Verifying Dynamic Race Detection

William Mansky[†] Yuanfeng Peng[*] Steve Zdancewic[*] Joseph Devietti[*]

[†]Princeton University [*]University of Pennsylvania

wmansky@cs.princeton.edu, yuanfeng@cis.upenn.edu, stevez@cis.upenn.edu, devietti@cis.upenn.edu

Abstract

Writing race-free concurrent code is notoriously difficult, and races can result in bugs that are difficult to isolate and reproduce. Dynamic race detection can catch races that cannot (easily) be detected statically. One approach to dynamic race detection is to instrument the potentially racy code with operations that store and compare metadata, where the metadata implements some known race detection algorithm (e.g. vector-clock race detection). In this paper, we describe the process of formally verifying several algorithms for dynamic race detection. We then turn to implementations, laying out an instrumentation pass for race detection in a simple language and presenting a mechanized formal proof of its correctness: all races in a program will be caught by the instrumentation, and all races detected by the instrumentation are possible in the original program.

1. Introduction

Multicore processors have steadily invaded a broad swath of the computing ecosystem in everything from datacenters to smart watches. Writing multithreaded code for such platforms can bring good performance but also a host of programmability challenges. One of the key challenges is dealing with data races, as the presence of a data race introduces non-sequentially-consistent [10] and in some cases undefined [1] behavior into programs, making program semantics very difficult to understand. Races are not detected by default in current language runtimes, though there are many systems that provide sound and complete data race detection via dynamic analysis [5, 6, 18] to help programmers detect and remove data races from their programs.

While these analyses have been proven correct, such proofs have two main shortcomings: they are proofs of the algorithms, not of the implementations, and they are paper proofs instead of machine-checked proofs. As a consequence, it is possible that a race detector does not faithfully implement its algorithm, or that the algorithm itself is not fully correct. Moreover, it still remains to be shown that the implementation, often done via code instrumentation, does not itself introduce or mask races, and that in the absence of races, the program's behavior is unchanged.

We seek to rectify these concerns and place dynamic race detection on a provably correct foundation for the first time. We begin by formalizing the proof of the classic vector-clock race detection algorithm [4, 11] using the Coq interactive theorem prover [17].[1] Having established the correctness of this base algorithm, we extend our work in two dimensions.

We first explore the **algorithmic dimension**, formally establishing the correctness of the FASTTRACK algorithm [5]. FASTTRACK includes several significant optimizations over the base vector-clock algorithm. We find that the correctness of FASTTRACK can be demonstrated by proving its equivalence to the vector-clock algorithm, which is a more straightforward process than demonstrating correctness in isolation, and likely lends itself to formalizing additional algorithms with reduced effort. We also verify an extension of the base vector-clock algorithm that allows multiple joins to the same thread, as modern threading packages do.

We next explore the **implementation dimension**, formally establishing in Coq the correctness of an implementation of vector-clock race detection on a simple imperative language with threads. Given a program written in our language, our race detector adds instrumentation, written in the same language, to perform vector-clock race detection on the program. We demonstrate that this instrumentation correctly implements the vector-clock algorithm, again leveraging our previous verification effort. We also prove that our implementation preserves the program's semantics in the absence of races. This verification is much more difficult than proving the algorithm correct, since it must deal directly with the details of the implementation language and the concurrency model; we must precisely define the relation between uninstrumented and instrumented memory states, and prove noninterference of the code added by the instrumentation.

To the best of our knowledge, ours is the first work to adopt formal verification for either race detection algorithms *or* their implementations. Moreover, we believe our general

[1] All proofs are fully machine-checked in Coq, and the development has been submitted with the paper.

approach may be useful as a template for verifying a broad range of dynamic analyses, especially in the challenging domain of analyses for parallel programs. Verification helps ensure that debugging tools are themselves free from bugs.

This paper makes the following contributions:

- We present a method for verifying a dynamic data race detector from the algorithmic level to its implementation.
- We give the first machine-checked proofs of the vector-clock and FASTTRACK race-detection algorithms.
- We describe the first verified implementation of vector-clock race detection on a simple, multithreaded language.

Along the way, we discovered that one lemma in the paper proof of FASTTRACK was too strong, and noticed a minor discrepancy between the FASTTRACK algorithm and its implementation. In this case the effects were benign, but more generally, this highlights the value of formal verification for obtaining correct algorithms and implementations.

The remainder of this paper is organized as follows. In Section 2, we lay out our approach to verifying dynamic race detection algorithms and their implementations. In Section 3, we state and verify several algorithms for race detection. In Sections 4 and 5, we describe an instrumentation pass that implements dynamic race detection, and explain the verification process in detail. We compare our approach to related work in Section 7, and evaluate our results and describe future work in Section 8.

2. Proof Strategy

Our goal for each algorithm and program transformation presented in this paper is to prove that it implements sound and complete race detection, i.e., that it raises an alarm in all racy executions and only racy executions. However, as much as possible, we prefer not to do this by referring back to the base definition of racy executions. We begin by stating and proving correctness of a simple vector-clock race detection algorithm, by direct relation to the definition of a race. We then prove further results—the correctness of a more sophisticated algorithm, and of an instrumentation pass meant to implement the simple algorithm—by relating them to the verified base algorithm. We may think of this hierarchy in terms of specifications and refinement: we begin by proving that the base algorithm refines the abstract specification of race detection, and then use it in turn as a specification refined by more complex or detailed mechanisms. This approach is modular and avoids duplicate proof effort, but it also serves as further validation of the base vector-clock algorithm: by showing that it is *two-sided*, that is, that it both implements a higher-level specification of its desired behavior and is implemented by more concrete systems, we gain confidence that it is correctly stated (and not vacuous).

Our approach to showing that an instrumentation pass implements the vector-clock algorithm breaks down into several steps. We begin by defining the semantics of a target language, labeled with the abstract race detection operations produced by each step. This means that from each execution of an uninstrumented program, we can use the base algorithm to determine the behavior we *would have observed* if the program was correctly instrumented. Next, we define our instrumentation pass, and also write a bigger-step semantics for instrumented programs in which an instruction executes together with its instrumentation in a single step. This strategy is analogous to that used in other proof efforts [13], and makes it easier to directly relate executions of an instrumented program to executions of the original program. Given the bigger-step semantics, the proof of correctness of the instrumentation then breaks down into two parts: a proof of bisimulation between the bigger-step semantics and the "would have observed" semantics of the uninstrumented program, and a proof that the bigger-step semantics completely captures the possible behaviors of any instrumented program. This approach has potential applications beyond race detection: we believe that it could be applied to simplify the verification of any kind of instrumentation for dynamic checks, including memory safety and atomicity violation checking. The approach is particularly useful when verifying instrumentation of concurrent programs, since we are able to isolate all reasoning about interference between threads in the latter half of the third step—characterizing the possible behaviors of the instrumented program—and otherwise reason more or less sequentially.

3. Race Detection Algorithms

We begin by reviewing the classic vector-clock race detection algorithm [4, 11], which we call VECTORCLOCK, and briefly describing its verification in Coq. We then turn to the FASTTRACK [5] algorithm and its verification both by bisimulation with VECTORCLOCK and by direct proof, followed by the verification of a variant of VECTORCLOCK that handles multiple joins to the same thread.

3.1 Defining Data Races

Data races are formally defined in terms of a *happens-before* relation $<_{\mathrm{hb}}$, a partial order over events in a program trace [7]. Events consist of memory operations $rd(t, x)$ and $wr(t, x)$ where a thread t reads or writes a memory location x, lock operations $acq(t, m)$ and $rel(t, m)$ where a thread t acquires or releases a lock m, $fork(t, u)$ where a thread t creates a new thread u, and $join(t, u)$ where a thread t joins with a thread u. Given events a and b, we say *a happens before b* (and *b happens after a*), written $a <_{\mathrm{hb}} b$, if: (1) a precedes b in program order in the same thread; or (2) a precedes b in synchronization order $<_{\mathrm{sw}}$, *e.g.*, if a is $rel(t, m)$ and b is $acq(u, m)$; or (3) (a, b) is in the transitive closure of program order and synchronization order. Two events not ordered by happens-before are *concurrent*. Two memory accesses to the same location form a *data race* if they are concurrent and at least one is a write.

$$\textsc{Acquire}\ \frac{C' = C[t := C_t \sqcup L_m]}{(C, L, R, W) \xRightarrow{acq(t,m)} (C', L, R, W)}$$

$$\textsc{Release}\ \frac{L' = L[m := C_t] \quad C' = C[t := inc_t(C_t)]}{(C, L, R, W) \xRightarrow{rel(t,m)} (C', L', R, W)}$$

$$\textsc{Fork}\ \frac{C' = C[u := C_u \sqcup C_t, t := inc_t(C_t)]}{(C, L, R, W) \xRightarrow{fork(t,u)} (C', L, R, W)}$$

$$\textsc{Join}\ \frac{C' = C[t := C_t \sqcup C_u, u := inc_u(C_u)]}{(C, L, R, W) \xRightarrow{join(t,u)} (C', L, R, W)}$$

$$\textsc{Read}\ \frac{W_x \sqsubseteq C_t \quad R' = R[x := R_x[t := C_t(t)]]}{(C, L, R, W) \xRightarrow{rd(t,x)} (C, L, R', W)}$$

$$\textsc{ReadNoChange}\ \frac{R_x(t) = C_t(t)}{(C, L, R, W) \xRightarrow{rd(t,x)} (C, L, R, W)}$$

$$\textsc{Write}\ \frac{R_x \sqsubseteq C_t \quad W_x \sqsubseteq C_t \quad W' = W[x := W_x[t := C_t(t)]]}{(C, L, R, W) \xRightarrow{wr(t,x)} (C, L, R, W')}$$

$$\textsc{WriteNoChange}\ \frac{W_x(t) = C_t(t)}{(C, L, R, W) \xRightarrow{wr(t,x)} (C, L, R, W)}$$

Figure 1. VECTORCLOCK operational semantics

3.2 Vector-Clock Race Detection

One common algorithm for dynamic race detection is to use *vector clocks* to track the happens-before relation during execution [4, 11]. A vector clock V stores one (nonnegative) integer logical clock per thread; we write $V(t)$ for the data associated with thread t in clock V.

There are three key operations on vector clocks. *Union* is the element-wise maximum of two vector clocks: $V_1 \sqcup V_2 = V_3$ s.t. $\forall t.\ V_3(t) = \max(V_1(t), V_2(t))$. *Comparison* is the element-wise comparison of two vector clocks: $V_a \sqsubseteq V_b$ is defined to mean $\forall t.\ V_a(t) \le V_b(t)$. Finally, *increment* increases a single component of a vector clock, defined as $inc_t(V) = \lambda u.$ if $u = t$ then $V(u) + 1$ else $V(u)$.

The state of a vector-clock race detector is a tuple (C, L, R, W) of collections of vector clocks, where:

- Vector clock C_t stores the last time in each thread that happens before thread t's current logical time.
- Vector clock L_m stores the last time in each thread that happens before the last release of lock m.
- Vector clock R_x stores the time of each thread's last read of location x.
- Vector clock W_x stores the time of each thread's last write to location x.

Initially, all L, R, and W vector clocks are set to $\bot_V$, where $\forall t\ .\ \bot_V(t) = 0$. Each thread t's initial vector clock is C_t, where $C_t(t) = 1$ and $\forall u \neq t\ .\ C_t(u) = 0$.

VECTORCLOCK's operational semantics are presented in Figure 1. Each rule has the form $S \overset{a}{\Rightarrow} S'$, meaning that when the current vector clock state is S and the next event is a, the next vector clock state is S'. When a thread t acquires a lock m (the ACQUIRE rule) we update C_t to $C_t \sqcup L_m$. By acquiring lock m, thread t has synchronized with all events that happen before the last release of m, so all these events happen before all subsequent events in t. When t releases a lock m (the RELEASE rule), we update L_m to C_t, capturing all events that happen before this release in the lock. We then increment t's entry in its own vector clock C_t to ensure that subsequent events in t do not appear to happen before the release t just performed. The FORK and JOIN rules are similar to RELEASE and ACQUIRE, respectively. The increment $inc_u(C_u)$ in JOIN is needed to preserve the invariant that $C_u(u)$, thread u's entry for itself, is always higher than any other thread's entry for u.

When t reads a location x (the READ rule), we check if $W_x \sqsubseteq C_t$. If this check fails, there is a previous write to x that did not happen before this read, so there is a data race. (Note that the algorithm is considered to detect a race when it is *stuck*, that is, when there is no rule that can be applied for the next operation; when a state σ is stuck on an operation o, we write $\sigma \overset{o}{\not\Rightarrow}$.) Otherwise, we set t's entry in R_x to t's current logical clock, $C_t(t)$. It often arises that t will read the same location repeatedly without an intervening change in its own clock value $C_t(t)$. The READNOCHANGE rule covers this case, in which no metadata updates are necessary.

Writes operate similarly to reads, with an additional check in the WRITE rule that $R_x \sqsubseteq C_t$ to ensure that all previous reads of x are well-ordered before the current write. This is not necessary in the READ case because two concurrent reads are not considered to race.

3.2.1 Correctness

Correctness for dynamic race detection is phrased in terms of *soundness* and *completeness*. Soundness means that if the algorithm runs to completion, then there was no race in the program trace; completeness means that given a trace without races, the algorithm does not get stuck.

Theorem 1. VECTORCLOCK *is sound and complete.*

The soundness and completeness of VECTORCLOCK follows from the fact that the $\sqsubseteq$ relation between vector clocks precisely models the happens-before relation. Our Coq formalization follows the proof outline given in the presentation of FASTTRACK [5] (with the change described in Section 3.3.2), which can be straightforwardly translated into Coq. The main invariant of the algorithm is that $C_t(t) > C_u(t)$, that is, each thread always has a higher timestamp for

$$\textsc{ReadSameEpoch}\ \frac{\mathcal{R}_x = E(t)}{(\mathcal{C},\mathcal{L},\mathcal{R},\mathcal{W}) \xRightarrow{rd(t,x)} (\mathcal{C},\mathcal{L},\mathcal{R},\mathcal{W})}$$

$$\textsc{ReadExclusive}\ \frac{\begin{array}{c}\mathcal{R}_x \in \mathit{Epoch} \\ \mathcal{R}_x \preceq \mathcal{C}_t \qquad \mathcal{W}_x \preceq \mathcal{C}_t \\ \mathcal{R}' = \mathcal{R}[x := E(t)]\end{array}}{(\mathcal{C},\mathcal{L},\mathcal{R},\mathcal{W}) \xRightarrow{rd(t,x)} (\mathcal{C},\mathcal{L},\mathcal{R}',\mathcal{W})}$$

$$\textsc{ReadShare}\ \frac{\begin{array}{c}\mathcal{W}_x \preceq \mathcal{C}_t \\ \mathcal{R}_x = c@u \\ V = \bot_V[t := \mathcal{C}_t(t), u := c] \\ \mathcal{R}' = \mathcal{R}[x := V]\end{array}}{(\mathcal{C},\mathcal{L},\mathcal{R},\mathcal{W}) \xRightarrow{rd(t,x)} (\mathcal{C},\mathcal{L},\mathcal{R}',\mathcal{W})}$$

$$\textsc{ReadShared}\ \frac{\begin{array}{c}\mathcal{R}_x \in \mathit{Vector\ Clock} \\ \mathcal{W}_x \preceq \mathcal{C}_t \\ \mathcal{R}' = \mathcal{R}[x := \mathcal{R}_x[t := \mathcal{C}_t(t)]]\end{array}}{(\mathcal{C},\mathcal{L},\mathcal{R},\mathcal{W}) \xRightarrow{rd(t,x)} (\mathcal{C},\mathcal{L},\mathcal{R}',\mathcal{W})}$$

$$\textsc{WriteSameEpoch}\ \frac{\mathcal{W}_x = E(t)}{(\mathcal{C},\mathcal{L},\mathcal{R},\mathcal{W}) \xRightarrow{wr(t,x)} (\mathcal{C},\mathcal{L},\mathcal{R},\mathcal{W})}$$

$$\textsc{WriteExclusive}\ \frac{\begin{array}{c}\mathcal{R}_x \in \mathit{Epoch} \\ \mathcal{R}_x \preceq \mathcal{C}_t \qquad \mathcal{W}_x \preceq \mathcal{C}_t \\ \mathcal{W}' = \mathcal{W}[x := E(t)]\end{array}}{(\mathcal{C},\mathcal{L},\mathcal{R},\mathcal{W}) \xRightarrow{wr(t,x)} (\mathcal{C},\mathcal{L},\mathcal{R},\mathcal{W}')}$$

$$\textsc{WriteShared}\ \frac{\begin{array}{c}\mathcal{R}_x \in \mathit{Vector\ Clock} \\ \mathcal{R}_x \sqsubseteq \mathcal{C}_t \qquad \mathcal{W}_x \preceq \mathcal{C}_t \\ \mathcal{W}' = \mathcal{W}[x := E(t)] \\ \mathcal{R}' = \mathcal{R}[x := \bot_e]\end{array}}{(\mathcal{C},\mathcal{L},\mathcal{R},\mathcal{W}) \xRightarrow{wr(t,x)} (\mathcal{C},\mathcal{L},\mathcal{R}',\mathcal{W}')}$$

Figure 2. FASTTRACK operational semantics. FORK, JOIN, ACQUIRE and RELEASE are identical to VECTORCLOCK.

itself than any other thread has for it. This guarantees that no operation by a thread t can be seen by another thread u (without detecting a race) until t has synchronized with u.

3.3 FASTTRACK

The FASTTRACK algorithm [5] leverages the observation that, by the definition of a data race, all writes to a location must be totally ordered in race-free traces. FASTTRACK accordingly adopts a more sophisticated representation of vector clocks to save space and time. *Epochs* can often be used in place of vector clocks; an epoch $c@t$ holds a timestamp for just one thread, and is treated as a vector clock that is c for t and 0 for every thread other than t:

$$(c@t)(u) = \begin{cases} c & \text{if } t = u \\ 0 & \text{otherwise} \end{cases}$$

Because epochs have a single non-zero entry, an epoch can be compared with a vector clock, or another epoch, in O(1) time using the $\preceq$ operator. We say that $c@t \preceq V$ (and similarly $c@t \preceq e$) when $(c@t)(u) \leq V(u)$ for all u, which occurs exactly when $c \leq V(t)$. We write $\bot_e$ for a minimal epoch at an arbitrary thread, $0@t_0$.

A FASTTRACK analysis state is a tuple $(\mathcal{C},\mathcal{L},\mathcal{R},\mathcal{W})$ just as in VECTORCLOCK, except that $\mathcal{R}_x$ may be either a vector clock or an epoch, and $\mathcal{W}_x$ is always an epoch. FASTTRACK's initial analysis state is the tuple $(\lambda t.\, inc_t(\bot_V), \lambda l.\, \bot_V, \lambda x.\, \bot_e, \lambda x.\, \bot_e)$. Each thread initially has an empty vector clock with its own entry incremented, all locks have empty vector clocks, and all memory locations have empty read and write epochs. The FASTTRACK operational semantics are presented in Figure 2 (in which we use $E(t)$ to mean $\mathcal{C}_t(t)@t$). The READ and WRITE rules are split into several cases, as $\mathcal{R}_x$ transitions between an epoch and a full vector clock. When a write occurs to a location x for which $\mathcal{R}_x$ is a vector clock, $\mathcal{R}_x$ is set to an empty epoch $\bot_e$; conversely, when multiple concurrent reads to x occur, $\mathcal{R}_x$ is inflated to a full vector clock.

3.3.1 Proof by Bisimulation

Flanagan and Freund justify the optimizations of FASTTRACK by proving that the $\sqsubseteq$ relation on vector clocks still precisely captures the happens-before relation. However, as part of the proof strategy outlined in Section 2, we take a different approach, proving that the transition system of FASTTRACK is bisimilar to that of VECTORCLOCK. Since we have already verified VECTORCLOCK, this is sufficient to guarantee that FASTTRACK is sound and complete. In this section, we describe our novel bisimulation proof of FASTTRACK's correctness; in the following section, we describe our mechanization of the paper proof.

Intuitively, the metadata optimizations of FASTTRACK are safe to perform because the reduced metadata still captures the same happens-before relationships, which means that the same $\sqsubseteq$ relationships hold between corresponding state components. Thus, we need only present a relation between FASTTRACK states and full vector clock states that captures this correspondence in order to prove a bisimulation between the two systems. The C and L components should remain unchanged. To characterize the expected semantics of epochs, we define the following *emulation* relation:

Definition 1. *An epoch e* emulates *a vector clock V for another vector clock V' if $e \preceq V'$ implies that $V \sqsubseteq V'$. An epoch e emulates V in a state (C, L, R, W) if $\forall u.\ e$ emulates V for C_u and $\forall m.\ e$ emulates V for L_m.*

In FASTTRACK, when a write vector clock W_x is collapsed to an epoch $c@t$, it is precisely because $W_x(t) = c$ and $c@t$ emulates W_x. Figure 3 shows a simple program with two threads that illustrates write epoch emulation. VECTORCLOCK's W_x retains information about both threads' writes, but because t_0's write happens before t_1's

t_0	t_1	VC W_x	FT $\mathcal{W}_x$
write x		[1,0]	1@t_0
rel m		”	”
	acq m	”	”
	write x	[1,1]	1@t_1

Figure 3. An example of how FASTTRACK's write epoch emulates the conventional write vector clock.

	VC		FT	
t_0	R_x	W_x	$\mathcal{R}_x$	$\mathcal{W}_x$
read x	[1,1]	[0,0]	[1,1]	$\perp_e$
write x	”	[1,0]	$\perp_e$	1@t_0

Figure 4. An example of how FASTTRACK's write epoch emulates the conventional read vector clock.

t_0	t_1	t_2	VC R_x	FT $\mathcal{R}_x$
read x			[1,0,0]	1@t_0
rel m			”	”
	acq m		”	”
	read x		[1,1,0]	1@t_1
		read x	[1,1,1]	[0,1,1]

Figure 5. An example of how FASTTRACK's read vector clock partially emulates the conventional read vector clock.

write, the FASTTRACK write epoch retains information only about t_1's write. If t_1's write happens before some vector clock V', then t_0's write must happen before V' as well, so the write epoch emulates the write vector clock.

The relation for read metadata is more complicated, because there are more ways in which read data may be dropped. The WRITESHARED rule resets read metadata to an empty epoch, erasing all previous values, and as long as $\mathcal{R}_x$ remains an epoch, any information about previous reads is lost. This has two effects on the simulation relation. First, we must make a special allowance for the case in which the read metadata is empty; in this case, it is the write epoch that emulates the original read vector clock. Figure 4 shows an example program illustrating this case. If the location x is read by thread t_1, then read by t_0 (as shown), t_0's write will trigger the WRITESHARED rule, which clears FASTTRACK's $\mathcal{R}_x$ metadata, losing track of the reads. However, because the reads happen before t_0's write, FASTTRACK's write epoch still emulates the conventional R_x vector clock: if t_0's write happens before some vector clock V' the (lost) reads happen before V' as well.

The second effect is that even when the FASTTRACK state holds a vector clock in $\mathcal{R}_x$, that vector clock may contain some 0 values for threads that have in fact read x. In effect, because FASTTRACK read vector clocks are always derived from epochs, they carry forward a partial emulation relation, in which one component emulates all the relationships on zeroed values. Figure 5 shows an example program where each thread performs a read. t_0's read happens before t_1's read, causing FASTTRACK to discard t_0's read. t_2's read is concurrent with both other reads, so it forces FASTTRACK's R_x into vector clock format. The R_x entry for t_1 emulates the entry for t_0, since if t_1's read happens before some vector clock V', t_0's read must happen before V' as well. We capture this relation as follows:

Definition 2. *A vector clock V_0* partially emulates *a vector clock V for another vector clock V' if there is some thread t such that:*

- *for all u such that $V_0(u) = 0$, $V_0(t) \leq V'(t)$ implies that $V(u) \leq V'(u)$*
- *for all u such that $V_0(u) \neq 0$, $V_0(u) \leq V'(u)$ implies that $V(u) \leq V'(u)$*

A vector clock V_0 partially emulates a vector clock V in a state (C, L, R, W) if $\forall u$. V_0 partially emulates V for C_u and $\forall m$. V_0 partially emulates V for L_m.

We can now state the full simulation relation between VECTORCLOCK and FASTTRACK states.

Definition 3. *A* VECTORCLOCK *state (C, L, R, W) and a* FASTTRACK *state $(\mathcal{C}', \mathcal{L}', \mathcal{R}', \mathcal{W}')$ are in the relation $\sim$ when $\mathcal{C}' = C$, $\mathcal{L}' = L$, and for every location x:*

- *if $\mathcal{W}'_x = c@t$, then $W_x(t) = c$ and $c@t$ emulates W_x*
- *$\mathcal{R}'_x(t) \leq R_x(t)$ for all t*
- *if $\mathcal{R}'_x = \perp_e$, then $\mathcal{W}'_x$ emulates R_x*
- *if $\mathcal{R}'_x = c@t$, then $R_x(t) = c$ and $c@t$ emulates R_x*
- *if $\mathcal{R}'_x = V$, then V partially emulates R_x*

Lemma 1. *The relation $\sim$ is a bisimulation.*

Proof. In each direction, the relation $\sim$ is preserved by corresponding steps in the two systems, which we show by case analysis on the rule applied. ☐

Theorem 2. FASTTRACK *is sound and complete.*

Proof. For each successful execution of FASTTRACK, there is a successful execution of VECTORCLOCK on the same trace; by Theorem 1 the trace is race-free. Conversely, for each race-free trace, there is a successful execution of VECTORCLOCK, so there is also one using FASTTRACK. ☐

While the statement of the simulation relation is complicated, once it is correctly stated, the proof itself is reduced to proving that various $\leq$ relationships are preserved by mathematical operations on vector clocks.

3.3.2 Direct Proof

As a baseline for comparison, we formalized the original proof of FASTTRACK's correctness in Coq as well, directly relating the $\sqsubseteq$ relation on vector clocks and epochs to the happens-before relation. In the process, we discovered an error in the paper proof. FASTTRACK's Lemma 4 reads as follows: *Suppose σ is well-formed and $\sigma \overset{\alpha}{\Rightarrow} \sigma'$ and*

$a, b \in \alpha$. Let $t = tid(a)$ and $u = tid(b)$. If $a <_\alpha b$ then $\mathcal{K}^a(t) \sqsubseteq \mathcal{K}^b(t)$. ($\mathcal{K}^a(t)$ refers to t's vector clock at the time a was performed.) This lemma is then used to prove that if some operation b is *stuck*, then a previous operation a must have raced with it, since $\mathcal{K}^a(t) \not\sqsubseteq \mathcal{K}^b(t)$. However, the premise of Lemma 4 requires that a and b not be stuck, since they are in a trace α that successfully executes to a state σ'. Because of this constraint, the use of Lemma 4 in the proof of completeness is in fact invalid. Fortunately, this premise is stronger than necessary to prove Lemma 4, and in Coq we are able to prove a more general version:

Lemma 2 (FASTTRACK Lemma 4, Fixed)**.** *Suppose σ is well-formed and $\sigma \overset{\alpha}{\Rightarrow} \sigma'$ and $a \in \alpha$. Let $t = tid(a)$ and $u = tid(b)$. If $a <_{\alpha;b} b$ then $\mathcal{K}^a(t) \sqsubseteq \mathcal{K}^b(u)$.*

With this statement of the lemma, the completeness proof follows as outlined in the paper.

The size of this proof and the effort involved are comparable to that of the proof by bisimulation. The direct proof also requires a significant amount of inductive reasoning, which might have been difficult to synthesize without the guide of the paper proof. In this case, since FASTTRACK preserves the same invariants as VECTORCLOCK, its proof has largely the same structure as that of VECTORCLOCK; however, if a different algorithm modified some of these invariants, we believe that the proof by bisimulation would be significantly easier to construct than the proof that recapitulates the relationship between $\sqsubseteq$ and happens-before.

3.4 Handling Multiple Joins

In many concurrent languages, once a thread has terminated, it is possible for multiple threads to join with it. According to the JOIN rule of Figure 1, on a *join* operation, C_t is updated to $C_t \sqcup C_u$ and $C_u(u)$ is incremented, maintaining the invariant that $C_u(u) > C_t(u)$. This increment is only necessary to maintain the invariant, since u will perform no more operations and does not require race detection, but it appears harmless. However, if we consider the problem of implementing this algorithm, we see that the increment is potentially dangerous: if two threads join with u concurrently, the updates to C_u will race. (See Section 4.2.1 for more on races in instrumentation.)

If we remove the unnecessary increment, then there is no risk of races between joins: both threads read C_u, but only modify their own vector clocks, and concurrent reads do not constitute a race. Without this increment, however, we must modify the invariants of the race detection algorithm; in particular, we must allow the basic invariant to be violated for terminated threads. One possible approach is to add an event $exit(t)$ produced when a thread terminates, and to extend the state with an additional component, a set X of threads that have $exit$ed. A thread can then only perform operations before it $exit$s, and only be the target of a *join* after it $exit$s. With these changes, we obtain a new vector clock race detection algorithm that supports multiple joins.

Theorem 3. VECTORCLOCK *modified for multiple joins is sound and complete.*

We have verified this modified algorithm in Coq by modifying the proofs of VECTORCLOCK; note that since it accepts traces that are rejected by VECTORCLOCK (in the original presentation of VECTORCLOCK, multiple joins to the same thread are ill-formed), we cannot hope to prove its correctness by simulation.

4. Race Detection Instrumentation

In the previous section, we described the statement and verification of *algorithms* for dynamic race detection. For these algorithms to be useful, they must be implemented in a tool that runs during the execution of a program. Most commonly, this is achieved by *instrumenting* the program with additional instructions that carry out the algorithm. Instrumentation must be defined in terms of the instructions of some language instead of abstract arithmetic operations, and must take place within the flow of a program rather than alongside it. Because of this, the process of translating from algorithm to instrumentation is error-prone, and any discrepancy weakens the guarantee provided by the paper proof.

In practice, we found that the implementation of FASTTRACK in the RoadRunner framework uses a different version of the RELEASE rule from that shown in Figure 1: the release handler updates L_m to $L_m \sqcup C_t$ rather than C_t. In this case, the difference can be shown not to affect the correctness of the algorithm, and the FASTTRACK authors have confirmed that it has no impact on performance.

To avoid such discrepancies, we must verify both the algorithm and the instrumentation pass that implements it. In this and the following sections, we present an instrumentation pass in a simple language that is formally proven to implement the VECTORCLOCK algorithm of Section 3.2.

4.1 The Language

We begin with a multithreaded language just complicated enough to have races and implement race detection instrumentation. The instructions of the language are inductively defined as follows, where n is a natural number, a is a local variable, e, e_1, e_2 are $expr$s, x is a memory location, l is a lock, t is a numeric thread id, and li is a list of $instr$s:

$$\begin{array}{rcl} e & ::= & n \mid a \mid e_1 + e_2 \mid \mathtt{max}(e_1, e_2) \\ instr & ::= & a \mathbin{\mathtt{:=}} e \mid a \mathbin{\mathtt{:=}} \mathtt{load}\ x \mid \mathtt{store}\ e\ x \mid \mathtt{lock}\ l \\ & \mid & \mathtt{unlock}\ l \mid \mathtt{spawn}\ t\ li \mid \mathtt{wait}\ t \\ & \mid & \mathtt{assert}(e_1 \mathbin{\mathtt{<=}} e_2) \end{array}$$

A *program* is a list of instructions, which will serve as the body of the thread with id 0. The dynamic state of a program contains a collection P of thread states—each a triple (t, li, ρ) of a thread id, a list of instructions (we write $\cdot$ for the empty list), and an environment mapping local variables to values—alongside a memory state[2] m. We also include a special error state err for detected races (i.e., failed `assert`s). We give semantics to the language via a labeled transition system: each step is labeled with the race detection operation it produces, if any. These operations do not effect the program's execution, but let us track the behavior that "would have happened" if the VECTORCLOCK algorithm was run in parallel with the program, which serves as a reference for the correctness of the instrumentation. In each step, we break the thread states into a composition $P \uplus (t, li, \rho)$ of an arbitrary thread t and the remaining threads, and execute the next instruction in t, updating the state as necessary. Thus, the transition rules for the language are of the form $(P, m) \xrightarrow{o}_t (P', m')$, where t is the executing thread and o is the race detection operation produced, if any.

The semantics of the language is shown in Figure 6. Note that we implement locks as memory locations in which we store 0 if the lock is free or $t + 1$ if it is held by thread t; we assume a strict separation between locks and normal memory locations. The initial environment G_0 maps all variables to 0, and the initial memory m_0 likewise maps all locations to 0. We call a state P *final* if all of its threads have executed to completion, with no instructions remaining; err is also considered final. A *result* of a program $prog$ is a final state R such that $((0, prog, G_0), m_0) \xrightarrow{\vec{o}}{}^* R$; depending on the order in which threads are interleaved, a program (even a race-free one) may have many possible results.

$$\begin{array}{c}(P \uplus (t, a \texttt{ := } e; li, G), m) \to_t \\ (P \uplus (t, li, G[a \mapsto \mathsf{eval}(G, e)]), m)\end{array}$$

$$\begin{array}{c}(P \uplus (t, a \texttt{ := load } x; li, G), m) \xrightarrow{rd(t,p)}_t \\ (P \uplus (t, li, G[a \mapsto m(x))], m)\end{array}$$

$$\begin{array}{c}(P \uplus (t, \texttt{store } e\ p; li, G), m) \xrightarrow{wr(t,x)}_t \\ (P \uplus (t, li, G), m[x \mapsto \mathsf{eval}(G, e)])\end{array}$$

$$\frac{m(\ell) = 0}{\begin{array}{c}(P \uplus (t, \texttt{lock } \ell; li, G), m) \xrightarrow{acq(t,\ell)}_t \\ (P \uplus (t, li, G), m[\ell \mapsto t + 1])\end{array}}$$

$$\frac{m(\ell) = t + 1}{\begin{array}{c}(P \uplus (t, \texttt{unlock } \ell; li, G), m) \xrightarrow{rel(t,\ell)}_t \\ (P \uplus (t, li, G), m[\ell \mapsto 0])\end{array}}$$

$$\frac{\forall li' G'.\ (u, li', G') \notin P}{\begin{array}{c}(P \uplus (t, \texttt{spawn } u\ li'; li, G), m) \xrightarrow{fork(t,u)}_t \\ (P \uplus (t, li, G) \uplus (u, li', G_0), m)\end{array}}$$

$$\frac{(u, \cdot, G') \in P}{\begin{array}{c}(P \uplus (t, \texttt{wait } u; li, G), m) \xrightarrow{join(t,u)}_t \\ (P \uplus (t, li, G), m)\end{array}}$$

$$\frac{\mathsf{eval}(G, e_1) \leq \mathsf{eval}(G, e_2)}{\begin{array}{c}(P \uplus (t, \texttt{assert(} e_1 \texttt{ <= } e_2 \texttt{);} li, G), m) \to_t \\ (P \uplus (t, li, G), m)\end{array}}$$

$$\frac{\mathsf{eval}(G, e_1) > \mathsf{eval}(G, e_2)}{(P \uplus (t, \texttt{assert(} e_1 \texttt{ <= } e_2 \texttt{);} li, G), m) \to_t err}$$

Figure 6. Semantics of the simple language

4.2 Instrumentation

We instrument programs by augmenting each race-detection-relevant instruction with a code snippet implementing the corresponding VECTORCLOCK rule. We begin by defining macros for the basic mathematical operations of the algorithm, as shown in Figure 7. These macros make use of two pieces of data from the program to be instrumented: `tmp1` and `tmp2` are two local variables unused in the program, which will be used as temporaries for the instrumentation, and z is the largest thread id generated in the lifetime of the program. (Note that in our simple language, z can be determined statically; in real-world implementations, the vector clock data would need to support dynamic resizing.)

With these macros as building blocks, we can straightforwardly translate the rules of VECTORCLOCK from Figure 1 into code. For this purpose, we use a CompCert-style block-offset memory model [8]; memory references have the form $(block, offset)$, and locations accessed in the original program are considered to be blocks of size 1. We set aside dedicated areas of memory for each of the components of the vector clock state, labeled C, L, R, and W accordingly, and index into them by assigning a unique numeric identifier to each thread, local variable, and lock, so that offset t into block $L[m]$ in memory contains the value of $L_m(t)$ (we write $L[m]$ as a shorthand for $L + m$, using the commonly understood equivalence between pointers and arrays). Note that the ordering of the associated vector clock operations depends on the instruction being instrumented; in particular, the blocking operations `lock` and `wait` must not change the vector clock state until after they successfully complete. Because in our language the bodies of future threads are embedded in the `spawn` instructions, we instrument these threads by recursively instrumenting each instruction in their bodies. The instrumented version of $prog$ is then simply $\mathsf{instrument}(prog) \triangleq [\![prog]\!]_0$, the result of applying the instrumentation function (Figure 8) to each instruction in $prog$.

4.2.1 Necessary Synchronization

A direct translation of the race detection rules into code cannot implement sound and complete race detection for one important reason: when a race does occur, the corresponding

[2] While our semantics is phrased in terms of memory states and updates, the Coq implementation uses an approach based on previous work by some of the authors [9], in which memory is represented as a sequence of operations and program and memory semantics are stated separately. This approach is convenient for verification, but its details are orthogonal to race detection instrumentation.

```
move(p, q)       ≜   tmp1 := load p;
                     store tmp1 q;

                     // V := V'
set(a, b)        ≜   move((a, 0), (b, 0));
                     ...
                     move((a, z − 1), (b, z − 1));

                     // C = C[t := inc_t(C_t)]
inc(b, o)        ≜   tmp1 := load (b, o);
                     tmp1 := tmp1 + 1;
                     store tmp1 (b, o);

max(p, q)        ≜   tmp1 := load p;
                     tmp2 := load q;
                     tmp2 := max tmp1 tmp2;
                     store tmp2 p;

                     // C = C ⊔ C'
merge(a, b)      ≜   max((a, 0), (b, 0));
                     ...
                     max((a, z − 1), (b, z − 1));

lea(p, q)        ≜   tmp1 := load p;
                     tmp2 := load q;
                     assert(tmp1 <= tmp2);

                     // C ⊑ C'
hb_check(a, b)   ≜   lea((a, 0), (b, 0));
                     ...
                     lea((a, z − 1), (b, z − 1));
```

Figure 7. Helper macros

```
⟦a := load x⟧_t    ≜   lock X[x];
                       hb_check(W[x], C[t]);
                       move((C[t], t), (R[b], t));
                       a := load x;
                       unlock X[x];

⟦store e x⟧_t      ≜   lock X[x];
                       hb_check(W[x], C[t]);
                       hb_check(R[b], C[t]);
                       move((C[t], t), (W[x], t));
                       store e x;
                       unlock X[x];

⟦lock m⟧_t         ≜   lock m;
                       merge(L[m], C[t]);

⟦unlock m⟧_t       ≜   set(C[t], L[m]);
                       inc(t, C[t]);
                       unlock m;

⟦spawn u li⟧_t     ≜   merge(C[t], C[u]);
                       inc(t, C[t]);
                       spawn u (⟦li⟧_u);

⟦wait u⟧_t         ≜   wait u;
                       merge(C[u], C[t]);
                       inc(u, C[u]);
```

Figure 8. Instrumented versions of each instruction, with instrumentation highlighted in gray.

//hb_check($W[x], C[t_1]$)	//hb_check($W[x], C[t_2]$)
tmp1 := load ($W[x], 0$)	tmp1 := load ($W[x], 0$)
...	...
	//hb_check($R[x], C[t_2]$)
	tmp1 := load ($R[x], 0$)
...	...
store tmp1 ($R[x], t_1$)	store tmp1 ($W[x], t_2$)
a := load x	store 2 x

Figure 9. Races in instrumentation

instrumentation also races. In instrumented programs such as that shown in Figure 9, depending on the ordering of updates to metadata locations, the instrumentation may fail to detect the race between the two threads. In general, poorly synchronized race detection instrumentation cannot hope to successfully detect all races. At the same time, adding too much synchronization could significantly hurt performance.

Given the set of operations available in our language, we can show that it suffices to add a lock for each memory location, which is used to protect the instrumentation on that memory location. (Intuitively, the other cases do not require extra locks because locks prevent races on their associated metadata, and a thread cannot race with the thread that spawns it or waits for it to terminate; the instrumentation on two different locations also never conflicts.) To implement the necessary synchronization in our instrumentation, we add another designated area of memory, X, such that $X[x]$ holds the lock protecting the metadata for x, and add locking to the `load` and `store` instrumentation, as reflected in Figure 8. This guarantees that the instrumentation will never race with itself, which is sufficient to allow us to prove correctness of the instrumentation in the next section.

5. Verifying Instrumentation

We can now prove the correctness of the instrumentation pass we have described. We verify the instrumentation pass by showing that the instrumentation records the same information and performs the same checks as the VECTORCLOCK algorithm would perform on the input program. Our verification strategy is as follows: first, we define a bigger-step semantics for instrumented programs in our target language. In this semantics, an instruction and its instrumentation execute together in a single step. We can show that every behavior of an instrumented program under the small-step semantics is equivalent to one in this bigger-step semantics, using "reordering" lemmas that let us gather all of the steps in an instrumentation section into a consecutive sequence. Second, we use a bisimulation argument to show that every execution of an uninstrumented program is matched by an execution of the instrumented program with the same behavior, taking advantage of the bigger-step relation to characterize the precise correspondence between instrumented and uninstrumented steps. This bisimulation allows us to conclude that for every race-free behavior of a program there exists a corresponding successful run of the instrumented program and vice versa, and similarly that for every racy behavior of a program there exists a corresponding failing run of the instrumented program and vice versa.

5.1 Bigger-Step Semantics and Reordering

Key to our correctness proof is the idea that the instrumented program can be seen as executing under a bigger-step semantics in which each instruction and its instrumentation execute in a single step. These steps are of the form $(P, m) \Rightarrow_t R$, as shown in Figure 10. Each step may modify the temporary variables in an arbitrary way, but otherwise performs a combination of the underlying operation and the corresponding checks and changes to the metadata. Instrumentation for `load` and `store` may also fail a check and step to the *err* state. We can show that each bigger step corresponds to a sequence of steps in the small-step semantics[3]:

Lemma 3. *If* $(P, m) \Rightarrow_t (P', m')$*, then* $(P, m) \rightarrow^* (P', m')$.

Proof. By case analysis and the small-step rules. $\square$

Proving the correspondence in the other direction is harder. Any given execution of an instrumented program may not line up with one in which the instrumentation for each instruction executes in a single step; at a state in the middle of the execution, the program may be executing as many different instrumentation sections as there are threads. We resolve this problem by "reordering" the steps of any execution so that the instrumentation for each instruction executes contiguously. More precisely, we show that each execution of an instrumented program is equivalent to one in which the instrumentation executes atomically.

We begin by defining the uninstrumented state represented by each intermediate state of the execution:

Definition 4. *An instrumented state P is an* instrumented suffix *of an uninstrumented state P_0 if P_0 and P contain the same threads and for each thread t:*

- *if $P_0(t) = \cdot$, then $P(t) = \cdot$*
- *if $P_0(t) = i; li$, then $P(t) = li_i; [\![li]\!]_t$, where li_i is some non-empty suffix of $[\![i]\!]_t$*

[3] From now on, we omit the race detection operation labeling on steps taken by instrumented programs; it may differ arbitrarily from the labels on the original program and is not relevant to detecting races.

$$\frac{\begin{array}{c} m(X[x]) = 0 \quad \forall o.\, m(W[x], o) \le m(C[t], o) \\ m' = m[(R[x], t) \mapsto m(C[t], t)] \end{array}}{\begin{array}{c} (P \uplus (t, [\![a\ \texttt{:=}\ \texttt{load}\ x]\!]_t \,; li, G), m) \Rightarrow_t \\ (P \uplus (t, li, G[a \mapsto m(x), \texttt{tmp1} \mapsto ?, \texttt{tmp2} \mapsto ?]), m') \end{array}}$$

$$\frac{m(X[x]) = 0 \quad \exists o.\, m(W[x], o) > m(C[t], o)}{(P \uplus (t, [\![a\ \texttt{:=}\ \texttt{load}\ x]\!]_t \,; li, G), m) \Rightarrow_t err}$$

$$\frac{\begin{array}{c} m(X[x]) = 0 \quad \forall o.\, m(W[x], o) \le m(C[t], o) \\ \forall o.\, m(R[x], o) \le m(C[t], o) \\ m' = m[x \mapsto \mathsf{eval}(G, e), (W[x], t) \mapsto m(C[t], t)] \end{array}}{\begin{array}{c} (P \uplus (t, [\![\texttt{store}\ e\ x]\!]_t \,; li, G), m) \Rightarrow_t \\ (P \uplus (t, li, G[\texttt{tmp1} \mapsto ?, \texttt{tmp2} \mapsto ?]), m') \end{array}}$$

$$\frac{\begin{array}{c} m(X[x]) = 0 \\ \exists o.\, m(W[x], o) > m(C[t], o) \vee m(R[x], o) > m(C[t], o) \end{array}}{(P \uplus (t, [\![\texttt{store}\ e\ x]\!]_t \,; li, G), m) \Rightarrow_t err}$$

$$\frac{\begin{array}{c} m(\ell) = 0 \quad m' = m[\ell \mapsto t + 1, \\ C[t] \Mapsto \mathsf{max}(m(L[m]), m(C[t]))] \end{array}}{\begin{array}{c} (P \uplus (t, [\![\texttt{lock}\ \ell]\!]_t \,; li, G), m) \Rightarrow_t \\ (P \uplus (t, li, G[\texttt{tmp1} \mapsto ?, \texttt{tmp2} \mapsto ?]), m') \end{array}}$$

$$\frac{\begin{array}{c} m(\ell) = t + 1 \\ m' = m[\ell \mapsto 0, L[m] \Mapsto m(C[t]), \\ (C[t], t) \mapsto m(C[t], t) + 1] \end{array}}{\begin{array}{c} (P \uplus (t, [\![\texttt{unlock}\ \ell]\!]_t \,; li, G), m) \Rightarrow_t \\ (P \uplus (t, li, G[\texttt{tmp1} \mapsto ?, \texttt{tmp2} \mapsto ?]), m') \end{array}}$$

$$\frac{\begin{array}{c} m' = m[C[u] \Mapsto \mathsf{max}(m(C[u]), m(C[t])), \\ (C[t], t) \mapsto m(C[t], t) + 1] \end{array}}{\begin{array}{c} (P \uplus (t, [\![\texttt{spawn}\ u\ li']\!]_t \,; li, G), m) \Rightarrow_t \\ (P \uplus (t, li, G[\texttt{tmp1} \mapsto ?, \texttt{tmp2} \mapsto ?]) \uplus (u, [\![li']\!]_u, G_0), m') \end{array}}$$

$$\frac{\begin{array}{c} (u, \cdot, G_0) \in P \\ m' = m[C[t] \Mapsto \mathsf{max}(m(C[t]), m(C[u])), \\ (C[u], u) \mapsto m(C[u], u) + 1] \end{array}}{\begin{array}{c} (P \uplus (t, [\![\texttt{wait}\ u]\!]_t \,; li, G), m) \Rightarrow_t \\ (P \uplus (t, li, G[\texttt{tmp1} \mapsto ?, \texttt{tmp2} \mapsto ?], m') \end{array}}$$

Figure 10. Bigger-step semantics of instrumented programs

This definition relates each currently executing instrumentation section to the corresponding original program instruction.

Now we need to characterize the circumstances in which steps of an execution can be reordered. Steps by one thread have no effect on the state or local environment of other threads; the only way in which they communicate is via their effects on the shared memory. As such, the main obligation in proving that we can reorder steps in an execution is to show that reordering the associated memory operations does not change the behavior of the program. It suffices to show the stronger condition that if two instrumentation sections execute simultaneously, then the memory locations that they access do not overlap. We refer to this property as *noninterference*. In the following, we use $(P, m) \rightarrow_t^* R$ to mean that there is a (possibly empty) sequence of steps $(P, m) \rightarrow_t (P_1, m_1) \rightarrow_t \ldots \rightarrow_t R$, and $(P, m) \rightarrow_{\neg t}^* R$ to mean that there is a (possibly empty) sequence of steps $(P, m) \rightarrow_a (P_1, m_1) \rightarrow_b \ldots \rightarrow_z R$ where $a, b, \ldots, z \neq t$.

The core of noninterference is the fact that while an instrumentation section is executing, the memory locations it accesses are protected from access by other threads, by either a lock or the innate mechanics of thread creation.

Lemma 4. *Let P_0 be a well-formed uninstrumented state and P'_0 its instrumented version. Suppose we have some state P'_1 and instruction i such that*

$$(P'_0, m_0) \rightarrow^* (P'_1, m_1) \text{ where } P'_1(t) = [\![i]\!]_t; li \text{ and}$$

$$(P'_1, m_1) \rightarrow_t (P_2, m_2) \rightarrow^* (P_3, m_3) \rightarrow (P_4, m_4)$$

such that $P_3(t) = i'; \ldots; li$. Then the locations accessed by t from m_1 to m_4 do not overlap with the locations accessed by threads other than t from m_1 to m_4.

Proof. By case analysis on i, using the guarantees provided by locking and spawn-wait synchronization. $\square$

We can then prove our main noninterference lemma.

Lemma 5 (Noninterference)**.** *Let P_0 be a well-formed uninstrumented state and P its instrumented version. Suppose that $(P, m) \rightarrow_t^* (P_1, m_1) \rightarrow_{\neg t}^* (P_2, m_2) \rightarrow_t (P_3, m_3)$ such that P_2 is an instrumented suffix of P_0. Then the operation performed to reach m_3 does not overlap with the locations accessed from m_1 to m_2.*

Proof. By induction on the derivation of $(P_1, m_1) \rightarrow_{\neg t}^* (P_2, m_2)$, using Lemma 4. $\square$

We call two memory states *similar*, written $m_1 \sim m_2$, if they allow the same values to be read at all non-metadata locations. This similarity is preserved by reordering operations to unrelated locations, which allows us to use Lemma 5 to reorder steps and obtain similar memories. In an execution in which every thread terminates, we can reorder the entire execution into successful sections:

Lemma 6. *Let P_0 be a well-formed uninstrumented state and P its instrumented version. If $(P, m) \rightarrow^* (P', m')$ and P' is a final state, then $(P, m) \Rightarrow^* (P', m'')$ and $m'' \sim m'$.*

We must also consider the case in which the instrumented program detects a race, and so fails an assertion before reaching a final state. In this case, other threads may be in the midst of instrumentation sections when the execution terminates. In order to show that the instrumented program is nonetheless in the same state (up to metadata) as the original program, we must use a slightly more sophisticated approach. The core of the approach, diagrammed in Figure 11, is a confluence property stating that in an execution in which an instrumentation section has an effect on non-metadata

$$\begin{tikzcd}
(P, m) \arrow[r, "*"] \arrow[drr, Rightarrow, dotted, "i"', "t"] & \cdot \arrow[r, "i", "t"'] & \cdot \arrow[r, "*"] & \cdot \arrow[r, dotted, "*"] & (P', m') \\
 & & \cdot \arrow[urr, dotted, "*"'] & &
\end{tikzcd}$$

Figure 11. Confluence diagram for completing partially executed instrumentation block for the source instruction i.

state, 1) we can reorder the section into an atomic block, and 2) executing all the remaining steps after the atomic block yields the same state as completing the incomplete section in the starting execution.

Lemma 7. *Let P_0 be a well-formed uninstrumented state and P its instrumented version. Suppose $(P, m) \rightarrow^* (P_1, m_1) \rightarrow_t (P_2, m_2) \rightarrow^* (P_3, m_3)$, where P_1 is an instrumented suffix of P_0 and the step from P_1 to P_2 affects non-metadata state. Then there exist P', m', P'_3, m'_3 such that $(P, m) \Rightarrow_t (P', m') \rightarrow^* (P'_3, m'_3)$, $m'_3 \sim m_3$, and $(P_3, m_3) \rightarrow_t^* (P'_3, m'_3)$ by only executing metadata operations.*

Proof. By noninterference and the confluence properties of the step relation. $\square$

This lets us reorder an execution in which a race is detected into a sequence of successful instrumentation sections followed by a failing section:

Lemma 8. *Let P_0 be a well-formed uninstrumented state and P its instrumented version. If $(P, m) \rightarrow^* (P', m') \rightarrow err$, then $(P, m) \Rightarrow^* (P', m'')$ such that $m'' \sim m'$ and $(P', m'') \Rightarrow err$.*

5.2 Simulation

Lemmas 3, 6, and 8 let us reason entirely in terms of the bigger-step relation in which instrumentation executes atomically. The correctness of the instrumentation now becomes a matter of simulation: we need only prove that there is a bisimulation between states of the uninstrumented program and the instrumented program such that they mirror each other's behavior. First we define the relationship between states of the abstract algorithm and memory configurations.

Definition 5. *A block b in a memory m* encodes *a vector clock V if for all $t \leq z$, the value at (b, t) in m is equal to $V(t)$. A* VectorClock *state $\sigma = (C, L, R, W)$ is encoded by a memory m, written $m \models (C, L, R, W)$, if $C[t]$ encodes C_t for every t, $L[m]$ encodes L_m for every m, and $R[x]$ and $W[x]$ encode R_x and W_x respectively for every x.*

Definition 6. *Uninstrumented state P relates to instrumented state P', written $P \sim P'$, if the same thread ids exist in P and P', and for each pair of corresponding threads $(t, li, G), (t, li', G')$, $li' = [\![li]\!]_t$ and $G'(a) = G(a)$ for all non-*`tmp` *variables a. Say $(P, m) \sim (P', m')$ if $P \sim P'$ and $m \sim m'$.*

For each of the two directions of bisimulation, we must consider the case in which the original program does not

When $m' \models \sigma$ and $\sigma \overset{o}{\Rightarrow} \sigma_1$

$$\begin{tikzcd}(P,m) \arrow[r, "o", "t"'] \arrow[d, dash, "\sim"'] & (P_1,m_1) \arrow[d, dash, dotted, "\sim"] \\ (P',m') \arrow[r, Rightarrow, dotted, "t"'] & (P_1',m_1')\end{tikzcd}$$

such that $m_1' \models \sigma_1$.

When $m' \models \sigma$ and $\sigma \overset{o}{\not\Rightarrow}$

$$\begin{tikzcd}(P,m) \arrow[r, "o", "t"'] \arrow[d, dash, "\sim"'] & (P_1,m_1) \\ (P',m') \arrow[r, Rightarrow, dotted, "t"'] & err\end{tikzcd}$$

When $m' \models \sigma$

$$\begin{tikzcd}(P,m) \arrow[r, dotted, "o", "t"'] \arrow[d, dash, "\sim"'] & (P_1,m_1) \arrow[d, dash, dotted, "\sim"] \\ (P',m') \arrow[r, Rightarrow, "t"'] & (P_1',m_1')\end{tikzcd}$$

s.t. $\sigma \overset{o}{\Rightarrow} \sigma_1$ and $m_1' \models \sigma_1$.

When $m' \models \sigma$

$$\begin{tikzcd}(P,m) \arrow[r, dotted, "o", "t"'] \arrow[d, dash, "\sim"'] & (P_1,m_1) \\ (P',m') \arrow[r, Rightarrow, "t"'] & err\end{tikzcd}$$

such that $\sigma \overset{o}{\not\Rightarrow}$

Figure 12. Bisimulation lemmas needed to prove Theorems 4 and 5. Solid arrows denote assumptions that imply the existence of the dotted arrows, assuming that P is a well-formed initial state.

race (and the instrumented program matches its behavior), and the case in which it does race (and the instrumented program fails an assertion). The lemmas summarizing these relationships are illustrated in Figure 12.

Lemma 9 (Bisimulations)**.** *Let P be a well-formed state*

- *If $(P,m) \sim (P',m')$, $m' \models \sigma$, $(P,m) \xrightarrow{o}_t (P_2,m_2)$, and $\sigma \overset{o}{\Rightarrow} \sigma'$, then $(P',m') \Rightarrow_t (P_2',m_2')$ such that $(P_2,m_2) \sim (P_2',m_2')$ and $m_2' \models \sigma'$.*
- *If $(P,m) \sim (P',m')$, $m' \models \sigma$, $(P,m) \xrightarrow{o}_t (P_2,m_2)$, and $\sigma \overset{o}{\not\Rightarrow}$, then $(P',m') \Rightarrow_t err$.*
- *If $(P,m) \sim (P',m')$, $m' \models \sigma$, and $(P',m') \Rightarrow_t (P_2',m_2')$, then $(P,m) \xrightarrow{o}_t (P_2,m_2)$ such that $(P_2,m_2) \sim (P_2',m_2')$, $\sigma \overset{o}{\Rightarrow} \sigma'$, and $m_2' \models \sigma'$.*
- *If $(P,m) \sim (P',m')$, $m' \models \sigma$, and $(P,m) \Rightarrow_t err$, then $(P,m) \xrightarrow{o}_t (P_2,m_2)$ and $s \overset{o}{\not\Rightarrow}$.*

Proof. Each lemma is proved, by induction on the execution, case analysis on the next instruction to execute, and computation showing that the instrumentation performs exactly the operations of the VectorClock algorithm. □

The overall correctness of the instrumentation is expressed by the following theorems:

Theorem 4 (Race-free bisimulation)**.** *For all well-formed programs prog,*

$$((0, \mathsf{instrument}(prog), G_0), m_0) \rightarrow^* (P_f', m_f')$$
for some final state P_f'
iff
$$((0, prog, G_0), m_0) \xrightarrow{\alpha}{}^* (P_f, m_f)$$

where α is race-free, and $(P_f,m_f) \sim (P_f',m_f')$.

Theorem 5 (Race-detected bisimulation)**.** *For all well-formed programs prog,*

$$((0, \mathsf{instrument}(prog), G_0), m_0) \rightarrow^* (P_1', m_1') \rightarrow_t err$$
iff
$$((0, prog, G_0), m_0) \xrightarrow{\alpha}{}^* (P_1, m_1) \xrightarrow{o}_t (P_2, G_2)$$

where $(P_1,m_1) \sim (P_1',m_1')$, and, according to the VectorClock *semantics of Figure 1, $\sigma_0 \overset{\alpha}{\Rightarrow} \sigma$ and $\sigma \overset{o}{\not\Rightarrow}$.*

Since $(P,m) \sim (P',m')$ implies that the memory and local environments agree on all locations except those involved in the instrumentation, these are strong correctness properties. Theorem 4 guarantees that for each race-free execution of the original program, there is a successful instrumented execution that produces the same values in the environment and memory (and vice versa). Theorem 5 guarantees that for each racy execution of the original program, there is a corresponding instrumented execution that produces the same values in the environment and memory up until the first race, then fails (and vice versa). While it is difficult to talk about "the same" execution across two different programs, these lemmas guarantee that in terms of observable results, the original and instrumented programs have the same behavior modulo race detection, and that all racy executions are successfully detected.

6. Coq Formalization

All the theorems stated in this paper have been formally verified in Coq. The size of the definitions and proofs in the verification is shown in Table 1. The proofs varied considerably in complexity. The proof of correctness for FastTrack by bisimulation with VectorClock is particularly simple, consisting mainly of proving that various inequalities are preserved by vector clock operations; this arithmetic could probably be automated, decreasing the burden of verifying related algorithms. The proofs of correctness of the instrumentation were much more complicated, even in proportion to the definitions; the bulk of the work involved reasoning about noninterference and reordering. Our proofs do not make use of any particularly advanced features of Coq, and many of them were completed by one of the authors with no prior Coq experience.

7. Related Work

There are a few examples of verified program instrumentation from the literature, such as the implementation of the SoftBound system for enforcing memory safety in the Vellvm framework [19] and the RockSalt system for software fault isolation on x86 [12]. Neither of these systems support instrumentation of multithreaded code, which is the main source of complexity in our verification. Sadowski et al. [15] partially verified an algorithm for atomicity analysis in Coq,

	definitions (loc)	proofs (loc)
VECTORCLOCK	100	1150
FASTTRACK	70	800
multi-join VC (Δ)	30	100
instrumentation	780	18900

Table 1. Size of definitions and proofs in lines of code.

using its semantics on traces of events and statements of the invariants of the analysis, but have not yet extended it to a verified instrumentation pass.

There is a rich history of systems that provide dynamic data race detection, dating back to the original proposals of the vector-clock algorithm [4, 7, 11]. Subsequent systems have implemented vector clocks, with various optimizations, using dynamic analysis to provide race detection for C/C++ [14, 16] and Java [2, 3, 5, 18] programs. Notably, the proof of correctness of SLIMSTATE [18] uses a similar approach to our bisimulation proof of FASTTRACK, relating its behavior to a simpler algorithm (in this case FASTTRACK itself). Before our work, none of these systems had been mechanically verified. Furthermore, these paper proofs operate at a high level of abstraction, using an operational semantics that elides important details such as the synchronization used within the race detector itself. Thus, the implementations of these algorithms are far removed from the algorithms themselves, leaving the door open for bugs.

8. Conclusions and Future Work

We have presented the first machine-verified proofs of correctness of the VECTORCLOCK and FASTTRACK race detection algorithms and a race detection instrumentation pass for a simple multithreaded language. The proofs provide a strong connection between the instrumentation and the abstract algorithm, and ensure that the instrumented program has the same behavior as the original program. Our verification efforts have revealed the need for a revised lemma in the original paper proof of correctness for FASTTRACK, and brought up a small discrepancy between the FASTTRACK algorithm and its implementation. Our work places dynamic data race detection on a formally verified foundation for the first time, and our reordering-based approach to verifying the instrumentation pass should be useful in verifying other kinds of instrumentation on multithreaded programs.

References

[1] H.-J. Boehm and S. V. Adve. Foundations of the C++ Concurrency Memory Model. PLDI '08, pages 68–78, New York, NY, USA, 2008. ACM.

[2] M. Christiaens and K. D. Bosschere. TRaDe: Data Race Detection for Java. ICCS '01, pages 761–770, London, UK, UK, 2001. Springer-Verlag.

[3] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware Java runtime. PLDI '07, pages 245–255, June 2007.

[4] C. Fidge. Logical Time in Distributed Computing Systems. *Computer*, 24(8):28–33, Aug. 1991.

[5] C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. PLDI '09, pages 121–133, New York, NY, USA, 2009. ACM.

[6] A. Itzkovitz, A. Schuster, and O. Zeev-Ben-Mordehai. Toward Integration of Data Race Detection in DSM Systems. *J. Parallel Distrib. Comput.*, 59(2):180–203, Nov. 1999.

[7] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.

[8] X. Leroy and S. Blazy. Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations. *J. Autom. Reason.*, 41:1–31, July 2008.

[9] W. Mansky, D. Garbuzov, and S. Zdancewic. An Axiomatic Specification for Sequential Memory Models. CAV '15, pages 413–428, 2015.

[10] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. POPL '05, pages 378–391, New York, NY, USA, 2005. ACM.

[11] F. Mattern. Virtual time and global states of distributed systems. Parallel and Distributed Algorithms, pages 215–226, 1989.

[12] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. RockSalt: Better, Faster, Stronger SFI for the x86. PLDI '12, pages 395–404, New York, NY, USA, 2012. ACM.

[13] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. PLDI '09, pages 245–258, New York, NY, USA, 2009. ACM.

[14] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. PPoPP '03, pages 179–190, New York, NY, USA, 2003. ACM.

[15] C. Sadowski, J. Yi, K. Knowles, and C. Flanagan. Proving correctness of a dynamic atomicity analysis in Coq. Workshop on Mechanizing Metatheory '08, 2008.

[16] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer: Data Race Detection in Practice. WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM.

[17] The Coq Development Team. The Coq Proof Assistant Reference Manual (Version 8.5), 2016. URL `https://coq.inria.fr/refman/`.

[18] J. Wilcox, P. Finch, C. Flanagan, and S. N. Freund. Array Shadow State Compression for Precise Dynamic Race Detection. ASE '15, 2015.

[19] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. POPL '12, pages 427–440, New York, NY, USA, 2012. ACM.