VerifiedFT: A Verified, High-Performance Precise Dynamic Race Detector

James R. Wilcox
University of Washington

Cormac Flanagan
University of California, Santa Cruz

Stephen N. Freund
Williams College

Abstract

Dynamic data race detectors are valuable tools for testing and validating concurrent software, but to achieve good performance they are typically implemented using sophisticated concurrent algorithms. Thus, they are ironically prone to the exact same kind of concurrency bugs they are designed to detect. To address these problems, we have developed VerifiedFT, a clean slate redesign of the FastTrack race detector [19]. The VerifiedFT analysis provides the same precision guarantee as FastTrack, but is simpler to implement correctly and efficiently, enabling us to mechanically verify an implementation of its core algorithm using CIVL [27]. Moreover, VerifiedFT provides these correctness guarantees without sacrificing any performance over current state-of-the-art (but complex and unverified) FastTrack implementations for Java.

CCS Concepts • **Software and its engineering → Synchronization**; **Concurrent programming languages**; **Software defect analysis**; **Formal software verification**;

Keywords Data races, concurrency, dynamic analysis

ACM Reference Format:
James R. Wilcox, Cormac Flanagan, and Stephen N. Freund. 2018. VerifiedFT: A Verified, High-Performance Precise Dynamic Race Detector. In *Proceedings of PPoPP '18: Principles and Practice of Parallel Programming, Vienna, Austria, February 24–28, 2018 (PPoPP '18),* 14 pages.
https://doi.org/10.1145/3178487.3178514

1 Introduction

In order to fully exploit the performance available on modern hardware, programmers must write concurrent software. But designing fast concurrent algorithms requires avoiding the dual pitfalls of (1) over-synchronization, which leads to poor performance, and (2) under-synchronization, which leads to subtle correctness bugs that are difficult to detect, reproduce, and eliminate. Consequently, validating a concurrent algorithm requires both empirical performance measurement as well as rigorous testing and formal, even mechanically-verified, correctness arguments.

Given the difficulty of these tasks, a wide variety of tools exist to facilitate building correct concurrent code. In this paper, we focus on dynamic race detectors. A dynamic race detector typically instruments the target program to execute an *event handler* for each relevant action of the target program, such as a heap memory access or synchronization operation. These event handlers are called on *every* such program action. One common strategy to mitigate the resulting analysis overhead is to execute each event handler in the same thread that generated the event, as in [19, 49, 51] and others. Thus, event handlers may execute concurrently, making dynamic race detectors *themselves* concurrent algorithms, subject to the same performance and correctness challenges mentioned above.

To address those challenges, we present VerifiedFT, a clean-slate redesign of the FastTrack race detector [19, 20]. Like FastTrack, VerifiedFT is designed to be precise: it reports an error *if and only if* a race occurs. Moreover, VerifiedFT achieves state-of-the-art performance and while also supporting a mechanically-verified correctness proof for an idealized implementation of its core algorithm.

Earlier papers [19, 20] presented an idealized sequential implementation of the FastTrack analysis specification that was not mechanically verified. To improve performance, our FastTrack implementations for Java (available on `github` [45]) extended that idealized implementation with complex synchronization, including write-protected data, optimistic concurrency with retries, benign (but subtle) data race conditions, and intricate data invariants. This complexity led to implementations that we found difficult to build, debug, and maintain. Indeed, several committed changes (*e.g.* [46, 47]) address problems where our analysis code handled rare but possible situations incorrectly, leading to the irony of concurrency bugs in a concurrency bug detector.

Such bugs could in theory lead to missed races or false positives. Missing races compromises the checker's primary objective of providing a strong race-free guarantee. False positives are time-consuming and challenging to recognize because they often require a deep understanding of both the

PPoPP '18, February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-4982-6/18/02…$15.00
https://doi.org/10.1145/3178487.3178514

target program and the race detector itself. Mechanically verifying the core VerifiedFT implementation enables us to avoid these scenarios.

Our approach for developing VerifiedFT involves the co-design of three components: a high-level specification; an optimized idealized implementation using a sophisticated synchronization discipline; and a formal argument showing that the implementation satisfies the specification. The process required carefully considering both the performance and proof-burden tradeoffs of each design decision.

At the specification level, we modified the original FastTrack analysis rules in several ways. One of the key changes eliminates an optimization that added significant implementation complexity but ultimately offered no clear performance benefit. Another introduces specialized, lower-overhead handling of a common memory access pattern.

At the implementation level, our primary objective was to develop lock-free code for the three most common analysis rules, which handle 85% of all accesses. In order to permit these cases to be lock-free, the synchronization discipline is necessarily sophisticated, utilizing a mix of immutable data, thread-local data, read-only data, lock-protected data, write-protected data, and `volatile` data, as well as synchronization mechanisms that change over time. Despite having a sophisticated synchronization discipline, the code itself is straightforward.

We prove the idealized implementation correct by showing that each event handler is serializable and then using sequential reasoning to show each handler satisfies the analysis specification. Our proof of serializability is based on an extended form of Lipton's theory of reduction [24, 25, 33, 57]. We have mechanically verified serializability and functional correctness for our idealized implementation using the CIVL verification tool [27].

Complementary prior work used Coq [37] to mechanically verify that the original FastTrack specification is precise and that an idealized implementation of a basic vector-clock dynamic race detector employing simple lock-based synchronization is correct with respect to its specification [34]. We go beyond that work to mechanically verify the correctness of an idealized implementation containing both 1) FastTrack's more advanced dynamic analysis and 2) a sophisticated synchronization discipline enabling lock-free fast paths for common cases. Both are essential for reducing checking overhead.

We have implemented VerifiedFT for Java in the RoadRunner framework [21, 45]. Our implementation is faster than our earlier lock-based FastTrack implementation and comparable in speed to our earlier CAS-based variant. It is also compatible and complementary with techniques to reduce FastTrack checking overhead by, for example, compressing analysis state or optimizing where checks occur [22, 29, 44, 53, 58]. VerifiedFT can be readily used in those systems to provide strong correctness guarantees without impacting performance.

In summary, this paper contributes the following:

- The VerifiedFT high-level specification that, while only a modest extension of the original FastTrack specification, serves as a better foundation for a fast and correct implementation (Section 3).
- The VerifiedFT concurrent algorithm, presented as basic and optimized idealized implementations (Sections 4 and 5) and embodied in our Java implementation (Section 7).
- A mechanically-verified proof that each concurrent event handler satisfies its specification (Section 6).
- Performance results showing that the VerifiedFT implementation for Java is as fast as or faster than existing state-of-the-art FastTrack implementations (Section 8).

2 Preliminaries

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 $t \in Tid = \{A, B, \ldots\}$, and a *trace* α captures an execution of a program by listing the sequence of operations performed by the various threads.

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

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

- $rd(t,x)$ and $wr(t,x)$ read and write a variable 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.

We restrict our attention to *feasible* traces that 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 $join(t',u)$.

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:

[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.

Program order: The two operations are performed by the same thread.

Locking: The two operations acquire/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.

3 VerifiedFT Analysis

We now introduce our VerifiedFT analysis for traces and its formal specification. We begin with an overview of the information VerifiedFT tracks at run time and how VerifiedFT detects races. We then describe VerifiedFT's high-level specification as a state transition system. VerifiedFT modifies and extends the original FastTrack analysis in several ways, and we conclude this section with a discussion of how VerifiedFT's analysis differs from FastTrack's.

Epochs and Vector Clocks. At any point during execution, each thread t has an associated *clock* $c \in Nat$ that is incremented at each release (or fork) operation. We refer to a pair of a thread and clock as an *epoch*, denoted $t@c$. Epochs support the following operations (which are undefined if they mix epochs from different threads) and minimal element.

$$\begin{aligned} t@c_1 \le t@c_2 \quad &\text{iff} \quad c_1 \le c_2 \\ max(t@c_1, t@c_2) \quad &= \quad t@max(c_1, c_2) \\ t@c + 1 \quad &= \quad t@(c+1) \\ tid(t@c) \quad &= \quad t \\ \bot_e \quad &= \quad A@0 \end{aligned}$$

Note that the minimal epoch is not unique; another is $B@0$.

A vector clock $V \in VC = Tid \to Epoch$ records an epoch for each thread. We maintain the invariant that all vector clocks are well-formed in that for all t, $tid(V(t)) = t$. Vector clocks are partially-ordered ($\sqsubseteq$) in a point-wise manner, with a join operation ($\sqcup$) and minimal element ($\bot_V$). The helper function inc_t increments the t-component of a vector clock:

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

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

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

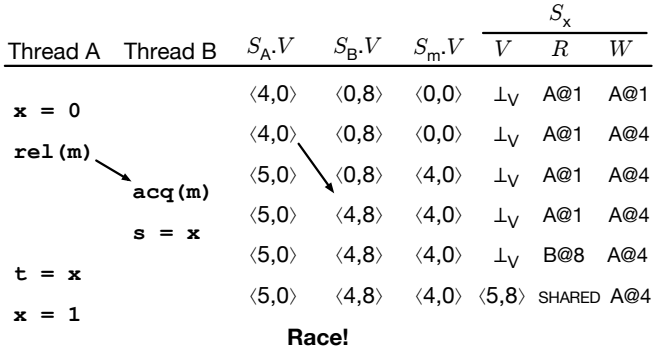

Figure 1. Example VerifiedFT analysis state.

Analysis State. VerifiedFT maintains an analysis state S that records history information for each thread t, lock m, and variable x. As in FastTrack, that state includes a vector clock $S_t.V$ for each thread t such that, for any other thread u, the clock entry $S_t.V(u)$ records the epoch for the last operation of u that happens before the current operation of thread t. Additionally, $S_t.V(t)$ records the current epoch of thread t. The algorithm also maintains a vector clock $S_m.V$ for each lock m. That vector clock records the time of the last release of the lock.

Figure 1 illustrates how these components are updated while analyzing part of a trace. We use $\langle m, n \rangle$ to abbreviate vector clock $\langle A@m, B@n \rangle$. When thread A releases lock $\mathtt{m}$, $S_m.V$ becomes $S_A.V = \langle 4,\ 0 \rangle$, and $S_A.V$ is incremented to $\langle 5,\ 0 \rangle$ to reflect that later steps of A happen after that release. Similarly, when thread B acquires $\mathtt{m}$, $S_B.V$ is joined via $\sqcup$ with $S_m.V$, reflecting that later steps of B happen after the release by A.

To identify when an access to a variable x races with a previous write to x, VerifiedFT records in $S_x.W$ the epoch of the last write to each variable x. If the trace so far is race-free, then all observed writes are totally ordered by the happens-before relation, and this information about the last write is sufficient to detect if any previous write races with a subsequent access to x. Specifically, if thread t accesses x, that access does not race with any previous writes if and only if $S_x.W \preceq S_t.V$. For example, consider the read of $\mathtt{x}$ by thread B in Figure 1. Immediately before the read, $S_x.W = A@4$ and $S_B.V = \langle 4, 8 \rangle$. Since $A@4 \preceq \langle 4, 8 \rangle$, this access is race-free.

To identify when a write to a variable x races with a previous read of x, VerifiedFT records the read history of x using FastTrack's adaptive representation. If all reads of x have been totally ordered, then we only record in $S_x.R$ the epoch of the last read of x. Alternatively, if there have been concurrent reads of x, then $S_x.R$ is a special flag Shared, and $S_x.V$ is a clock vector in which $S_x.V(t)$ is the epoch of the last read of x by thread t. Switching to the vector clock is necessary because a subsequent write could race with *any* of those unordered reads.

In Figure 1, after thread B reads x, $S_x.R$ is set to $B@8$ to reflect when that read occurred. The subsequent read by A is a race-free concurrent read, leading $S_x.R$ to become SHARED and $S_x.V$ to become $\langle 5, 8\rangle$, the vector clock recording the two concurrent reads. When thread A writes to x in the final step, $S_x.V = \langle 5, 8\rangle \not\sqsubseteq \langle 5, 0\rangle = S_A.V$, indicating a race.

In summary, the VERIFIEDFT analysis state records a *ThreadState*, *LockState*, or *VarState* for each thread, lock, and variable, respectively:

$$\begin{aligned} S \in \mathit{State} :\ & \mathit{Tid} \to \mathit{ThreadState} \\ & \cup\ \mathit{Lock} \to \mathit{LockState} \\ & \cup\ \mathit{Var} \to \mathit{VarState} \end{aligned}$$

$$\begin{aligned} \mathit{ThreadState} &= \{\ V{:}\,\mathit{VC}\ \} \\ \mathit{LockState} &= \{\ V{:}\,\mathit{VC}\ \} \\ \mathit{VarState} &= \{\ V{:}\,\mathit{VC}, R{:}(\mathit{Epoch} \cup \{\textsc{Shared}\}), W{:}\,\mathit{Epoch}\ \} \end{aligned}$$

We use S_t, S_m, and S_x, to refer to the records for thread t, lock m, and variable x. The initial analysis state is:

$$\begin{aligned} S_0 = \lambda t.\{V : \mathit{inc}_t(\bot_V)\} \\ \cup\ \lambda m.\{V : \bot_V\} \\ \cup\ \lambda x.\{V : \bot_V, R : \bot_e, W : \bot_e\} \end{aligned}$$

Analysis Rules. Whenever a target program performs an operation a, VERIFIEDFT updates its analysis state via the transition relation $S \Rightarrow^a S'$ shown in Figure 3. The relation $S \Rightarrow^a$ ERROR indicates a race has been detected, and the analysis stops once this occurs.

The first six rules analyze a read operation $rd(t, x)$ performed by the target program. Rule [READ SAME EPOCH] optimizes the case where x was already read in the current epoch, *i.e.* since the last synchronization operation. In this situation, checks on the earlier access are sufficient to ensure the read is race-free [19]. The term E_t abbreviates $S_t.V(t)$, the current epoch for thread t. Rule [READ SHARED SAME EPOCH] similarly optimizes the case where x is read-shared and previously read in this epoch.

The next three read rules all check for write-read conflicts via the epoch-VC comparison $S_x.W \preceq S_t.V$, but differ in how they update the read history in $S_x.R$ and $S_x.V$. If $S_x.R = \textsc{Shared}$, then [READ SHARED] simply updates $S_x.V(t)$ appropriately. (Here, $S[x.V := v]$ denotes the state that is identical to S except that $S_x.V$ is now updated to v.)

If $S_x.R \neq \textsc{Shared}$ and the current read happens after the previous read (*i.e.*, $S_x.R \preceq S_t.V$), then [READ EXCLUSIVE] updates $S_x.R$ to be the thread's current epoch. Alternatively, if the current read is concurrent with the previous read, then [READ SHARE] uses a vector clock to record the epochs of *both* reads, since either read could subsequently participate in a read-write race. The rule [WRITE-READ RACE] detects when a read races with the most recent write.

The next six rules analyze a write operation $wr(t, x)$ performed by the target program. Rule [WRITE SAME EPOCH] optimizes the case where x was already written in this epoch,

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

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

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

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

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

[READ SHARE]
$$\frac{S_x.R = u@c \qquad S_x.W \preceq S_t.V \qquad u \neq t \qquad v = \bot_V[t := E_t, u := S_x.R] \qquad S' = S[x.R := \textsc{Shared}, x.V := v]}{S \Rightarrow^{rd(t,x)} S'}$$

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

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

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

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

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

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

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

[ACQUIRE]
$$\frac{S' = S[t.V := (S_t.V \sqcup S_m.V)]}{S \Rightarrow^{acq(t,m)} S'}$$

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

[JOIN]
$$\frac{S' = S[t.V := (S_u.V \sqcup S_t.V)]}{S \Rightarrow^{join(t,u)} S'}$$

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

Figure 2. VERIFIEDFT Specification.

as above for reads. Rule [Write Exclusive] provides a fast path when $S_x.R$ is an epoch; this rule checks that the write happens after all previous accesses.

In the case where $S_x.R$ = Shared, [Write Shared] requires a full (slow) VC comparison. The remaining three write rules handle write-write and read-write races.

VerifiedFT handles acquire, release, fork, and join operations as in FastTrack. When thread t acquires lock m, the [acquire] rule joins t's vector clock $S_t.V$ with $S_m.V$, the time of m's last release, to record that subsequent operations by thread t happen after that release. Correspondingly, when thread t releases lock m, rule [release] records thread t's current vector clock in $S_m.V$, and begins a new epoch for thread t by incrementing its clock. When thread t joins on thread u, rule [join] sets the current time for t to be the join of the vector clocks for t and u. When thread t forks a thread u, rule [fork] sets the forked thread's time to be after the time of the fork operation and moves the forking thread into a new epoch by incrementing its clock.

Comparison to the FastTrack Specification. We made three changes to the FastTrack analysis rules [19], both to improve performance and to make them simpler to implement and reason about.

- We added the [Read Shared Same Epoch] rule, which is not present in FastTrack. It substantially improves performance in some programs, because it avoids unnecessary checks when accessing a read-shared variable multiple times in the same epoch.
- In the case where $S_x.R$ = Shared, the FastTrack [Write Shared] rule changes $S_x.R$ back to $\bot_e$, essentially forgetting all reads before the write. This enables subsequent write checks to be constant-time, since a subsequent race with any of those reads would also race with the write. We found that this optimization led to no improvement in practice, and actually degraded performance for some usage patterns by causing $S_x.R$ to "thrash" between the shared and unshared states. Consequently, the VerifiedFT [Write Shared] rule does not change $S_x.R$. This change also simplifies the correctness argument we present below.
- The FastTrack [Join] rule increments the vector clock entry $S_u.V[u]$ of the thread being joined. That update is unnecessary and eliminating it in VerifiedFT simplifies the synchronization discipline (but does add some minor complexity to the proof of Theorem 3.1 below).

Correctness of the VerifiedFT Analysis. Like FastTrack, the VerifiedFT analysis rules are precise and report an error if and only if the observed trace contains a data race, as characterized by the following theorem.

Theorem 3.1 (Correctness)**.** *Suppose α is a feasible trace. Then α is race-free if and only if $S_0 \Rightarrow^{\alpha} S$ for some S.*

The proof primarily follows the same structure as the original FastTrack proof [19].

4 Idealized Implementation, Version 1

Analysis State and Event Handlers. Figure 3 presents VerifiedFT-v1, a basic concurrent implementation of the analysis rules. The analysis state is represented as the `ThreadState`, `LockState`, and `VarState` objects, as defined on lines 1–7 . That figure also includes a `VectorClock` class, as well as an `epoch` data type and associated operations. (Our actual implementation bit-packs epochs in 32-bit integers, but here we use the datatype to simplify our exposition.) These classes match the analysis state S defined in Section 3, with the additional `t` field in `ThreadState` to facilitate recovering a thread's identifier (encoded as an integer) from its corresponding `ThreadState` object.

Each `VectorClock` contains an internal array `V` of epochs indexed by thread identifiers. Our implementation preserves the invariant that each entry `V[t]` is a valid epoch for thread t. The `VectorClock` class contains `set` and `get` methods to access the elements in `V`. Setting the epoch for a thread t that is outside the current bounds of `V` necessitates allocating a larger array via `ensureCapacity`. Getting the epoch for a thread t outside the current bounds of `V` simply results in the minimal epoch `t@0` for t. `VectorClocks` also support `inc`, `leq`, and `join` methods implemented according to the definitions in Section 3, as well as a `copy` method.

We assume the underlying run-time system maintains a one-to-one mapping between threads, locks, and variables in the target program and corresponding `ThreadState`, `LockState`, and `VarState` objects, and that each handler runs in the thread performing the operation. Thus multiple event handlers may run concurrently. These assumptions match features provided by the RoadRunner analysis framework for Java [21], which we use for our prototypes, and they enable us to model and reason about analyses at the level of our trace language, independent of the target language or platform. The handlers for acquire and join run *after* the target operations inducing those events. The rest run *before* the corresponding target operation.

The right column of Figure 3 presents code to handle the six operations in our trace language. Each handler manipulates the analysis state in the exact same way as the formal analysis rules in Section 3. In `read` and `write`, there is a clear correspondence between execution paths through the methods and the analysis rules, which we highlight. by labelling the end of each path with the rule that it implements. The code for the [. . . Same Epoch] cases are first because they are the most common. The remaining cases are ordered to minimize the number of memory accesses and computation steps required.

Synchronization Discipline. VerifiedFT-v1 uses mutex locks to protect all mutable shared data in the analysis state. In more detail, given the `VarState sx` for a variable `x` in the target program, the `LockState sm` for lock `m`, and the

```
class ThreadState extends VectorClock {
 final int t;
}
class LockState extends VectorClock { }
class VarState extends VectorClock {
 epoch R,W;
}

datatype epoch = t@c | SHARED;

     TID(t@c) ≡ t
LEQ(t@c, t@d) ≡ c ≤ d
MAX(t@c, t@d) ≡ t@(max(c,d))
     INC(t@c) ≡ t@(c+1)


class VectorClock {
 // invariant: V is unique pointer
 // invariant: for all t, TID(get(t)) == t
 epoch[] V = new epoch[0];

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

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

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

 void inc(int t) { set(t, INC(get(t)); }

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

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

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

```
void read(ThreadState st, VarState sx) {
 int t = st.t;
 epoch e = st.get(t);
 synchronized(sx) {
  epoch r = sx.R;
  if (r == e) return;                    [Read Same Epoch]
  if (r == SHARED && sx.get(t) == e)
   return;                               [Read Shared Same Epoch]
  epoch w = sx.W;
  assert LEQ(w, st.get(TID(w)));         [Write-Read Race]
  if (r != SHARED) {
   if (LEQ(r, st.get(TID(r))) {
    sx.R = e;                            [Read Exclusive]
   } else {
    sx.set(TID(r), r);
    sx.set(t, e);
    sx.R = SHARED;                       [Read Share]
   }
  } else {
   sx.set(t, e);                         [Read Shared]
  }
 } // release
}

void write(ThreadState st, VarState sx) {
 int t = st.t;
 epoch e = st.get(t);
 synchronized(sx) {
  epoch w = sx.W;
  if (w == e) return;                    [Write Same Epoch]
  assert LEQ(w, st.get(TID(w)));         [Write-Write Race]
  epoch r = sx.R;
  if (r != SHARED) {
   assert LEQ(r, st.get(TID(r)));        [Read-Write Race]
   sx.W = e;                             [Write Exclusive]
  } else {
   assert sx.leq(st);                    [Shared-Write Race]
   sx.W = e;                             [Write Shared]
  }
 } // release
}

void acquire(ThreadState st, LockState sm) {
 st.join(sm); // m is held
}

void release(ThreadState st, LockState sm) {
 sm.copy(st); // m is held
 st.inc(st.t);
}

void join(ThreadState st, ThreadState su) {
 st.join(su); // has joined on u
}

void fork(ThreadState st, ThreadState su) {
 su.join(st); // t will start u
 st.inc(st.t);
}
```

Figure 3. VerifiedFT-v1 Idealized Implementation.

`ThreadState st` for thread `t`, the synchronization discipline is as follows.

`sx.W`, `sx.R`, `sx.V`, `sx.V[*]`: Protected by the lock `sx`.[2]

`sm.V`, `sm.V[*]`: Protected by the lock `m` from the target.

`st.t`: Read-only.

`st.V`, `st.V[*]`: Initially thread-local to the unique thread that will fork thread `t`. Once thread `t` has been forked, these locations are thread-local to `t`. After `t` terminates, they are read-only and accessible by any thread that has joined on `t`. The fork-join happens-before orderings prevent race conditions resulting from these phase changes.

If followed, this discipline ensures race freedom, since at most one thread can ever access any mutable analysis state.

Serializability. The first step in showing that VerifiedFT-v1 correctly implements the analysis rules is to prove that each handler is serializable. A method is *serializable* [23, 33] if, for every (arbitrarily interleaved) program execution, there is an equivalent execution with the same overall behavior in which the method executes serially, with no interleaved actions of other threads. To show serializability, we employ Lipton's theory of reduction [23, 33]. We label each memory and synchronization operation in each handler with its commuting behavior (**R**, **L**, **B**, or **N**) to indicate whether it:

- **R**: right-commutes with operations of other threads;
- **L**: left-commutes with operations of other threads;
- **B**: both right- and left-commutes; or
- **N**: is a single non-mover atomic action.

We illustrate these annotations with the `write` handler in Figure 3. Consider the acquire of `sx` on line 87 of `write`. Suppose this acquire is immediately followed in a trace by an operation b of a second thread. Since `sx` is already held by the first thread, the action b neither acquires nor releases that lock, and hence the acquire operation can be moved after (to the right of) b without changing the resulting state. Thus a lock acquire operation is a right-mover (**R**).

Similarly, suppose the lock release on line 99 of `write` is preceded by an operation b by a second thread. Operation b can neither acquire nor release the lock since it is held by the first thread. Hence the lock release operation can be moved to the left of b without changing the resulting state, and thus a lock release operations is a left-mover (**L**).

Next, consider the read of `sx.W` on line 88. Since that field is race-free, there are no concurrent conflicting accesses, and this access is thus a both-mover (**B**). Indeed, all data accesses in `write` are race-free and both-movers.

Finally, consider the call to `sx.get(t)` on line 86. Since the two reads of `st.V` and `st.V[t]` inside the callee are both-movers (**B**), we also label the call itself as a both-mover (**B**), and similarly for the other calls. The annotations for the entire `write` method are thus:

```
void write(ThreadState st, VarState sx) {
 B int t = st.t;
 B epoch e = st.get(t);
 R synchronized(sx) {
 B   epoch w = sx.W;
 B   //...remainder of write code...
 L }
}
```

According to Lipton's theory, any sequence of steps matching the pattern (**B**|**R**)*[**N**](**B**|**L**)* is serializable. Since all execution paths through `write` match this pattern, `write` is serializable. The `read` method is similarly serializable.

The `acquire` and `release` event handlers follow the prescribed synchronization discipline since they are always called with the lock `m` held, and all accesses they perform are both-movers (**B**). Thus, they too are serializable.

The `fork(st,su)` handler is run by a thread `t` before it forks thread `u` corresponding to the state object `su`. According to the synchronization discipline, it has exclusive access to `su.V` and `su.V[*]`, meaning that all memory accesses in `fork` are again both-movers (**B**). That is also the case for `join(st,su)` since the thread `t` running the `join` handler has previously joined on thread `u`. Given that `su.V` and `su.V[*]` are now read-only, `join` is also serializable.

Functional Correctness. After proving that the six event handlers are serializable, sequential reasoning suffices to prove that each code path accesses and modifies the analysis state in exactly the same way as the specification provided by the analysis rules.

Comparison to Prior FastTrack Implementations. VerifiedFT-v1 is correct but slow. A Java implementation of VerifiedFT-v1 exhibits an overhead of about 15x in our experiments (Section 8). Two key contributors to this overhead are acquiring/releasing the lock `sx` on every read or write to `x` and lock contention on `sx` when there are concurrent reads of `x`. The latter in effect serializes otherwise-concurrent accesses to read-shared variables in the target program.

Our two prior FastTrack implementations for Java, FT-Mutex and FT-CAS attempt to mitigate the overhead via complex synchronization mechanisms for `VarState` objects. FT-Mutex uses the lock `sx` to write-protect the fields of `sx`. That is, writes to those fields are protected by the lock, but reads may not be. To ensure serializability, the handlers use an optimistic control mechanism that detects whether any value read from memory has been modified prior to updating the analysis state. The handler retries in the case that there is interference. This approach enables the [Read Same Epoch] and [Write Same Epoch] cases to be lock free (since no writes occur) but introduces a number of subtle ordering issues related to accessing the vector clock component of `sx`.

[2] When defining the synchronization discipline, a term of the form `sx.V[i]` describes the single memory operation of reading the value at index `i` out of an array reference previously read from `sx.V`.

```
127 void read(ThreadState st, VarState sx) {
128 B int t = st.t;
129 B epoch e = st.get(t);
130    | pure {
131    |  N/R epoch r = sx.R;
132 B  |       if (r == e) return;   [Read Same Epoch]
133    |  N    if (r == SHARED && sx.get(t) == e)
134    |        return;              [Read Shared Same Epoch]
135    | }
136 R synchronized(sx) {
137 B  epoch w = sx.W;
138 B  assert LEQ(w, st.get(TID(w)));[Write-Read Race]
139 B  epoch r = sx.R;
140    if (r != SHARED) {
141 B   if (LEQ(r, st.get(TID(r))) {
142 N    sx.R = e;                  [Read Exclusive]
143     } else {
144 B    sx.set(TID(r), r);
145 B    sx.set(t, e);
146 N    sx.R = SHARED;             [Read Share]
147     }
148    } else {
149 B   sx.set(st.t, e);           [Read Shared]
150    }
151 L }
152 }
153
154 void write(ThreadState st, VarState sx) {
155 B int t = st.t;
156 B epoch e = st.get(t);
157    | pure {
158    |  N epoch w = sx.W;
159 B  |    if (w == e) return;     [Write Same Epoch]
160    | }
161 R synchronized(sx) {
162 B  epoch w = sx.W;
163 B  assert LEQ(w, st.get(TID(w)));[Write-Write Race]
164 B  epoch r = sx.R;
165    if (r != SHARED) {
166 B   assert LEQ(r, st.get(TID(r)));[Read-Write Race]
167 N   sx.W = e;                   [Write Exclusive]
168    } else {
169 B   assert sx.leq(st);         [Shared-Write Race]
170 N   sx.W = e;                   [Write Shared]
171    }
172 L }
173 }
```

Figure 4. VerifiedFT-v2 Idealized Implementation.

The FT-CAS variant embeds `sx.W` and `sx.R` in a single 8-byte long that is always read and written atomically and uses a similar optimistic mechanism based on atomic CAS operations. The lock `sx` is still used for the vector clock, and this implementation suffers from some of the same complexities as FT-Mutex.

5 Idealized Implementation, Version 2

We now present, VB, an optimized variant of our idealized implementation. In VerifiedFT-v2, we eliminate the synchronization overheads mentioned above by removing locking from the code for the three most common analysis rules: [Read Same Epoch], [Write Same Epoch], and [Read-Shared Same Epoch], which cover 60%, 14%, and 12% of all accesses in our benchmarks, respectively. While effective, these optimizations necessitate a more complex synchronization discipline than VerifiedFT-v1. However, the approach is still significantly simpler than the earlier FastTrack implementations.

Optimized* `write` *Handler. We improve the VerifiedFT-v1 `write` handler by initially reading `sx.W` without holding `sx`:

```
void write(ThreadState st, VarState sx) {
 B int t = st.t;
 B epoch e = st.get(t);
 N epoch w = sx.W;
   if (w == e) return;          [Write Same Epoch]
 R synchronized(sx) {
 B   epoch w = sx.W;
     //... remainder of write from V1...
 L }
}
```

If the initial read of `sx.W` yields the current epoch `e`, the code simply returns as prescribed by the [Write Same Epoch] rule. Otherwise, the handler acquires the lock `sx`, re-reads `sx.W` in case it has changed, and proceeds as before. To avoid stale reads under relaxed memory models, we declare `VarState`'s `W` field to be `volatile` [35]. We also modify the synchronization discipline for `sx.W` as follows:

`sx.W`: Write-protected by lock `sx`.

This *write-protected* synchronization discipline means that the lock `sx` must be held for all writes to `sx.W`, but not necessarily for reads. Consequently, lock-protected writes to `sx.W` are non-movers (**N**), as there may be concurrent unprotected reads, which would also be non-movers (**N**). However, lock-protected reads of `sx.W` are both-movers (**B**) since holding the lock `sx` precludes any concurrent writes.

Given this discipline, the new code block no longer matches the reducible pattern (**B**|**R**)*[**N**](**B**|**L**)* because the initial read of `sx.W` (**N**) precedes the lock acquire (**R**), as indicated above.

To verify that `write` is still serializable, we move the [Write Same Epoch] code into a *pure block* [24, 25, 57], as shown in Figure 4 (lines 157–160). Pure blocks do not change behavior but simply indicate that:

1. every normally terminating execution of the pure block (*i.e.*, one that does not `return`) does not change the program state; and
2. the execution of the pure block is optional (because the handler still behaves correctly via the slower synchronized code even if the pure block is skipped.)

The revised `write` handler can be proven serializable. In particular, a skipped execution of the pure block is clearly a both-mover (**B**) since it does not interact with concurrent threads. Further, since any normally terminating execution of the pure block does not change program state, that block is observationally equivalent to a skipped pure block, and so also is a both-mover (**B**). Thus, our proof strategy considers normally terminating `pure` blocks to be both-movers [24, 25, 57], enabling us to verify that `write` is serializable.

Optimized* `read` *Handler. We optimize the `read` handler by moving the [Read Same Epoch] case outside the critical section on `sx`, and also the [ReadShared Same Epoch] case to avoid serializing accesses to read-shared data. The new `read` handler is shown in Figure 4 (lines 130–135).

When called by thread `t`, the handler now performs an unsynchronized read of `sx.R`. If that yields the value Shared, it also performs unsynchronized reads of `sx.V` and `sx.V[t]` in the nested call to `sx.get(t)`. It is important to note that the array is only accessed at index `t` in this case. In addition, the `read` handler also performs synchronized reads and writes of all these locations and may call `sx.set()` which then calls `sx.ensureCapacity()`, which may perform synchronized reads of all array entries `sx.V[*]` and a write to `sx.V` while resizing the vector clock. We adopt the following synchronization discipline, which is permissive enough to permit the above access patterns while still being strict enough to ensure serializability since most accesses remain race-free both-movers (**B**).

`sx.R`: Initially write-protected by lock `sx`, and immutable if it becomes Shared. Consequently, reading `sx.R` with `sx` held is a both-mover (**B**), as there are no concurrent writes. Reading Shared from `sx.R` (possibly without the lock) is a right-mover (**R**), as there are no subsequent writes. Other accesses to `sx.R` are non-movers (**N**).

`sx.V`: Protected by lock `sx` when `sx.R` ≠ Shared, and write-protected by `sx` when `sx.R` = Shared. Thus, when not shared, all accesses are lock-protected both-movers (**B**). Once shared, unprotected reads are non-movers (**N**), protected reads are both-movers (**B**), and protected writes are non-movers (**N**), *e.g.* lines 142 and 146.

`sx.V[t]`: Protected by lock `sx` when `sx.R` ≠ Shared. If `sx.R` = Shared, then `sx.V[t]` can be read by any thread hold lock `sx` (to permit vector clock resizing) *or* by thread `t` without hold the lock (to support the lock-free fast path). Conversely, `sx.V[t]` can be written only by thread `t` *and* only when hold lock `sx`. Under this discipline, all accesses are race free and thus both-movers (**B**).

We also declare `ThreadState.R` and `VectorClock.V` to be `volatile` to avoid stale reads.

In the optimized handler in Figure 4 that follows this discipline, if the pure block reads the current epoch from `sx.R`, that read is a non-mover (**N**), and the handler returns via the [Read Same Epoch] fast path, which is serializable. If instead `sx.R` contains Shared, that read is a right-mover (**R**) and the subsequent reads of `sx.V` and `sx.V[t]` inside `sx.get(t)` are a non-mover (**N**) and a both-mover (**B**), respectively, and the sequence **RNB** is serializable. If neither fast path applies, the analysis state is unchanged, and we may consider the pure block to have been skipped, and hence a both-mover (**B**). The rest of the `read` handler code is verified in the same manner as before. The synchronization discipline ensures all remaining accesses are both-movers (**B**), except for at most one non-mover (**N**) write to `sx.R` on each code path, and thus these paths are all reducible.

6 Verifying VerifiedFT in CIVL

We have translated the VerifiedFT-v2 idealized implementation and the analysis rule specification from Figure 3 into the input language for the CIVL concurrent program verification tool [12, 27, 28] and mechanically verified that each event handler is serializable and functionally correct with respect to its specification.

Following the CIVL methodology, we express VerifiedFT-v2 as a stack of layers, with each layer only calling functions from lower layers. Each CIVL function definition includes a commuting annotation (*e.g.*: `atomic`, `both`, etc.) as well as a functional specification expressed as an atomic action.

Layer 0, the lowest layer of abstraction, contains primitive memory and synchronization operations. For example, the functions to acquire/release the lock `sx` are written in **Layer 0** as follows:[3]

```
void AcquireVarLock(linear st: ThreadState, sx: VarState)
  right [ assume sx.holder == nil; sx.holder = st; ];

void ReleaseVarLock(linear st: ThreadState, sx: VarState)
  left [ assert sx.holder == st; sx.holder = nil; ];
```

Like all **Layer 0** functions, the specification in square brackets also is the implementation, and CIVL verifies that this implementation satisfies the declared commutativity.

We implement the lock on `sx` as an extra field `sx.holder`, which is set to either the `ThreadState` object for the thread holding the lock, or `nil` if it is not held. The specifications indicate that the lock can only be acquired if it is not currently held, and that only the holding thread can release it.

The `linear` annotation on the `st` parameter indicates that concurrent calls to these functions will be passed different `st` objects since each `ThreadState` object is local to a single thread of the target program.

In CIVL, we formalize the write-protected synchronization discipline for `sx.W` via the following three **Layer 0** functions, which express, for example, that reading `sx.W` is atomic (**N**) if the lock is not held, and a both-mover if the lock is held.

```
epoch VarStateGetWNoLock(sx: VarState)
  atomic [ return sx.W; ];
```

[3] Since CIVL's syntax can be cumbersome, we present our code fragments in a more readily-understood pseudo-code.

```
epoch VarStateGetW(linear st: ThreadState, sx: VarState)
  both [ assert sx.holder == st; return sx.W; ];

void VarStateSetW(linear st: ThreadState, sx: VarState,
                  e: epoch)
  atomic [ assert sx.holder == st; sx.W = e; ];
```

Additional **Layer 0** functions include accessors for other analysis state components, **Layer 1** contains vector clock operations, and **Layer 2** contains the six event handlers.

Figure 5 contains an excerpt from the **Layer 2** `write` handler. The specification encodes a nondeterministic choice among goto labels formalizing the analysis rules, and the body uses the **Layer 0** functions defined above, plus the both-mover function `ThreadStateGetE` to read the executing thread's current epoch. The handler returns false if it detects a data race.

The body of `write` contains explicit `yield` points to indicate where interference may be visible, including at start of the code and immediately before each exit. To ensure that `write` is atomic and correct, CIVL partitions each path through the body into fragments separated by yields. It then verifies that one of those fragments conforms to the method's atomic specification and that the rest have no effect on visible state. Thus, while CIVL does not directly support `pure` blocks, it does provide a means to verify the code for `write` is correct. In particular, we insert a `yield` point after the pure block. For paths extending past that point, CIVL verifies that the fragment before the `yield` has no effect and that the fragment after the `yield` properly implements the given atomic specification.

While not shown due to space limitations, we also introduce invariants and function pre-/post conditions necessary to carry out the verification at each layer.[4] They capture, for example, that vector clocks store appropriate epochs at each index, that a `VarState` object that has entered Shared mode remains in Shared mode, and that the event handler calls respect the conditions for feasible traces in Section 2.

We encode this feasible trace restriction by introducing additional global variables to capture which threads are currently running, which have terminated, which have joined on each thread, etc. Explicit conditions on `yield` instructions restrict how other threads may alter visible state at yield points.

To gain confidence that we properly encoded the feasible trace restriction, we extend our CIVL implementation with an additional non-deterministic test driver at **Layer 3** that models every thread in the system performing every possible sequence of operations compatible with the trace invariants.

Overall, our experience with CIVL was quite positive. The iterative abstraction refinement methodology it supports was an excellent match for the approach we took in our development. **Layers 0–2** contains roughly 900 lines of CIVL code, and **Layer 3** contains an additional 250.

[4]Full details of these specifications can be found in our implementation, which is available on line [13].

```
boolean write(linear st: ThreadState, sx: VarState)
  atomic [
    goto WriteFastPath, WriteExclusive, WriteWriteRace, ...;
    WriteFastPath:
      assume sx.W == st.V[t];
      return true;
    WriteExclusive:
      assume LEQ(sx.W, st.V[TID(sx.W)]);
      assume sx.R != SHARED && LEQ(sx.R, st.V[TID(sx.R)]);
      sx.W = st.V[t];
      return true;
    WriteWriteRace:
      assume !LEQ(sx.W, st.V[TID(sx.W)]);
      return false;
    ...
  ]
{
  yield;
  epoch e = ThreadStateGetE(st);
  // pure {
    epoch w = VarStateGetWNoLock(sx);
    if (w == e) { yield; return true; } // [Write Same Epoch]
  // }
  yield;
  AcquireVarLock(st, sx);
  w = VarStateGetW(st, sx);
  ...
  VarStateSetW(st, sx, e);
  ReleaseVarLock(st, sx);
  yield;
  return true;
}
```

Figure 5. CIVL pseudo-code for `write`.

7 VerifiedFT Implementation for Java

We have implemented VerifiedFT v1 and v2 in the RoadRunner analysis framework for Java [21]. RoadRunner takes as input a compiled target program and inserts instrumentation code to generate an event stream of memory and synchronization operations that is then processed by the analysis. While the analysis rules and idealized implementations stop at the first error, our Java implementations continue checking until the target finishes. This section outlines a number of implementation details not present in our idealized implementations.

Additional Synchronization Primitives. Our implementation supports other forms of synchronization, including volatile variables, wait/notify, and barriers, as in the standard FastTrack implementations [19, 45]. Our implementation also captures the happens-before orderings between the static initializers and uses of a static variables or classes.

State Representation and Fast Paths. RoadRunner's programming model is optimized for performance in various ways and necessitates maintaining objects for thread, lock,

and variable state via different mechanisms than the simplified model presented here. RoadRunner also enables tools to provide specialized `read`/`write` fast path handlers that are typically inlined into the target at each memory access. VerifiedFT's fast paths are essentially the same as the presented event handlers, except that they fail over to the slow path handler on a detected race rather than raising an error. This enables better error reporting because more diagnostic information is available on the slow path and also because static initializer order constraints are not considered on the fast path for performance reasons.

Local Optimizations. A thread reads its current epoch frequently. Thus we cache it in `ThreadState` to avoid extracting it from `ThreadState`'s vector clock. Our vector clock implementation optimizes Figure 3 by unrolling loops and inlining nested method calls. We also optimize tests guaranteed to succeed via program order. For example, in the `write` handler we rewrite `LEQ(w, st.get(TID(w))` as

```
st.t == TID(w) || LEQ(w, st.get(TID(w))
```

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

8 Experimental Validation

We report the performance of our implementation on the JavaGrande [32] and DaCapo [5] 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. Dacapo's tradebeans and eclipse programs 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 10 iterations for measurement, and repeated each experiment 10 times. Our test platform was a 2.4GHz 16 core AMD Opteron processor with 64GB of memory running Ubuntu Linux and Java 1.8.

Table 1 summarizes the base running time for each program, as well as the VerifiedFT overhead. (The VerifiedFT-v1.5 variant optimizes only the [Write Same Epoch] and [Read Same Epoch] cases to highlight the necessity of optimizing the [ReadShared Same Epoch] case.) We also report overheads for the FT-Mutex and FT-CAS implementations distributed with RoadRunner 0.4 [45]. These are described in Section 4. Overhead is the additional time required to check a program:

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

Program	Base Time (sec)	Overhead (x Base Time)				
		FastTrack		VerifiedFT		
		Mutex	CAS	v1	v1.5	v2
crypt	0.4	112.6	90.97	165.3	109.63	92.14
lufact	0.69	69.68	55.7	117.78	115.08	71.23
moldyn	4.87	29.11	27.77	47.45	32.4	25.26
montecarlo	2.24	8.7	9.84	13.34	7.27	7.32
raytracer	1.85	19.15	19.31	82.15	19.31	13.3
series	119.14	0.01	0.01	0.01	0.01	0.01
sor	0.74	15.29	11.59	19.04	15.86	15.84
sparse	1.28	36.15	26.81	316.85	246.02	25.5
avrora	6.18	1.6	1.37	3.81	1.61	1.56
batik	1.27	3.77	3.83	4.2	3.91	3.89
fop	0.3	10.33	9.96	11.39	10.13	10.19
h2	9.62	7.63	7.21	10.92	8.21	7.92
jython	5.37	8.68	8.29	9.02	8.47	8.5
luindex	0.54	17.01	13.84	33.91	16.53	16.49
lusearch	0.64	19.81	19.5	24.27	20.11	19.89
pmd	0.89	3.44	3.22	5.58	3.41	3.32
sunflow	1.47	30.95	29.52	158.82	152.67	25.38
tomcat	0.68	2.71	2.51	2.26	2.32	2.36
xalan	0.47	12.21	11.2	13.06	11.04	10.88
Geo Mean		8.87	8.11	15.0	10.8	8.12

Table 1. Overhead for FastTrack and VerifiedFT.

VerifiedFT-v2's sophisticated synchronization discipline is critical for performance. VerifiedFT-v1 lacks these optimizations and has an overhead of 15.0x. Eliminating locking for the [Read Same Epoch] and [Write Same Epoch] cases reduced VerifiedFT-v1.5's overhead to 10.8x. Eliminating locking from the [ReadShared Same Epoch] case further reduces the overhead to 8.12x. This final optimization is particularly beneficial for benchmarks like `sunflow` and `sparse` that heavily use read-shared data. VerifiedFT-v2 is as fast or faster than both previous implementations FT-Mutex (8.87x) and FT-CAS (8.11x). (We also note that modifying FT-Mutex and FT-CAS to use the revised VerifiedFT analysis rules does not meaningfully improve their performance.)

In summary, these results show that VerifiedFT provides simplicity and correctness without sacrificing any performance in comparison to previous complex, error-prone, and hard-to-maintain FastTrack implementations.

The performance of all of these race detectors can be further improved via the static and dynamic optimization techniques found in systems like BigFoot [44], which lowers checking overhead to roughly 2.5x when built on top of either the earlier FastTrack implementations or VerifiedFT-v2.

9 Related Work

Mechanically-Verified Concurrency Analyses. While there has been much work verifying concurrent programs in general, relatively little has looked at verified *analyses* of concurrent programs.

In the most closely related work, Mansky et al. used the Coq proof assistant [37] to formalize data races and prove the soundness and completeness of the specification rules for several high-level race detection analyses, including FastTrack [34]. This corresponds to mechanically proving Theorem 3.1 from Section 3. They present two proofs. One is a direct proof following a proof outlined in our earlier work [19], which we also follow to prove Theorem 3.1. The second is based on a bisimulation with a basic vector-clock-based analysis.

Mansky et al. also prove that an idealized implementation of the basic vector-clock-based analysis is correct with respect to its specification. In contrast to VerifiedFT-v2, their implementation (1) does not use FastTrack's adaptive epoch representation and (2) encloses all analysis code for reads and writes within mutex-protected critical sections (as in VerifiedFT-v1). As such, a complete implementation based on that approach is likely to incur overhead higher than VerifiedFT-v1.

One strength of their approach is that they show that the analysis code, which is inserted into the target during an instrumentation pass, does not interfere with the target code. While valuable to show, proving this non-interference property introduces significant complexity to their proof, with handling instrumentation accounting for roughly 19,000 out of 21,000 lines of Coq definitions and proofs [34].

To avoid this complexity, our proof instead uses an event handler model in which the target program cannot modify the shadow state and the event handlers cannot modify the program state. We believe non-interference could be shown via a technique similar to theirs.

As another example of verified concurrent program analysis, Sadowski et al. described a partial verification of some properties of the Velodrome dynamic atomicity checker [48]. They reasoned at the specification level using trace-based semantics, and thus did not verify implementations.

Efficient Race Detection. Much prior work has focused on static [1–3, 8, 15, 17, 26, 38, 56] and dynamic [4, 11, 14, 16, 30, 39, 41–43, 49–52, 55, 60] data race detection.

FastTrack introduced *epochs* to reduce the overhead of precise dynamic tools using vector clocks to represent the happens-before relation [14, 36, 42]. Another well-studied approach to reduce overhead is using a single shadow location for whole arrays and objects [7, 10, 19, 40, 42, 55], although this may generate false alarms, motivating additional technology to distinguish real races [9, 17].

Other work has focused on shadow state compression [53, 58] and check buffering or redundancy elimination techniques [29, 51]. In other recent work, we explored static shadow compression in conjunction with analyses to reduce the number of checks performed by a race detector without impacting precision [22, 44]. A number of such approaches are based on extensions to the FastTrack analysis. VerifiedFT complements that work by providing a high-performance implementation with strong correctness guarantees upon which these other systems can be built. Another complementary approach for reducing overhead is sampling [6, 16, 18], again with some loss of soundness.

Eraser verifies race freedom only for data that is thread-local, read-shared, or lock protected [49], and has been extended to produce fewer false alarms [9, 17, 40, 59]. Many other imprecise analyses, some of which have been widely used, have been developed as well. Checkers in this category include ThreadSanitizer [51, 52, 54], Intel Inspector [30, 31], and Archer [4], which utilizes ThreadSanitizer in conjunction with static and dynamic analyses specially designed for OpenMP programs.

10 Summary

VerifiedFT provides a simpler and mechanically-verified race detection algorithm with performance comparable to existing highly-tuned but complex and hard-to-maintain alternatives. Our work leverages several key program verification techniques, many of which are embodied in the CIVL verifier, and it may serve as a starting point for further work on developing, specifying, and verifying the correctness of high-performance concurrent algorithms.

VerifiedFT for Java is available as part of the RoadRunner Analysis Framework distribution, version 0.5.

Acknowledgments

We thank Shaz Qadeer for his assistance with CIVL, and the anonymous reviewers and our shepherd Murali Krishna Ramanathan for their feedback and assistance. This work was supported, in part, by NSF Grants 1337278, 1421051, 1421016, and 1439042.

References

[1] Martín Abadi, Cormac Flanagan, and Stephen N. Freund. 2006. Types for Safe Locking: Static Race Detection for Java. *Transactions on Programming Languages and Systems* 28, 2 (2006), 207–255.

[2] Rahul Agarwal and Scott D. Stoller. 2004. Type Inference for Parameterized Race-Free Java. In *VMCAI*. 149–160.

[3] Alexander Aiken and David Gay. 1998. Barrier Inference. In *POPL*. 243–354.

[4] Simone Atzeni, Ganesh Gopalakrishnan, Zvonimir Rakamaric, Dong H. Ahn, Ignacio Laguna, Martin Schulz, Gregory L. Lee, Joachim Protze, and Matthias S. Müller. 2016. ARCHER: Effectively Spotting Data Races in Large OpenMP Applications. In *IPDPS*. 53–62.

[5] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg,

Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*. 169–190.

[6] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. 2010. PACER: proportional detection of data races. In *PLDI*. 255–268.

[7] Michael D. Bond, Milind Kulkarni, Man Cao, Minjia Zhang, Meisam Fathi Salmi, Swarnendu Biswas, Aritra Sengupta, and Jipeng Huang. 2013. OCTET: capturing and controlling cross-thread dependences efficiently. In *OOPSLA*. 693–712.

[8] Chandrasekhar Boyapati and Martin Rinard. 2001. A parameterized type system for race-free Java programs. In *OOPSLA*. 56–69.

[9] Cardelli, L. 1984. A Semantics of Multiple Inheritance. In *Semantics of Data Types (Lecture Notes in Computer Science 173)*. Springer Verlag, Berlin.

[10] Chiyan Chen and Hongwei Xi. 2005. Combining programming with theorem proving. In *ICFP*. 66–77.

[11] Mark Christiaens and Koenraad De Bosschere. 2001. TRaDe: Data Race Detection for Java. In *International Conference on Computational Science*. 761–770.

[12] CIVL Distribution 2017. (2017). https://github.com/boogie-org/boogie

[13] Cormac Flanagan and Stephen N. Freund and James R. Wilcox 2017. VerifiedFT CIVL Implementation. (2017). https://github.com/boogie-org/boogie/blob/civl/Test/civl/verified-ft.bpl

[14] DRD: a thread error detector 2014. (2014). http://valgrind.org/docs/manual/drd-manual.html

[15] Matthew B. Dwyer and Lori A. Clarke. 1994. *Data Flow Analysis for Verifying Properties of Concurrent Programs*. Technical Report 94-045. Department of Computer Science, University of Massachusetts at Amherst.

[16] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-Juergen Boehm. 2012. IFRit: interference-free regions for dynamic data-race detection. In *OOPSLA*. 467–484.

[17] Dawson R. Engler and Ken Ashcraft. 2003. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP*.

[18] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. 2010. Effective Data-Race Detection for the Kernel. In *OSDI*. 151–162.

[19] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and precise dynamic race detection. In *PLDI*. 121–133.

[20] Cormac Flanagan and Stephen N. Freund. 2010. FastTrack: efficient and precise dynamic race detection. *Commun. ACM* 53, 11 (2010), 93–101.

[21] Cormac Flanagan and Stephen N. Freund. 2010. The RoadRunner dynamic analysis framework for concurrent programs. In *PASTE*. 1–8.

[22] Cormac Flanagan and Stephen N. Freund. 2013. RedCard: Redundant Check Elimination for Dynamic Race Detectors. In *ECOOP*. 255–280.

[23] Cormac Flanagan, Stephen N. Freund, Marina Lifshin, and Shaz Qadeer. 2008. Types for atomicity: Static checking and inference for Java. *ACM Trans. Program. Lang. Syst.* 30, 4 (2008).

[24] Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. 2004. Exploiting purity for atomicity. In *ISSTA*. 221–231.

[25] Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. 2005. Exploiting Purity for Atomicity. *IEEE Trans. Software Eng.* 31, 4 (2005), 275–291.

[26] Dan Grossman. 2003. Type-Safe Multithreading in Cyclone. In *Proceedings of the ACM Workshop on Types in Language Design and Implementation*. 13–25.

[27] Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. 2015. Automated and Modular Refinement Reasoning for Concurrent Programs. In *CAV*. 449–465.

[28] Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. 2015. *Automated and Modular Refinement Reasoning for Concurrent Programs*. Technical Report MSR-TR-2015-8. Microsoft Research.

[29] Jeff Huang and Arun K. Rajagopalan. 2017. What's the Optimal Performance of Precise Dynamic Race Detection? - A Redundancy Perspective. In *ECOOP*. 15:1–15:22.

[30] Intel. 2018. Intel Inspector. (2018). http://software.intel.com/en-us/intel-inspector-xe

[31] Intel. 2018. Intel Inspector Issues and Limitations. (2018). http://software.intel.com/en-us/intel-inspector-2018-release-notes-issues-and-limitations

[32] Java Grande Forum. 2017. Java Grande Benchmark Suite. (2017). http://www2.epcc.ed.ac.uk/computing/research_activities/jomp/grande.html

[33] Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (1975), 717–721.

[34] William Mansky, Yuanfeng Peng, Steve Zdancewic, and Joseph Devietti. 2017. Verifying dynamic race detection. In *Conference on Certified Programs and Proofs*. 151–163.

[35] Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java memory model. In *POPL*. 378–391.

[36] Friedemann Mattern. 1988. Virtual Time and Global States of Distributed Systems. In *Workshop on Parallel and Distributed Algorithms*.

[37] The Coq development team. 2017. *The Coq Reference Manual, version 8.6*. http://coq.inria.fr

[38] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective Static Race Detection for Java. In *PLDI*. 308–319.

[39] Hiroyasu Nishiyama. 2004. Detecting Data Races Using Dynamic Escape Analysis Based on Read Barrier. In *Virtual Machine Research and Technology Symposium*. 127–138.

[40] Robert O'Callahan and Jong-Deok Choi. 2003. Hybrid Dynamic Data Race Detection. In *PPOPP*. 167–178.

[41] Boris Petrov, Martin T. Vechev, Manu Sridharan, and Julian Dolby. 2012. Race detection for web applications. In *PLDI*. 251–262.

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

[43] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin T. Vechev, and Eran Yahav. 2012. Scalable and precise dynamic datarace detection for structured parallelism. In *PLDI*. 531–542.

[44] Dustin Rhodes, Cormac Flanagan, and Stephen N. Freund. 2017. BigFoot: Static check placement for dynamic race detection. In *PLDI*. 141–156.

[45] RoadRunner Team. 2016. RoadRunner Analysis Framework, Version 0.4. (2016). https://github.com/stephenfreund/RoadRunner/tree/618f3ae5a24f702719f6b6c0422fc1a488cf16bf

[46] RoadRunner Team. 2016. RoadRunner GitHub Source Code Commit 54ae0b0. (2016). https://github.com/stephenfreund/RoadRunner/tree/b1d39a192e6e2330a95408c7e4030f85354ae0b0

[47] RoadRunner Team. 2016. RoadRunner GitHub Source Code Commit 8b2e9a7. (2016). https://github.com/stephenfreund/RoadRunner/tree/a1f547350e90e7092a21dd1d95b1714528b2e9a7

[48] Caitlin Sadowski, Jaeheon Yi, Kenneth Knowles, and Cormac Flanagan. 2008. Proving correctness of a dynamic atomicity analysis in Coq. In *Workshop on Mechanizing Metatheory*, Vol. 8.

[49] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. *TOCS* 15, 4 (1997), 391–411.

[50] Edith Schonberg. 1989. On-The-Fly Detection of Access Anomalies. In *PLDI*. 285–297.

[51] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*. 62–71.

[52] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. 2011. Dynamic Race Detection with LLVM Compiler - Compile-Time Instrumentation for ThreadSanitizer. In *RV*.

110–114.

[53] Young Wn Song and Yann-Hang Lee. 2014. Efficient Data Race Detection for C/C++ Programs Using Dynamic Granularity. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 679–688.

[54] ThreadSanitizer 2018. ThreadSanitizer. (2018). http://clang.llvm.org/docs/ThreadSanitizer.html

[55] Christoph von Praun and Thomas Gross. 2001. Object Race Detection. In *OOPSLA*. 70–82.

[56] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. RELAY: static race detection on millions of lines of code. In *FSE*. 205–214.

[57] Liqiang Wang and Scott D. Stoller. 2005. Static analysis of atomicity for programs with non-blocking synchronization. In *PPOPP*. 61–71.

[58] James R. Wilcox, Parker Finch, Cormac Flanagan, and Stephen N. Freund. 2015. Array Shadow State Compression for Precise Dynamic Race Detection. In *ASE*. 155–165.

[59] Xinwei Xie and Jingling Xue. 2011. Acculock: Accurate and efficient detection of data races. In *CGO*. 201–212.

[60] Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP*. 221–234.