condition on a memory location. A race condition is either: a *write-write race condition* (where a write is concurrent with a later write); a *write-read race condition* (where a write is concurrent with a later read); or a *read-write race condition* (where a read is concurrent with a later write).

Detecting Write-Write Races: We first consider how to efficiently analyze write operations. At the second write operation in the trace in Figure 1, DJIT⁺ compares the VCs $\mathbb{W}_x \subseteq \mathbb{C}_1$ to determine whether there is a race. A careful inspection reveals, however, that it is not necessary to record the *entire* VC $\langle 4, 0, \dots \rangle$ from the first write to x . Assuming no races have been detected on x so far, then all writes to x are totally ordered by the happens-before relation, and 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 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 VCs, an epoch requires only constant space and supports constant-time operations.

An epoch $c@t$ *happens before* a VC 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:

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

We use $\perp_t$ to denote a minimal epoch $0@0$.

Using this optimized representation, FASTTRACK analyzes the trace from Figure 1 using a more compact instrumentation state that records only a write epoch $\mathbb{W}_x$ for variable x , rather than the entire VC $\mathbb{W}_x$, reducing space overhead, as

Figure 1. Execution trace under DJIT⁺.

$\mathbb{C}_0$	$\mathbb{C}_1$	$\mathbb{L}_m$	$\mathbb{W}_x$
$\langle 4, 0, \dots \rangle$	$\langle 0, 8, \dots \rangle$	$\langle 0, 0, \dots \rangle$	$\langle 0, 0, \dots \rangle$
$wr(0, x) \downarrow$			
$\langle 4, 0, \dots \rangle$	$\langle 0, 8, \dots \rangle$	$\langle 0, 0, \dots \rangle$	$\langle 4, 0, \dots \rangle$
$rel(0, m) \downarrow$			
$\langle 5, 0, \dots \rangle$	$\langle 0, 8, \dots \rangle$	$\langle 4, 0, \dots \rangle$	$\langle 4, 0, \dots \rangle$
	$acq(1, m) \downarrow$		
$\langle 5, 0, \dots \rangle$	$\langle 4, 8, \dots \rangle$	$\langle 4, 0, \dots \rangle$	$\langle 4, 0, \dots \rangle$
	$wr(1, x) \downarrow$		
$\langle 5, 0, \dots \rangle$	$\langle 4, 8, \dots \rangle$	$\langle 4, 0, \dots \rangle$	$\langle 4, 8, \dots \rangle$

^a For clarity, we present a variant of the DJIT⁺ algorithm where some clocks are one less than in the original formulation.¹⁶ This revised algorithm has the same performance as the original but is slightly simpler and more directly comparable to FASTTRACK.

Figure 2. Execution trace under FASTTRACK.

C_0	C_1	L_m	W_x
$\langle 4, 0, \dots \rangle$	$\langle 0, 8, \dots \rangle$	$\langle 0, 0, \dots \rangle$	$\perp_e$
$\downarrow wr(0, x)$			
$\langle 4, 0, \dots \rangle$	$\langle 0, 8, \dots \rangle$	$\langle 0, 0, \dots \rangle$	$4@0$
$\downarrow rel(0, m)$			
$\langle 5, 0, \dots \rangle$	$\langle 0, 8, \dots \rangle$	$\langle 4, 0, \dots \rangle$	$4@0$
	$\downarrow acq(1, m)$		
$\langle 5, 0, \dots \rangle$	$\langle 4, 8, \dots \rangle$	$\langle 4, 0, \dots \rangle$	$4@0$
	$\downarrow wr(1, x)$		
$\langle 5, 0, \dots \rangle$	$\langle 4, 8, \dots \rangle$	$\langle 4, 0, \dots \rangle$	$8@1$

shown in Figure 2. (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:

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

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 VC comparison “ $\sqsubseteq$ ” with the $O(1)$ -time comparison “ $\preceq$ ”.

Detecting Write-Read Races: Detecting write-read races under the new representation is also straightforward. On each read from x with current VC 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 in race-free programs, reads are not necessarily totally ordered. 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. Thus, we may need to record an entire VC R_x , in which $R_x(t)$ records the clock of the last read from x by thread t .

We can avoid keeping a complete VC in many cases, however. Our examination of data access patterns across a variety of multithreaded Java programs indicates that read operations 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 accesses 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 the read history of each variable that is optimized for the common case of totally-ordered reads, while still retaining the full precision of VCs 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 write 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 VC, but still handles read operations in $O(1)$ time. Since such data is typically read-shared, writes to such variables are rare, and their analysis overhead is negligible.

3.1. Analysis details

Based on the above intuition, we now describe the FASTTRACK algorithm in detail. Our analysis is an online algorithm whose analysis state consists of four components:

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

The analysis starts with $C_t = inc_t(\perp_v)$, since the first operations of all threads are not ordered by happens-before. In addition, initially $L_m = \perp_v$ and $R_x = W_x = \perp_e$.

Figure 3 presents the key details of how FASTTRACK (left column) and DJIT⁺ (right column) handle read and write operations of the target program. For each read or write operation, the relevant rules are applied in the order presented until one matches the current instrumentation state. If an assertion fails, a race condition exists. The figure shows the instruction frequencies observed in the programs described in Section 4, 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. Expensive $O(n)$ -time operations are highlighted in grey.

Read Operations: The first four rules provide various alternatives for analyzing a read operation $rd(t, x)$. The first rule

Figure 3. FASTTRACK race detection algorithm and its comparison to DJIT⁺.

FASTTRACK State: $C_t : VC$ $L_m : VC$ $W_x : Epoch$ $R_x : Epoch \cup VC$		DJIT⁺ State: $C_t : VC$ $L_m : VC$ $W_x : VC$ $R_x : VC$	
When Thread t performs $rd(t, x)$:		82.3% of all Operations	
[FT READ SAME EPOCH] if $R_x = E_t$ then skip endif		[DJIT⁺ READ SAME EPOCH] if $R_x(t) = C_t(t)$ then skip endif	
[FT READ SHARED] if $R_x \in VC$ then assert $W_x \preceq C_t$ $R_x(t) := C_t(t)$ endif		[DJIT⁺ READ] if $R_x(t) \neq C_t(t)$ then assert $W_x \sqsubseteq C_t$ $R_x(t) := C_t(t)$ endif	
[FT READ EXCLUSIVE] if $R_x \in Epoch$ and $R_x \preceq C_t$ then assert $W_x \preceq C_t$ $R_x := E_t$ endif		[DJIT⁺ READ] if $R_x(t) \neq C_t(t)$ then assert $W_x \sqsubseteq C_t$ $R_x(t) := C_t(t)$ endif	
[FT READ SHARE] if $R_x \in Epoch$ then let $c@u = R_x$ assert $W_x \preceq C_t$ $R_x := \perp_v[t \mapsto C_t(t), u \mapsto c]$ endif			
When Thread t performs $wr(t, x)$:		14.5% of all Operations	
[FT WRITE SAME EPOCH] if $W_x = E_t$ then skip endif		[DJIT⁺ WRITE SAME EPOCH] if $W_x(t) = C_t(t)$ then skip endif	
[FT WRITE EXCLUSIVE] if $R_x \in Epoch$ then assert $R_x \preceq C_t$ assert $W_x \preceq C_t$ $W_x := E_t$ endif		[DJIT⁺ WRITE] if $W_x(t) \neq C_t(t)$ then assert $W_x \sqsubseteq C_t$ assert $R_x \sqsubseteq C_t$ $W_x(t) := C_t(t)$ endif	
[FT WRITE SHARED] if $R_x \in VC$ then assert $R_x \sqsubseteq C_t$ assert $W_x \preceq C_t$ $W_x := E_t$ $R_x := \perp_e$ endif		[DJIT⁺ WRITE] if $W_x(t) \neq C_t(t)$ then assert $W_x \sqsubseteq C_t$ assert $R_x \sqsubseteq C_t$ $W_x(t) := C_t(t)$ endif	

[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. If R_x is already a VC, 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 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 thread's current epoch. For the more general situation where the current read is concurrent with the previous read, [FT READ SHARE] allocates a VC 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 rule [DJIT⁺ READ] *always* executes an $O(n)$ -time VC 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

Figure 4. Synchronization, threading operations.

Other:	3.3% of all Operations
When Thread t performs $acq(t, m)$:	
$C_t := C_t \sqcup L_m$	
When Thread t performs $rel(t, m)$:	
$L_m := C_t$	
$C_t := inc_t(C_t)$	
When Thread t performs $fork(t, u)$:	
$C_u := C_u \sqcup C_t$	
$C_t := inc_t(C_t)$	
When Thread t performs $join(t, u)$:	
$C_t := C_t \sqcup C_u$	
$C_u := inc_u(C_u)$	

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 VC, [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 a VC comparison on 29.0% of writes.

Other Operations: Figure 4 shows how FASTTRACK handles synchronization operations. These operations are rare, and 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 5 illustrates how FASTTRACK dynamically adapts the representation for the read history R_x of a variable x . Initially, R_x is $\perp_e$, 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 VC representation $\langle 8, 1, \dots \rangle$ 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 $\perp_e$, which also switches x back into epoch mode and so

Figure 5. Adaptive read history representation.

C_0	C_1	W_x	R_x
$\langle 7, 0, \dots \rangle$	$\langle 0, 1, \dots \rangle$	$\perp_e$	$\perp_e$
$\downarrow wr(0, x)$			
$\langle 7, 0, \dots \rangle$	$\langle 0, 1, \dots \rangle$	7@0	$\perp_e$
$\downarrow fork(0, 1)$			
$\langle 8, 0, \dots \rangle$	$\langle 7, 1, \dots \rangle$	7@0	$\perp_e$
	$\downarrow rd(1, x)$		
$\langle 8, 0, \dots \rangle$	$\langle 7, 1, \dots \rangle$	7@0	1@1
$\downarrow rd(0, x)$			
$\langle 8, 0, \dots \rangle$	$\langle 7, 1, \dots \rangle$	7@0	$\langle 8, 1, \dots \rangle$
	$\downarrow rd(1, x)$		
$\langle 8, 0, \dots \rangle$	$\langle 7, 1, \dots \rangle$	7@0	$\langle 8, 1, \dots \rangle$
$\downarrow join(0, 1)$			
$\langle 8, 1, \dots \rangle$	$\langle 7, 2, \dots \rangle$	7@0	$\langle 8, 1, \dots \rangle$
$\downarrow wr(0, x)$			
$\langle 8, 1, \dots \rangle$	$\langle 7, 2, \dots \rangle$	8@0	$\perp_e$
$\downarrow rd(0, x)$			
$\langle 8, 1, \dots \rangle$	$\langle 7, 2, \dots \rangle$	8@0	8@0

optimizes later accesses to x . The last read in the trace then sets R_x to a nonminimal epoch.

4. EVALUATION

To validate FASTTRACK, we implemented it as a component of the ROADRUNNER dynamic analysis framework for multithreaded Java programs.¹⁰ ROADRUNNER takes as input a compiled Java target program and inserts instrumentation code into the target to generate an event stream of memory and synchronization operations. Back-end checking tools process these events as the target executes. The FASTTRACK implementation extends the algorithm described so far to handle additional Java primitives, such as `volatile` variables and `wait()`, as outlined previously.⁸ Some of the benchmarks contain faulty implementations of barrier synchronization.⁹ FASTTRACK contains a specialized analysis to compensate for these bugs.

We compare FASTTRACK's precision and performance to six other analyses implemented in the same framework:

- EMPTY, a trivial checker that performs no analysis and is used to measure the overhead of ROADRUNNER
- ERASER,¹⁸ an imprecise race detector based on the LockSet algorithm described in Section 1
- GOLDOLOCKS, a precise race detector based on an extended notion of LockSets⁷
- BASICVC, a traditional 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
- DJIT⁺, a high-performance VC-based race detector¹⁶ described in Section 2

– MULTIRACE, a hybrid LockSet/DJIT⁺ race detector¹⁶

4.1. Performance and precision

Table 1 lists the size, number of threads, and uninstrumented running times for a variety of benchmark programs drawn from the Java Grande Forum,¹² Standard Performance Evaluation Corporation,¹⁹ and elsewhere.^{2,7,11,21} All timing measurements are the average of 10 test runs. Variability across runs was typically less than 10%.

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.

The variations in slowdowns for different programs are not uncommon for dynamic race condition checkers. Different programs exhibit different memory access and synchronization patterns, some of which 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. 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. All eight warnings from FASTTRACK reflect real race conditions. Some of these are benign (as in `tsp`, `mtrt`, and `jbb`) but others can impact program behavior (as in `raytracer` and `hedc`).^{15, 20, 21}

Table 1. Benchmark results. Programs marked with “*” are not compute-bound and are excluded from average slowdowns.

Program	Size (loc)	Thread Count	Base Time (s)	Instrumented Time (slowdown)							Warnings					
				EMPTY	ERASER	MULTIRACE	GOLDOLOCKS	BASICVC	DJIT ⁺	FASTTRACK	ERASER	MULTIRACE	GOLDOLOCKS	BASICVC	DJIT ⁺	FASTTRACK
colt	111,421	11	16.1	0.9	0.9	0.9	1.8	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	–	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	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	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	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	1.0	1.0	1.0	1	0	0	0	0	0
sor	1,005	4	0.2	4.4	9.1	16.9	63.2	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	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	0	0	0	0	0	0
hedc*	24,937	6	5.9	1.1	0.9	1.1	1.1	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 slowdown/total warnings				4.1	8.6	21.7	31.6	89.8	20.2	8.5	27	5	3	8	8	8

ERASER Comparison: Our reimplementation of ERASER incurs an overhead of 8.7x, which is competitive with similar Eraser implementations built on top of unmodified JVMs.¹⁵ 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. Augmenting our ERASER implementation to reason about additional synchronization constructs, such as `fork/join` or `wait/notify` operations,^{16, 22} would eliminate some of these spurious warnings, but not all. On `hedc`, ERASER reported a spurious warning but also missed two of the real race conditions reported by FASTTRACK, due to an (intentional) unsoundness in how the Eraser algorithm reasons about thread-local and read-shared data.¹⁸

BASICVC and DJIT⁺ Comparison: DJIT⁺ and BASICVC reported exactly the same race conditions as FASTTRACK. That is, all three checkers provide identical precision. However, FASTTRACK outperforms the other checkers. 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 VCs. Across all benchmarks, DJIT⁺ allocated more over 790 million VCs, whereas FASTTRACK allocated only 5.1 million. DJIT⁺ performed over 5.1 billion $O(n)$ -time VC operations, while FASTTRACK performed only 17 million. The memory overhead for storing the extra VCs leads to significant cache performance degradation in some programs, particularly those that randomly access large arrays. These tools are likely to incur even greater overhead when checking programs with larger numbers of threads.

MULTIRACE Comparison: MULTIRACE maintains DJIT⁺'s instrumentation state, as well as a lock set for each memory location.¹⁶ The checker updates the lock set for a location on the first access in an epoch, and full VC comparisons are performed only after this lock set becomes empty. This synthesis substantially reduces the number of VC 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.

Our reimplementation of the MULTIRACE algorithm exhibited performance comparable to DJIT⁺. Performance of MULTIRACE (and, in fact, all of our other checkers) can be improved by adopting a *coarse-grain* analysis in which all fields of an object are represented as a single "logical location" in the instrumentation state.^{16, 22}

GOLDILOCKS Comparison: GOLDILOCKS⁷ is a precise race detector that does not use VCs 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 to optimize, and ideally requires tight integration with the underlying virtual machine and garbage collector, which is not possible under

ROADRUNNER. Because of these difficulties, GOLDILOCKS reimplemented in ROADRUNNER incurred a slowdown of 31.6x across our benchmarks, but ran out of memory on `lufact`. Our GOLDILOCKS reimplementation missed three races in `hedc`, due to an unsound performance optimization for handling thread-local data efficiently.⁷ We believe some performance improvements are possible, for both GOLDILOCKS and the other tools, by integration into the virtual machine.

4.2. Checking eclipse for race conditions

To validate FASTTRACK in a more realistic setting, we also applied it to five common operations in the Eclipse development environment.⁶ These include launching Eclipse, importing a project, rebuilding small and large workspaces, and starting the debugger. The checking overhead for these operations is as follows:

Operation	Base Time (s)	Instrumented Time (Slowdown)			
		EMPTY	ERASER	DJIT ⁺	FASTTRACK
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

ERASER reported potential races on 960 distinct field and array accesses for these five tests, largely because Eclipse uses many synchronization idioms that ERASER cannot handle, such as `wait()` and `notify()`, semaphores, and readers-writer locks. FASTTRACK reported 27 distinct warnings, 4 of which were subsequently verified to be potentially destructive.⁹ DJIT⁺ reported 28 warnings, which overlapped heavily with those reported by FASTTRACK, but scheduling differences led to several being missed and several new (benign) races being identified. 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.

5. CONCLUSION

Race conditions are difficult to find and fix. Precise race detectors avoid the programmer-overhead of identifying and eliminating spurious warnings, which are particularly problematic when using imprecise checkers on large programs with complex synchronization. Our FASTTRACK analysis 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. We have used FASTTRACK to identify data races in programs as large as the Eclipse programming environment, and also to improve the performance of other analyses that rely on precise data race information, such as serializability checkers.⁸ The FASTTRACK algorithm and adaptive epoch representation is 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 implementing Goldilocks in ROADRUNNER and for comments on a draft of this paper, and Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran for their assistance with Goldilocks.



References

1. Adve, S.V., Hill, M.D., Miller, B.P., Netzer, R.H.B. Detecting data races on weak memory systems. In *ISCA* (1991), 234–243.
2. CERN. Colt 1.2.0. Available at <http://dsd.lbl.gov/~hoschek/colt/> (2007).
3. Choi, J.-D., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., Sridhara, M. Efficient and precise data race detection for multithreaded object-oriented programs. In *PLDI* (2002), 258–269.
4. Choi, J.-D., Miller, B.P., Netzer R.H.B. Techniques for debugging parallel programs with flowback analysis. *TOPLAS* 13, 4 (1991), 491–530.
5. Christiaens, M., Bosschere, K.D. TRaDe: Data race detection for Java. In *International Conference on Computational Science* (2001), 761–770.
6. The Eclipse programming environment, version 3.4.0. Available at <http://www.eclipse.org>, 2009.
7. Elmas, T., Qadeer, S., Tasiran, S. Goldilocks: A race and transaction-aware Java runtime. In *PLDI* (2007), 245–255.
8. Flanagan, C., Freund, S.N. FastTrack: Efficient and precise dynamic race detection. In *PLDI* (2009), 121–133.
9. Flanagan, C., Freund, S.N. Adversarial memory for detecting destructive races. In *PLDI* (2010), 244–254.
10. Flanagan, C., Freund, S.N. The RoadRunner dynamic analysis framework for concurrent programs. In *PASTE* (2010), 1–8.
11. Fleury, E., Sutre, G. Raja, version 0.4.0-pre4. Available at <http://raja.sourceforge.net/>, 2007.
12. Java Grande Forum. Java Grande benchmark suite. Available at <http://www.javagrande.org/>, 2008.
13. Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558–565.
14. Mattern, F. Virtual time and global states of distributed systems. In *Workshop on Parallel and Distributed Algorithms*, 1988.
15. O'Callahan, R., Choi J.-D. Hybrid dynamic data race detection. In *PPOPP* (2003), 167–178.
16. Pozniansky, E., Schuster, A. MultiRace: Efficient on-the-fly data race detection in multithreaded C++ programs. *Concurrency and Computation: Practice and Experience* 19, 3 (2007), 327–340.
17. Ronsse, M., Bosschere, K.D. RecPlay: A fully integrated practical record/replay system. *TCS* 17, 2 (1999), 133–152.
18. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.E. Eraser: A dynamic data race detector for multi-threaded programs. *TOCS* 15, 4 (1997), 391–411.
19. Standard Performance Evaluation Corporation. SPEC benchmarks. <http://www.spec.org/>, 2003.
20. von Praun, C., Gross, T. Object race detection. In *OOPSLA*, 2001, 70–82.
21. von Praun, C., Gross, T. Static conflict analysis for multi-threaded object-oriented programs. In *PLDI* (2003), 115–128.
22. Yu, Y., Rodeheffer, T., Chen, W. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP* (2005), 221–234.

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

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

© 2010 ACM 0001-0782/10/1100 \$10.00

Take Advantage of ACM's Lifetime Membership Plan!

- ◆ **ACM Professional Members** can enjoy the convenience of making a single payment for their entire tenure as an ACM Member, and also be protected from future price increases by taking advantage of **ACM's Lifetime Membership** option.
- ◆ **ACM Lifetime Membership** dues may be tax deductible under certain circumstances, so becoming a Lifetime Member can have additional advantages if you act before the end of 2010. (Please consult with your tax advisor.)
- ◆ Lifetime Members receive a certificate of recognition suitable for framing, and enjoy all of the benefits of **ACM Professional Membership**.

Learn more and apply at:

<http://www.acm.org/life>



Association for
Computing Machinery

Advancing Computing as a Science & Profession