The FASTTRACK2 Race Detector

Working Draft — Technical Report CSTR201701, Revised: February 25, 2017

Cormac Flanagan
UC Santa Cruz

Stephen Freund
Williams College

Abstract

FASTTRACK is a precise and efficient dynamic race detector. In the years since we initially designed that analysis, we have experimented with a number of different variants and implementation strategies. Drawing on the insights from those experiences, we present here the FASTTRACK2 analysis and idealized implementation. FASTTRACK2 improves upon the original in several ways: it refines several analysis rules, it is more readily translated into an executable implementation, and that implementation enjoys a simpler correctness argument while still exhibiting improved performance when compared to our earlier prototype. Our Java implementation of FASTTRACK2 is available as part of the ROADRUNNER 0.5 distribution [4].

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 [9], 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 and dynamic analysis tools for detecting race conditions. Please see other recent work [5, 16] for a summary of many race detection techniques.

Typically, the happens-before relation is represented using *vector clocks* (VCs) [13], as in the DJIT$^+$ race detector [14]. 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. FASTTRACK2 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 FASTTRACK2 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.

In this paper, we present the FASTTRACK2 analysis and implementation, which revisits a number of design and implementation choices we made during our earlier work. In more detail, we

- refine several analysis rules based on our empirical studies and to simplify implementations;
- express the analysis as a transition system more readily translated into executable code than the original;
- present an idealized implementation of FASTTRACK2 and show corresponds to the specification; and
- implement FASTTRACK2 in the ROADRUNNER framework for Java.

A primary motivation of this work is to improve our ability to reason about the implementation of race detection analysis. Achieving good performance for a dynamic race detector like FASTTRACK can require sophisticated synchronization strategies. Over the years, we have explored prototypes using everything from basic mutual exclusion locks to various optimistic concurrency control mechanisms. While the more sophisticated strategies have yielded much better performance, their use has come with a cost; they rely on subtleties of the analysis rules, complex data invariants, and thorough knowledge of the underlying Java memory model [12] and ROADRUNNER framework. As such, those implementations have been difficult to design, implement, debug, and maintain.

Thus, a primary contribution of this work is an analysis specification and implementation strategy that is easier to understand, implement, and reason about while still employing the synchronization mechanism with good performance. Indeed, as we demonstrate, FASTTRACK2 exhibits performance slightly better than our previous versions.

Our development proceeds as follows:

- We formalize the FASTTRACK2 analysis algorithm and sketch the correctness proof showing that it precisely detects data races (Sections 2 and 3).
- We present an idealized implementation of the analysis as a collection of event handling methods written in a Java-like language (Section 5).
- We show that the implementation's event handling methods are serializable using existing techniques [6, 15, 7]. This facilitates reasoning about the correctness of the code (Section 6).
- We discuss the correspondence between the event handlers and formal rules (Section 7).
- We present a Java implementation based on the idealized code and validate its performances on two common benchmark suites (Sections 8 and 9).

2. 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 $t \in Tid$, and 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.

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

$$u, t \in Tid \qquad x, y \in Var \qquad m \in Lock$$

The set of operations that a thread t can perform include:

- $rd(t,x)$ and $wr(t,x)$ read and write a value from x;
- $acq(t,m)$ and $rel(t,m)$ acquire and release a lock m;
- $fork(t,u)$ forks a new thread u; and
- $join(t,u)$ 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 previously acquire, (3) each thread is forked at most once (4) there are no instructions of a thread u preceding an instruction $fork(t,u)$ or following an instruction $join(t,u)$, and (5) there is at least one instruction of thread u between $fork(t,u)$ and $join(t',u)$.

3. FastTrack2 Algorithm

We present in this section the formal specification of the FastTrack2 analysis, describe how it differs from the original FastTrack analysis, and outline the proof that the analysis is precise. We refer the reader to our earlier paper on FastTrack for a more comprehensive discussion of the intuition behind the analysis [3].

3.1 Preliminaries

A pair of a thread t and a clock c forms an *epoch*, denoted $t@c$. Although rather simple, epochs provide the crucial lightweight representation for recording sufficiently-precise aspects of the happens-before relation efficiently. We define several operations on epochs as follows:

$$\begin{array}{rcl} t@c_1 < t@c_2 & \text{iff} & c_1 < c_2 \\ t@c_1 \leq t@c_2 & \text{iff} & c_1 \leq c_2 \\ max(t@c_1, t@c_2) & = & t@max(c_1, c_2) \\ t@c + 1 & = & t@(c+1) \end{array}$$

The $<$, $\leq$, and max operations are undefined if the two operands are epochs for different threads. Epochs require only constant space, independent of the number of threads in the program, and all operations, including copying an epoch, are constant-time operations.

A vector clock $VC : Tid \to Epoch$ records an epoch 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:

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

We assume all vector clocks are well-formed in that for all i, $V(i)$ contains an epoch for thread i, *e.g.* $V(i) = i@c$ for some c.

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

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

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 $1@0$.)

3.2 Analysis State

FastTrack2 is an online algorithm that maintains an *analysis state* $S \in State$; when the target program performs an operation a, the analysis updates its state via the relation $S \Rightarrow^a S'$. $State$ is conceptually the union of three maps

$$\begin{array}{rcl} S \in State : & Tid & \to ThreadState \\ & \cup\, Lock & \to LockState \\ & \cup\, Var & \to VarState \end{array}$$

where

$$\begin{array}{rl} ThreadState = & \{\ V\colon VC, E\colon Epoch\ \} \\ LockState = & \{\ V\colon VC\ \} \\ VarState = & \{\ V\colon VC, R\colon (Epoch \cup \{\textsc{Shared}\}), W\colon Epoch\ \} \end{array}$$

We use S_t, S_m, S_x to index the various parts of S. These state components have the following meaning:

- $S_t.V$ is the current vector clock of the thread t, and $S_t.E$ is the current epoch. The epoch is cached for efficiency in the implementation, and the analysis maintains the invariant that $S_t.E = S_t.V(t)$.
- $S_m.V$ is the vector clock capturing the time of the last release of lock m.
- $x.W$ identifies the epoch of the last write to x. Also, $S_x.R$ identifies the epoch of the last read from x, or is the special value SHARED if there have been unordered reads of x. In that case $S_x.V$ is the join of all reads of x.

The initial analysis state is:

$$\begin{array}{l} S_0 = \lambda t.\{V : inc_t(\bot_V), E : t@1\} \\ \quad \cup\, \lambda m.\{V : \bot_V\} \\ \quad \cup\, \lambda x.\{V : \bot_V, R : \bot_e, W : \bot_e\} \end{array}$$

3.3 Analysis Rules

Figure 1 presents the key details of how FastTrack2 handles operations of the target program. The rules of the form $S \Rightarrow^a$

[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.

$\boxed{S \Rightarrow^{a} S',\ \ S \Rightarrow^{a} \textsc{Error}}$

[READ SAME EPOCH]
$$\frac{S_x.R = S_t.E}{S \Rightarrow^{rd(t,x)} S}$$

[READ SHARED SAME EPOCH]
$$\frac{S_x.R = \textsc{Shared} \quad S_x.V(t) = S_t.E}{S \Rightarrow^{rd(t,x)} S}$$

[READ EXCLUSIVE]
$$\frac{S_x.R \neq \textsc{Shared} \quad S_x.W \preceq S_t.V \quad S_x.R \preceq S_t.V}{S \Rightarrow^{rd(t,x)} S[x.R := S_t.E]}$$

[READ SHARED]
$$\frac{S_x.R = \textsc{Shared} \quad S_x.W \preceq S_t.V \quad v = S_x.V[t := S_t.E]}{S \Rightarrow^{rd(t,x)} S[x.V := v]}$$

[READ SHARE]
$$\frac{S_x.R = u@c \quad S_x.W \preceq S_t.V \quad v = \bot_V[t := S_t.E, u := S_x.R]}{S \Rightarrow^{rd(t,x)} S[x.R := \textsc{Shared}, x.V := v]}$$

[WRITE-READ RACE]
$$\frac{S_x.W \not\preceq S_t.V}{S \Rightarrow^{rd(t,x)} \textsc{Error}}$$

[ACQUIRE]
$$\frac{}{S \Rightarrow^{acq(t,m)} S[t.V := (S_t.V \sqcup S_m.V)]}$$

[RELEASE]
$$\frac{S' = S[m.V := S_t.V, \quad t.V := inc_t(S_t.V)]}{S \Rightarrow^{rel(t,m)} S'}$$

[WRITE SAME EPOCH]
$$\frac{S_x.W = S_t.E}{S \Rightarrow^{wr(t,x)} S}$$

[WRITE EXCLUSIVE]
$$\frac{S_x.R \neq \textsc{Shared} \quad S_x.R \preceq S_t.V \quad S_x.W \preceq S_t.V}{S \Rightarrow^{wr(t,x)} S[x.W := S_t.E]}$$

[WRITE SHARED]
$$\frac{S_x.R = \textsc{Shared} \quad S_x.V \sqsubseteq S_t.V \quad S_x.W \preceq S_t.V}{S \Rightarrow^{wr(t,x)} S[x.W := S_t.E]}$$

[WRITE-WRITE RACE]
$$\frac{S_x.W \not\preceq S_t.V}{S \Rightarrow^{wr(t,x)} \textsc{Error}}$$

[READ-WRITE RACE]
$$\frac{S_x.R \neq \textsc{Shared} \quad S_x.R \not\preceq S_t.V}{S \Rightarrow^{wr(t,x)} \textsc{Error}}$$

[SHARED-WRITE RACE]
$$\frac{S_x.R = \textsc{Shared} \quad S_x.V \not\sqsubseteq S_t.V}{S \Rightarrow^{wr(t,x)} \textsc{Error}}$$

[JOIN]
$$\frac{}{S \Rightarrow^{join(t,u)} S[t.V := (u.V \sqcup S_t.V)]}$$

[FORK]
$$\frac{S' = S[u.V := (u.V \sqcup S_t.V)] \quad t.V := inc_t(S_t.V)]}{S \Rightarrow^{fork(t,u)} S'}$$

Figure 1. FASTTRACK2 Algorithm.

ERROR indicate a race has been detected. The analysis stops once this occurs.

Rule [READ SAME EPOCH] optimizes the case where x was already read in this epoch. This rule requires only a single epoch comparison. Rule [READ SHARED SAME EPOCH] similarly optimizes the case where x is read-shared and has already been read in this epoch.

The next three read rules all check for write-read conflicts via the epoch-VC comparison $S_x.W \preceq S_t.V$, and then update R_x appropriately. 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 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), [READ EXCLUSIVE] updates $S_x.R$ with the accessing threads current epoch.

If $S_x.R$ indicates that x is shared, the [READ SHARED] rule 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.

For the more general situation where the current read is concurrent with the previous read epoch, [READ SHARE] allocates a vector clock to record the epochs of *both* reads, since either read could subsequently participate in a read-write race.

The final rule [WRITE-READ RACE] matches the case where the most recent write is concurrent with the current read.

Rule [WRITE SAME EPOCH] optimizes the case where x was already written in this epoch. Rule [WRITE EXCLUSIVE] provides a fast path when $S_x.R$ is an epoch, and this rule checks that the write happens after all previous accesses. In the case where R_x

```
class VectorClock {

  static final empty = new epoch[0];

  // inv: if V != empty, V is unique
  // inv: for all i, get(i) == i@c for some c.
  volatile epoch[] V = empty;

  void ensureCapacity(int n) {
    if (n > this.V.length) {
      epoch[] r = new epoch[n];
      for (int i = 0; i < n; i++) r[i] = get(i);
      this.V = r;
    }
  }

  int size() { return this.V.length; }

  void set(int i, epoch e) {
    ensureCapacity(i+1);
    this.V[i] = e;
  }

  void inc(int i) { set(i, Epoch.tick(get(i)); }

  epoch get(int i) {
    epoch a[] = this.V;
    return (i < a.length) ? a[i] : i@0;
  }

  boolean leq(VectorClock other) {
    int n = max(size(), other.size());
    for (int i = 0; i < n; i++) {
      if (!Epoch.leq(get(i), other.get(i))
        return false;
    }
    return true;
  }

  void join(VectorClock other) {
    for (int i = 0; i < other.size(); i++)
      set(i, Epoch.max(get(i), other.get(i)));
  }

  void copy(VectorClock other) {
    int n = max(size(), other.size());
    for (int i = 0; i < n; i++)
      set(i, other.get(i));
  }
}




class ThreadState extends VectorClock {
  final int tid;
  epoch E;  // inv: E == get(tid)
}

class VarState extends VectorClock {
  volatile epoch R,W;
}

class LockState extends VectorClock { }
```

```
void read(ThreadState st, VarState sx) {
  epoch e = st.E;
  optional {
    epoch r = sx.R;
    if (r == e) return;                    [READ SAME EPOCH]
    if (r == SHARED && sx.get(st.tid) == e)
      return;                              [READ SHARED SAME EPOCH]
  }
  synchronized(sx) {
    epoch w = sx.W;
    assert Epoch.leq(w, st.get(TID(w)); [WRITE-READ RACE]
    epoch r = sx.R;
    if (r != SHARED) {
      if (Epoch.leq(r, st.get(TID(r))) {
        sx.R = e;                          [READ EXCLUSIVE]
      } else {
        sx.set(TID(r), r);
        sx.set(st.tid, e);
        sx.R = SHARED;                     [READ SHARE]
      }
    } else {
      sx.set(st.tid, e);                   [READ SHARED]
    }
  }
}

void write(ThreadState st, VarState sx) {
  epoch e = st.E;
  optional {
    epoch w = sx.W;
    if (w == e) return;                    [WRITE SAME EPOCH]
  }
  synchronized(sx) {
    epoch w = sx.W;
    assert Epoch.leq(w, st.get(TID(w)); [WRITE-WRITE RACE]
    epoch r = sx.R;
    if (r != SHARED) {
      assert Epoch.leq(r, st.get(TID(r));[READ-WRITE RACE]
      sx.W = e;                            [WRITE EXCLUSIVE]
    } else {
      assert sx.leq(t);                    [SHARED-WRITE RACE]
      sx.W = e;                            [WRITE SHARED]
    }
  }
}

void acquire(ThreadState st, LockState sm) {
   st.join(sm); // m held
}
void release(ThreadState st, LockState sm) {
   sm.copy(st); // m held
   st.inc(st.tid);
   st.E = st.get(st.tid);
}

void join(ThreadState st, ThreadState su) {
   st.join(su); // t has joined on u
}
void fork(ThreadState st, ThreadState su) {
   su.join(st); // t will start u
   st.inc(st.tid);
   st.E = st.get(st.tid);
}
```

Figure 2. FASTTRACK2 idealized implementation.

indicates x is shared, [WRITE SHARED] requires a full (slow) VC comparison. The remaining three rules handle write-write and read-write races.

Other Operations. The remaining rules show how FASTTRACK2 handles all other operations (acquire, release, fork, and join). These match the original FASTTRACK definition, except where noted below.

Comparision to the Original* FASTTRACK *Rules.

- [READ SHARED SAME EPOCH]: Our original formulation of this analysis omitted this case because we did not find it demonstrably useful at the time [3]. However, we have now concluded based on more extensive experiments that it occurrs often enough to improve performance in a number of important scenarios.
- [WRITE SHARED]: In the original FASTTRACK2 specification, the rule [WRITE SHARED] changes the read epoch $vx.R$ to be $\bot_e$. That is, we revert from keeping a vector clock of last reads to keeping a single epoch. Future write checks are constant-time as a result. While this may improve performance in some specific cases, we found that not to be the case. Moreover, while we believe implementations using the original rule are correct, eliminating the transition out of the read-shared state simplifies the correctness argument we present below.
- [JOIN]: This rule no longer updates the vector clock for the thread being joined to enable a simpler implementation.
- [. . . RACE]: We now make the four error cases explicit, rather than implicitly defining as the cases not covered by any other rule.

4. Correctness of the FASTTRACK2 Algorithm

FASTTRACK2 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 $\exists S$ such that $S_0 \Rightarrow^{\alpha} S$.*

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

These theorems are stated below. Their proofs follow the same structure as for the original FASTTRACK analysis, except where noted below.

We introduce the following syntactic notation so that we can index epochs in a variable state as if they were vector clocks. The function for $S_x.R$ consults $S_x.V$ when x is in the SHARED state.

$$S_x.R(t) = \begin{cases} t@c & \text{if } S_x.R = t@c \\ t@0 & \text{if } S_x.R = u@c \text{ and } t \neq u \\ S_x.V(t) & \text{if } S_x.R = \text{SHARED} \end{cases}$$

$$S_x.W(t) = \begin{cases} t@c & \text{if } S_x.W = t@c \\ t@0 & \text{if } S_x.W = u@c \text{ and } t \neq u \end{cases}$$

Definition 1 (Active Threads)**.** *Thread t is* active in α *if α does not contain $join(\_, t)$.*

Definition 2 (Well-Formed States)**.** *State S is well-formed for α if for all t, u, m, and x such that t is active in α, and $t \neq u$, the following hold:*

$$\begin{aligned} S_t.E &= S_t.V(t) \\ S_u.V(t) &< S_t.V(t) \\ S_m.V(t) &< S_t.V(t) \\ S_x.R(t) &\leq S_t.V(t) \\ S_x.W(t) &\leq S_t.V(t) \end{aligned}$$

The restriction to active threads t avoids the extra update in the [JOIN] rule mentioned above.

Lemma 1. *S_0 is well-formed for ϵ.*

Lemma 2 (Preservation of Well-Formedness)**.** *If S is well-formed for α and $S \Rightarrow^a S'$ then S' is well-formed for $\alpha.a$.*

For transition $S_a \Rightarrow^a S'_a$, by convention we refer to components of S_a as S^a_t, S^a_u, S^a_v, S^a_m, S^a_x, and compontents of S'_a as $\acute{S}^a_t$, $\acute{S}^a_u$, $\acute{S}^a_v$, $\acute{S}^a_m$, $\acute{S}^a_x$.

Lemma 3 (Clocks Imply Happens-Before)**.** *Suppose S_a is well-formed for some history β and $S_a \Rightarrow^{a.\alpha} S_b \Rightarrow^b S'_b$. Let $t = Tid(a)$ and $u = Tid(b)$. If $S^a_t.V(t) \leq S^b_u.V(t)$ then $a <_{a.\alpha.b} b$.*

Theorem 2 (Soundness)**.** *If $S_0 \Rightarrow^{\alpha} S'$ then α is race-free.*

We introduce the abbreviation:

$$K^a = \begin{cases} \acute{S}^a_t.V & \text{if } a \text{ a } join \text{ or } acquire \text{ operation} \\ S^a_t.V & \text{otherwise} \end{cases}$$

Lemma 4. *Suppose S is well-formed for some history β and $S_a \Rightarrow^{\alpha} S'_a$ and $a, b \in \alpha$. Let $t = Tid(a)$ and $u = Tid(b)$. If $a <_{\alpha} b$ then $K^a(t) \sqsubseteq K^b(u)$.*

Theorem 3 (Completeness)**.** *If α is race-free then $S_0 \Rightarrow^{\alpha} S$.*

5. FASTTRACK2 Idealized Implementation

We now describe an idealized implementation of FASTTRACK2. The code we present is close to the Java implementation available with ROADRUNNER but hides some of the inherent complexities of using that framework.

We begin with a definition of epochs. The actual code encodes epochs as 32-bit or 64-bit integers (depending on configuration flags), but here we use a simple abstraction with the necessary operators defined as follows:

```
typedef epoch = t@c | SHARED;

          TID(t@c) ≡ t
    Epoch.tick(t@c) ≡ t@(c+1)
Epoch.leq(t@c, t@d) ≡ c ≤ d
Epoch.max(t@c, t@d) ≡ t@(max(c,d))
```

Cases not covered by the above rules are undefined.

At run time, ROADRUNNER maintains a mapping from each thread, lock, and variable to a shadow object:

$$\begin{aligned} H: \quad & Tid \rightarrow \texttt{ThreadState} \\ \cup\ & Lock \rightarrow \texttt{LockState} \\ \cup\ & Var \rightarrow \texttt{VarState} \end{aligned}$$

These three classes all extend the `VectorClock` class and are defined in Figure 2. They match the definitions of the analysis state, with the addition of the `tid` field in `ThreadState` to facilitate recovering a thread's identifier from its shadow object.

ROADRUNNER calls an event handling function for each operation performed by the target:

Operation	Event Handler Called	When
$rd(t, x)$	`read(`$H(t)$`,` $H(x)$`)`	before
$wr(t, x)$	`write(`$H(t)$`,` $H(x)$`)`	before
$acq(t, m)$	`acquire(`$H(t)$`,` $H(m)$`)`	after
$rel(t, m)$	`release(`$H(t)$`,` $H(m)$`)`	before
$fork(t, u)$	`fork(`$H(t)$`,` $H(u)$`)`	before
$join(t, u)$	`join(`$H(t)$`,` $H(u)$`)`	after

Each handler runs in the Thread performing the operation. Thus multiple event handlers may run concurrently. The handlers for acquire and join run *after* the Java operations inducing those events.

The rest run *before* the Java operation. These handlers are defined in Figure 2. In each handler, there is exactly one path for each analysis rule. Also, side-effect-free same-epoch tests are labelled as `optional` blocks — they are taken in preference to the other cases in the code, but they can also be elided without changing analysis behavior.

6. Correctness of the FASTTRACK2 Idealized Implementation

In this section, we demonstrate that the FASTTRACK2 event handlers are serializable. A method is *serializable* if for every (arbitrarily interleaved) program execution, there is an equivalent execution with the same overall behavior where the method is executed serially, with no interleaved actions of other threads. This property helps us reason about the correspondence between the formal rules and the code more easily.

6.1 Lipton's Theory Reduction

We employ Lipton's theory of reduction to reason about serializability [10, 6]. Using this theory, we will examine each path through the event handlers, and demonstrate that the steps along that path form an serializable sequence. To do this, we first label each individual step with its commuting behavior, captures as one of four possibilities:

$$Mover \in \{R, L, B, N\}$$

These indicate whether an evaluation step:

- right-commutes with operations of other threads (R);
- left-commutes with operations of other threads (L);
- both right- and left-commutes (B); or
- can be viewed as a single non-mover atomic action (N).

Consider an execution in which an acquire operation on some lock is immediately followed by an action b of a second thread. Since the lock is already held by the first thread, the action b neither acquires nor releases the lock, and hence the acquire operation can be moved to the right of b without changing the resulting state. Thus a lock acquire operation is a right mover.

Similarly, consider an action a of one thread that is immediately followed by a lock release operation by a second thread. During a, the second thread holds the lock, and a can neither acquire nor release the lock. Hence the lock release operation can be moved to the left of a without changing the resulting state, and thus a lock release operations is a left mover.

Finally, consider a read or write to a race-free shared variable. No other thread accesses the field at the same time, and therefore every access to the field is both a right mover and a left mover.

Lipton's theory of reduction shows that any sequence of steps matching the pattern

$$(B|R)^*[N](B|L)^*$$

is serializable and thus amenable to sequential reasoning. Various prior static and dynamic analyses have utilized this theory to ensure serializability of code blocks. (See, *e.g.*, [6, 2] for a more complete discussion of these techniques.)

6.2 Synchronization Discipline

We now discuss the synchronization discipline that describes when event handlers may access various portions of the analysis date and the corresponding mover properties of those accesses, which we summarize in Figure 3.

This synchronization discipline is necessarily complex, as we wish to optimize performance of the event handlers while simultaneously inserting sufficient synchronization for correctness. In particular, to permit the optional fast path code in each handler to execute in a lock free manner, we use a combination of immutable data, thread-local data, read-only data, lock-protected data, write-protected data, and `volatile` data, as well as synchronization disciplines that change over time.

- The location `st.E` is thread local to thread `st`; accesses by `st` are both-movers.
- The location `st.tid` is immutable, so all reads are both-movers.
- The locations `st.V` and `st.V[i]` use a synchronization discipline that changes over time. Initially these are thread local to the unique thread that will fork `st`; they are thread local to `st` once that thread is forked; and after `st` terminates they can be read only by threads that have joined on `st`. The fork-join happens-before orderings thus prevent any race conditions on these locations.
- The locations `sm.V` and `sm.V[i]` can only be accessed by thread holding the lock `sm`. Such accesses are both-movers.
- The location `sx.W` is volatile and write-protected by lock `sx`; that lock must be held for all writes. Hence reading `sx.W` with `sx` held is a both-mover, as there are no concurrent writes; otherwise accesses to `sx.W` are generally not movers.
- The location `sx.R` is write-protected by lock `sx`, and also immutable once it becomes SHARED. Consequently, reads protected by `sx` are both-movers, and reads without the lock are in general non-movers. Note that a read of SHARED is a right-mover, however, since any subsequent operation of another thread cannot modify `sx.R`. Conversely, a read of SHARED is not a left-mover, since a preceding operation of a concurrent thread could have written that value to `sx.R`, and clearly that read and write do not commute.
- The location `sx.V` is volatile and initially protected by lock `sx` for all accesses. Once `sx.R` becomes SHARED, this location is write-protected by `sx`, meaning unprotected reads are now allowed and are non-movers; protected reads are both-movers, and protected writes are non-movers.
- The location `sx.V` is volatile and initially protected by lock `sx` for all accesses. Once `sx.R` becomes SHARED, this location is write-protected by `sx`, meaning unprotected reads are non-movers; protected reads are both-movers, and protected writes are non-movers.
- The location `sx.V[i]` is initially protected by lock `sx` for all accesses. Once `sx.R` becomes SHARED, this location can only be written by thread `i` while it holds the lock `sx`. It can be read either by thread `i` or by any thread holding lock `sx`.

We also show in Figure 4 the moving behavior of each `VectorClock` method called from the event handlers. For all cases except the last two, the method calls are both-movers. That is, all accesses performed by the code in the calling context specified will be both-movers, and the composition of any number of individual both-mover steps is a both-mover [6].

The last two calls in that figure have non-mover behavior because they each include all both-mover accesses except for a single non-mover. For `sx.get(i)`, the single non-mover access is the read of `sx.V`. For `sx.set(i,e)`, the single non-mover access is the write to `sx.V` in the nested call to `ensureCapacity`.

6.3 Path Serializability

We now proceed to show that the event handlers from Figure 2 are serializable. To do this, we enumerate the control flow paths

Location	Mover	When	Property Name
`st.E`	B	Read/Write when current thread is `st`	[1] stE
`st.tid`	B	Read	[2] stTid
`st.V` or `st.V[i]`	B	Read/Write when current thread will fork `st`	[3] stV-Pre
	B	Read/Write when current thread is `st`	[4] stV-Running
	B	Read when current thread has joined `st`	[5] stV-Post
`sm.V` or `sm.V[i]`	N	Read/Write when `sm` is held	[6] smV
`sx.W`	N	Write when `sx` is held	[7] sxW-Write-Locked
	B	Read when `sx` is held	[8] sxW-Read-Locked
	N	Read when `sx` is not held	[9] sxW-Read
`sx.R`	N	Write when `sx.R` $\neq$ `SHARED` and `sx` is held	[10] sxR-Write
	N	Read when `sx.R` $\neq$ `SHARED` and `sx` is not held	[11] sxR-Read
	B	Read when `sx` is held	[12] sxR-Read-Locked
	R	Read when `sx.R` $=$ `SHARED` and `sx` is not held	[13] sxR-Read-Shared
`sx.V`	B	Write when `sx.R` $\neq$ `SHARED` and `sx` is held	[14] sxV-Write
	N	Write when `sx.R` $=$ `SHARED` and `sx` is held	[15] sxV-Write-Shared
	B	Read when `sx` is held	[16] sxV-Read-Locked
	N	Read when `sx` is not held and `sx.R` $=$ `SHARED`	[17] sxV-Read-Shared
`sx.V[i]`	B	Write when `sx` is held and `sx.R` $\neq$ `SHARED`	[18] Elem-Write
	B	Write when `sx` is held and `sx.R` $=$ `SHARED` and the current thread is i	[19] Elem-Write-Shared
	B	Read when `sx` is held	[20] Elem-Read-Locked
	B	Read when `sx.R` $=$ `SHARED` and the current thread is i	[21] Elem-Read-Shared

Figure 3. Mover properties for analysis state memory locations.

Call	Mover	When	Property Name
`st.get(i)`	B	When the current thread is st	[22] Get-Thread
`sx.set(i,e)`	B	When `sx` is held and `sx.R` $\neq$ `SHARED`	[23] Set
`sx.leq(st)`	B	When `sx` is held and the current thread is `st`	[24] Leq
`st.join(sm)`	B	When `sm` is held and the current thread is `st`	[25] Join-Lock
`sm.copy(st)`	B	When `sm` is held and the current thread is `st`	[26] Copy
`st.inc(i)`	B	When the current thread is `st`	[27] Inc
`st.join(su)`	B	When the current thread is `st` and `st` has joined `su`	[28] Join-Current
`su.join(st)`	B	When the current thread is `st` and `st` will fork `su`	[29] Join-Other
`sx.get(i)`	N	When `sx.R` $=$ `SHARED` and the current thread is i	[30] Get-Shared
`sx.set(i,e)`	N	When `sx` is held and tsx.R $=$ SHARED and the current thread is i	[31] Set-Shared

Figure 4. Mover properties for nested method calls locations.

through each event handler and demonstrate each path is serializable. We present several representative paths in Figure 5.

Consider the first two paths in that figure, which correspond to rules [Read Same Epoch] and [Read Shared Same Epoch], respectively. For each path, the first column shows the path rewritten so that each distinct step performs at most one memory or synchronization operation. The second column shows the mover property for each step, with the corresponding justification in the third column. Steps accessing only local memory are both-movers. An inspection of the sequence of mover properties for these two code paths shows that each matches the reducible pattern $(B|R)^*[N](B|L)^*$ and hence is serializable.

The final complexity in our serializability argument is demonstrate by the third example path. It concerns the paths in which the `optional` code blocks terminate normally, without returning from the method. In this situation, the normally-terminating paths are serializable, as demonstrated below using the same strategy as above. Note that once an `optional` and side effect free code block terminates normally, it gives the same observable behavior as if the optional block were simply not executed. A block of code that is not executed is a both mover since it performs no memory accesses or synchronization. Consequently, our proof strategy considers normally terminating optional blocks to be both-movers [7, 15]. This property is critical to the correctness arguments for several of the event handler code paths, including the third path in Figure 5.

Lemma 5. *The optional code block in* `read` *is a both-mover.*

Proof. There are two normally-terminating paths through the optional block. Each is serializable:

Path	Mover	Reason
`epoch r = sx.R;`	N	[12] sxR-Read-Locked
`assume r != e;`	B	
`assume r != SHARED;`	B	
`epoch r = sx.R;`	R	[12] sxR-Read-Locked
`assume r != e;`	B	
`assume r == SHARED`	B	
`int tid == st.tid;`	B	[2] stTid
`epoch v =`	N	[30] Get-Shared
`sx.get(tid);`		
`assume v == e;`	B	

[READ SAME EPOCH]:

Path	Mover	Reason	Simplified
`let e = st.E;`	B	[1] STE	
`let r = sx.R;`	N	[11] SXR-READ	
`assume r == e;`	B		`assume sx.R == st.E;`

[READ SHARED SAME EPOCH]:

Path	Mover	Reason	Simplified
`let e = st.E;`	B	[1] STE	
`let r = sx.R;`	R	[13] SXR-READ-SHARED	
`assume r != e;`	B		
`assume r == SHARED`	B		`assume sx.R == SHARED;`
`let tid == st.tid;`	B	[2] STTID	
`let v = sx.get(tid);`	N	[30] GET-SHARED	
`assume v == e;`	B		`assume sx.get(st.tid) == st.E;`

[READ SHARE]:

Path	Mover	Reason	Simplified
`let e = st.E;`	B	[1] STE	
`optional { ... }`	B	Lemma 5, below	
`lock(sx);`	R	Section 6.1	`lock(sx);`
`let w = sx.W;`	B	[8] SXW-READ-LOCKED	
`let vw = st.get(TID(w));`	B	[22] GET-THREAD	
`assume Epoch.leq(w, vw);`	B		`assume Epoch.leq(sx.W, st.get(TID(sx.W)));`
`let r = sx.R;`	B	[12] SXR-READ-LOCKED	
`assume r != SHARED;`	B		`assume sx.R != SHARED;`
`let vr = st.get(TID(r));`	B	[22] GET-THREAD	
`assume !Epoch.leq(r, vr);`	B		`assume !Epoch.leq(sx.R, st.get(TID(sx.R)));`
`sx.set(TID(r), r)`	B	[23] SET	`sx.set(TID(sx.R), sx.R)`
`let tid = st.tid;`	B	[2] STTID	
`sx.set(tid, e);`	B	[23] SET	`sx.set(st.tid, e);`
`sx.R = SHARED`	N	[10] SXR-WRITE	
`unlock(sx);`	L	Section 6.1	`unlock(sx);`

Figure 5. Representative event handler traces.

In addition, the each path is side-effect free. Thus, we may consider the block a both-mover [7, 15]. □

Lemma 6. *The optional code block in* `write` *is a both-mover.*

Proof. There is only one normally-terminating path through the optional block. It is serializable:

Path	Mover	Reason
`epoch w = sx.W;`	N	[9] SXW-READ
`assume w != e;`	B	

Thus, the pure block is a both-mover, as in the previous lemma. □

All other paths through the event handlers are treated similarly. We show the complete set of paths together with the corresponding reduction arguments in the Appendix.

7. Correspondence To Analysis Rules

Once we have shown the event handlers are serializable, we use sequential reasoning to simplify each code path into the steps shown in the fourth column of Figure 5. By inspection, each simplified code path has a fairly direct correspondence to the antecedents and state updates of the corresponding analysis rule from Figure 1. This correspondence can be formalized by developing a bisimulation between the state transition system described by Figure 1 and a transition system capturing the behavior of the serialized event handlers. This approach has been explored for a more basic race detection algorithm [11] and is readily adapted to this context.

8. FASTTRACK2 Implementation for Java

We have implemented FASTTRACK2 the ROADRUNNER analysis framework [4]. ROADRUNNER takes as input a compiled Java target program and inserts instrumentation code into the target to generate the event stream of memory and synchronization operations. This section outlines a number of FASTTRACK2 implementation details not present in our idealized implementation.

Additional Synchronization Primitives. FASTTRACK2 supports other forms of synchronization, including volatiles variables, wait-/notify, and barriers, as in the original [3]. FASTTRACK2 also ensures the appropriate static initializers happen before any use of a static variable or class.

Program	2.7GHz 12-core (hyper-threaded) Intel Xeon				2.4GHz 16-core AMD Opteron			
	Base Time (sec)	Overhead			Base Time (sec)	Overhead		
		FT	FT2	FT2L		FT	FT2	FT2L
crypt	0.22	74.09	71.51	81.21	0.41	93.68	79.69	80.72
lufact	0.55	61.69	63.31	64.59	0.68	68.95	68.92	74.89
moldyn	1.80	33.48	34.02	35.58	5.31	25.97	26.04	26.73
montecarlo	0.97	10.11	9.55	9.30	2.16	10.88	8.27	8.10
raytracer	0.91	22.87	21.95	25.10	2.03	19.16	19.20	19.42
series	58.15	0.05	0.05	0.05	111.32	0.01	0.01	0.01
sor	0.27	23.11	24.35	23.80	0.74	15.33	14.98	14.01
sparsematmult	0.68	34.69	34.52	36.35	1.28	27.88	29.23	28.07
avrora	3.73	2.00	1.93	1.95	7.69	1.38	1.40	1.39
batik	1.02	2.88	2.91	2.90	1.35	3.98	3.81	3.95
fop	0.30	7.93	7.74	7.77	0.43	7.75	7.97	8.14
h2	4.25	8.62	8.47	9.09	10.70	7.23	7.36	7.34
jython	1.62	17.77	16.59	17.39	2.72	16.89	16.24	16.60
luindex	0.43	13.10	13.36	13.30	0.58	16.19	15.56	16.58
lusearch	0.32	17.49	17.08	17.00	0.64	19.60	19.53	19.20
pmd	0.69	3.32	3.12	3.20	1.12	2.56	2.50	2.53
sunflow	0.69	29.80	28.97	28.48	1.39	34.14	26.70	27.11
tomcat	0.45	2.20	1.48	1.50	0.85	1.94	1.72	1.62
xalan	0.39	6.17	5.90	6.33	1.14	4.20	3.49	3.73
Geomean		9.40	9.07	9.33		8.21	7.79	7.78

Table 1. Checker Overhead for FASTTRACK (FT), FASTTRACK2 (FT2), and FASTTRACK2L (FT2L).

State Representation and Fast Paths. ROADRUNNER's programming model is optimized for performance in various ways and necessitates maintaining H for threads, locks, and variables via different mechanisms than the simplified model presented here. ROADRUNNER also enables tools to provide fast path code that is typically inlined into the target at each memory access. FASTTRACK2's fast paths are essentially the same as the read and write event handlers, except that they fail over to the slow path on a detected race rather than raising an error. This is to enable better error reporting, and also because static initializer order constraints are not considered on the fast path for performance reasons.

`VectorClock` ***Methods.*** The vector clock methods in our idealized implementation are designed for simplicity. In the actual implementation, we unroll loops, inline nested method calls, and perform various other local optimizations to improve performance.

Optimized `Epoch.leq` ***Tests.*** Our implementation also optimizes accesses to vector clocks by short-circuiting tests guaranteed to succeed via program order. For example, in the `write` handler we rewrite the test `Epoch.leq(w, st.get(TID(w))` as

```
st.E == TID(w) || Epoch.leq(w, st.get(TID(w))
```

If the thread for last write epoch `w` is the current thread, the previous write happens before the current operation via program order, and we can thus avoid accessing the vector clock.

Errors. As with our previous implementations, FASTTRACK2 is guaranteed to precisely report the first data race encountered. It continues to process events after that point and attempts to produce meaningful errors messages for subsequent data races on a best-effort basis.

9. FASTTRACK2 Performance

We report FASTTRACK2's performance on the JavaGrande [8] and DaCapo [1] benchmark suites. We configured the JavaGrande programs to use their largest data sizes and 16 worker threads, and we configured the DaCapo benchmarks to use their default sizes. Tradebeans and eclipse are incompatible with our ROADRUNNER framework and are omitted.

The DaCapo test harness uses a number of class loading features not supported in ROADRUNNER. Thus, we extracted the benchmarks from the DaCapo harness and ran them (and also the JavaGrande programs) in a simplified harness integrated into ROADRUNNER. That harness follows the same model of running the target's workload several times in a warm up phase and then measuring the running time for repeated iterations of the workload. We used 20 iterations for measurement. This methodology departs from our earlier validation experiments (*e.g.* [3]) in which we measured the time to perform only a single iteration of the workload at a time. We opted for a methodology similar to that of DaCapo's to better isolate the "steady-state" performance of the race detection algorithms from the overhead of JVM startup, class loading, bytecode instrumentation, etc.

Table 1 shows the results for two experimental platforms: 1) a 2.7GHz 12-core (hyper-threaded) Intel Xeon processor with 64GB of memory running OSX and Java 1.7; and 2) a 2.4GHz 16 core AMD Opteron processor with 64GB of memory running Ubuntu Linux and Java 1.7. That table presents the base running time for each program, as well as the overhead of using FASTTRACK, FASTTRACK2, FASTTRACK2L (a variant that encodes epochs as `long`s rather than `int`s to accommodate larger clock values). Overhead is the additional time required to check a program:

$$\frac{\text{CheckerTime} - \text{BaseTime}}{\text{BaseTime}}$$

On both platforms, FASTTRACK2's overhead is about 4-5% lower than FASTTRACK overall, and FASTTRACK2L's overhead is also slightly lower by about 1% and 5% lower than FASTTRACK on the two platforms, respectively. These benchmarks and test harness are available as part of the ROADRUNNER 0.5 release.

References

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

[2] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *POPL*, pages 256–267, 2004.

[3] C. Flanagan and S. N. Freund. FastTrack: Efficient and precise dynamic race detection. In *PLDI*, pages 121–133, 2009.

[4] C. Flanagan and S. N. Freund. The RoadRunner dynamic analysis framework for concurrent programs. In *PASTE*, pages 1–8, 2010.

[5] C. Flanagan and S. N. Freund. RedCard: Redundant check elimination for dynamic race detectors. In *ECOOP*, pages 255–280, 2013.

[6] C. Flanagan, S. N. Freund, M. Lifshin, and S. Qadeer. Types for atomicity: Static checking and inference for Java. *ACM Trans. Program. Lang. Syst.*, 30(4), 2008.

[7] C. Flanagan, S. N. Freund, and S. Qadeer. Exploiting purity for atomicity. *IEEE Trans. Software Eng.*, 31(4):275–291, 2005.

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

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

[10] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

[11] W. Mansky, Y. Peng, S. Zdancewic, and J. Devietti. Verifying dynamic race detection. In *Conference on Certified Programs and Proofs*, pages 151–163, 2017.

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

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

[14] 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.

[15] L. Wang and S. D. Stoller. Static analysis of atomicity for programs with non-blocking synchronization. In *PPOPP*, pages 61–71, 2005.

[16] J. R. Wilcox, P. Finch, C. Flanagan, and S. N. Freund. Array shadow state compression for precise dynamic race detection. In *ASE*, pages 155–165, 2015.

A. Serializability of All Paths

A.1 Reads

[READ SAME EPOCH]:

Path	Mover	Reason	Simplified
`epoch e = st.E;`	B	[1] STE	
`epoch r = sx.R;`	N	[11] SXR-READ	
`assume r == e;`	B		`assume sx.R == st.E;`

[READ SHARED SAME EPOCH]:

Path	Mover	Reason	Simplified
`epoch e = st.E;`	B	[1] STE	
`epoch r = sx.R;`	R	[13] SXR-READ-SHARED	
`assume r != e;`	B		
`assume r == SHARED`	B		`assume sx.R == SHARED;`
`int tid == st.tid;`	B	[2] STTID	
`epoch v = sx.get(tid);`	N	[30] GET-SHARED	
`assume v == e;`	B		`assume sx.get(st.tid) == st.E;`

[WRITE-READ RACE]:

Path	Mover	Reason	Simplified
`epoch e = st.E;`	B	[1] STE	
`optional { ... }`	B	Lemma 5, below	
`lock(sx);`	R	Section 6.1	`lock(sx);`
`epoch w = sx.W;`	B	[8] SXW-READ-LOCKED	
`epoch vw = st.get(TID(w));`	B	[22] GET-THREAD	
`assume !Epoch.leq(w, vw);`	B		`assume !Epoch.leq(sx.W, st.get(TID(sx.W)));`
`error;`	N		`error;`

[READ SHARED]:

Path	Mover	Reason	Simplified
`epoch e = st.E;`	B	[1] STE	
`optional { ... }`	B	Lemma 5, below	
`lock(sx);`	R	Section 6.1	`lock(sx);`
`epoch w = sx.W;`	B	[8] SXW-READ-LOCKED	
`epoch vw = st.get(TID(w));`	B	[22] GET-THREAD	
`assume Epoch.leq(w, vw);`	B		`assume Epoch.leq(sx.W, st.get(TID(w)));`
`epoch r = sx.R;`	B	[12] SXR-READ-LOCKED	
`assume r == SHARED;`	B		`assume sx.R == SHARED;`
`int tid = st.tid;`	B	[2] STTID	
`sx.set(tid, e);`	N	[31] SET-SHARED	`sx.set(st.tid, st.E);`
`unlock(sx);`	L	Section 6.1	`unlock(sx);`

[READ EXCLUSIVE]:

Path	Mover	Reason	Simplified
`epoch e = st.E;`	B	[1] STE	
`optional { ... }`	B	Lemma 5, below	
`lock(sx);`	R	Section 6.1	`lock(sx);`
`epoch w = sx.W;`	B	[8] SXW-READ-LOCKED	
`epoch vw = st.get(TID(w));`	B	[22] GET-THREAD	
`assume Epoch.leq(w, vw);`	B		`assume Epoch.leq(sx.W, st.get(TID(sx.W)));`
`epoch r = sx.R;`	B	[12] SXR-READ-LOCKED	
`assume r != SHARED;`	B		`assume sx.R != SHARED;`
`epoch vr = st.get(TID(r));`	B	[22] GET-THREAD	
`assume Epoch.leq(r, vr);`	B		`assume Epoch.leq(sx.R, st.get(TID(sx.R)));`
`sx.R = e;`	N	[10] SXR-WRITE	`sx.R = e;`
`unlock(sx);`	L	Section 6.1	`unlock(sx);`

[READ SHARE]:

Path	Mover	Reason	Simplified
`epoch e = st.E;`	B	[1] STE	
`optional { ... }`	B	Lemma 5, below	
`lock(sx);`	R	Section 6.1	`lock(sx);`
`epoch w = sx.W;`	B	[8] SXW-READ-LOCKED	
`epoch vw = st.get(TID(w));`	B	[22] GET-THREAD	
`assume Epoch.leq(w, vw);`	B		`assume Epoch.leq(sx.W, st.get(TID(sx.W)));`
`epoch r = sx.R;`	B	[11] SXR-READ	
`assume r != SHARED;`	B		`assume sx.R != SHARED;`
`epoch vr = st.get(TID(r));`	B	[22] GET-THREAD	
`assume !Epoch.leq(r, vr);`	B		`assume !Epoch.leq(sx.R, st.get(TID(sx.R)));`
`sx.set(TID(r), r)`	B	[23] SET	`sx.set(TID(sx.R), sx.R)`
`int tid = st.tid;`	B	[2] STTID	
`sx.set(tid, e);`	B	[23] SET	`sx.set(st.tid, e);`
`sx.R = SHARED`	N	[10] SXR-WRITE	
`unlock(sx);`	L	Section 6.1	`unlock(sx);`

A.2 Writes

[WRITE SAME EPOCH]:

Path	Mover	Reason	Simplified
`epoch e = st.E;`	B	[1] STE	
`epoch w = sx.W;`	N	[9] SXW-READ	
`assume w == e;`	B		`assume sx.W == st.E;`

[WRITE-WRITE RACE]:

Path	Mover	Reason	Simplified
`epoch e = st.E;`	B	[1] STE	
`optional { ... }`	B	Lemma 6, below	
`lock(sx);`	R	Section 6.1	`lock(sx);`
`epoch w = sx.W;`	B	[8] SXW-READ-LOCKED	
`epoch vw = st.get(TID(w));`	B	[22] GET-THREAD	
`assume !Epoch.leq(w, vw);`	B		`assume !Epoch.leq(sx.W, st.get(TID(sx.W)));`
`error;`	N		`error;`

[READ-WRITE RACE]:

Path	Mover	Reason	Simplified
`epoch e = st.E;`	B	[1] STE	
`optional { ... }`	B	Lemma 6, below	
`lock(sx);`	R	Section 6.1	`lock(sx);`
`epoch w = sx.W;`	B	[8] SXW-READ-LOCKED	
`epoch vw = st.get(TID(w));`	B	[22] GET-THREAD	
`assume Epoch.leq(w, vw);`	B		`assume Epoch.leq(sx.W, st.get(TID(w)));`
`epoch r = sx.R;`	B	[12] SXR-READ-LOCKED	
`assume r != SHARED;`	B		`assume sx.R != SHARED;`
`epoch vr = st.get(TID(r));`	B	[22] GET-THREAD	
`assume !Epoch.leq(r, vr);`	B		`assume !Epoch.leq(sx.R, st.get(TID(sx.R)));`
`error;`	N		`error;`

[WRITE EXCLUSIVE]:

Path	Mover	Reason	Simplified
`epoch e = st.E;`	B	[1] STE	
`optional { ... }`	B	Lemma 6, below	
`lock(sx);`	R	Section 6.1	`lock(sx);`
`epoch w = sx.W;`	B	[8] SXW-READ-LOCKED	
`epoch vw = st.get(TID(w));`	B	[22] GET-THREAD	
`assume Epoch.leq(w, vw);`	B		`assume Epoch.leq(sx.W, st.get(TID(w)));`
`epoch r = sx.R;`	B	[12] SXR-READ-LOCKED	
`assume r != SHARED;`	B		`assume sx.R != SHARED;`
`epoch vr = st.get(TID(r));`	B	[22] GET-THREAD	
`assume Epoch.leq(r, vr);`	B		`assume Epoch.leq(sx.R, st.get(TID(sx.R)));`
`sx.W = e;`	N	[7] SXW-WRITE-LOCKED	`s.W = e;`
`unlock(sx);`	L	Section 6.1	`unlock(sx);`

[SHARED-WRITE RACE]:

Path	Mover	Reason	Simplified
`epoch e = st.E;`	B	[1] STE	
`optional { ... }`	B	Lemma 6, below	
`lock(sx);`	R	Section 6.1	`lock(sx);`
`epoch w = sx.W;`	B	[8] SXW-READ-LOCKED	
`epoch vw = st.get(TID(w));`	B	[22] GET-THREAD	
`assume Epoch.leq(w, vw);`	B		`assume Epoch.leq(sx.W, st.get(TID(w)));`
`epoch r = sx.R;`	B	[12] SXR-READ-LOCKED	
`assume r == SHARED;`	B		`assume sx.R != SHARED;`
`assume !sx.leq(st);`	B	[24] LEQ	`assume !sx.leq(t);`
`error`	N		`error;`

[WRITE SHARED]:

Path	Mover	Reason	Simplified
`epoch e = st.E;`	B	[1] stE	
`optional { ... }`	B	Lemma 6, below	
`lock(sx);`	R	Section 6.1	`lock(sx);`
`epoch w = sx.W;`	B	[8] sxW-READ-LOCKED	
`epoch vw =`	B	[22] GET-THREAD	
`st.get(TID(w));`			
`assume Epoch.leq(w, vw);`	B		`assume Epoch.leq(sx.W, st.get(TID(w)));`
`epoch r = sx.R;`	B	[12] sxR-READ-LOCKED	
`assume r == SHARED;`	B		`assume sx.R == SHARED;`
`assume sx.leq(st);`	B	[24] LEQ	`assume sx.leq(t);`
`sx.W = e;`	N	[7] sxW-WRITE-LOCKED	`s.W = e;`
`unlock(sx);`	L	Section 6.1	`unlock(sx);`

A.3 Other

[ACQUIRE]:

Path	Mover	Reason	Simplified
`st.join(sm);`	B	[25] JOIN-LOCK	`st.join(sm);`

[RELEASE]:

Path	Mover	Reason	Simplified
`sm.copy(st);`	B	[26] COPY	`sm.copy(st);`
`int tid = st.tid;`	B	[2] stTID	
`st.inc(tid);`	B	[27] INC	`st.inc(st.tid);`
`val e = st.get(tid);`	B	[22] GET-THREAD	
`st.E = e;`	B	[1] stE	`st.E = st.get(st.tid);`

[JOIN]:

Path	Mover	Reason	Simplified
`st.join(su);`	B	[28] JOIN-CURRENT	`st.join(su);`

[FORK]:

Path	Mover	Reason	Simplified
`su.join(st);`	B	[29] JOIN-OTHER	`su.join(st);`
`int tid = st.tid;`	B	[2] stTID	
`st.inc(tid);`	B	[27] INC	`st.inc(st.tid);`
`val e = st.get(tid);`	B	[22] GET-THREAD	
`st.E = e;`	B	[1] stE	`st.E = st.get(st.tid);`