Verifying Dynamic Race Detection

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 is often used to 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 lay out an instrumentation pass for race detection in a simple language, and present 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. During the verification process, we discovered issues in both the paper proof and the implementation of the FASTTRACK race detection algorithm.

1. Introduction

Multicore processors have steadily invaded a broad swathe 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 [**?**] and in some cases undefined [**?**] 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 [**? ? ?**] 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, instead of the implementations, and they are paper proofs instead of machine-checked proofs. Because of these shortcomings, 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.

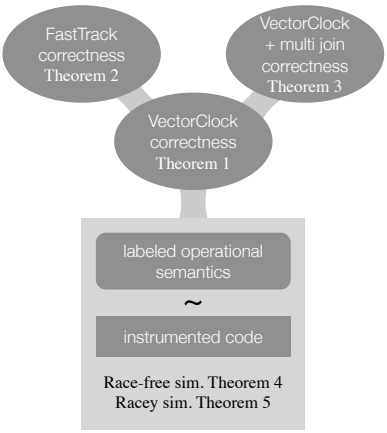

Figure 1. An overview of our work, showing verified algorithms (top) and our approach to verified instrumentation code via a labeled operational semantics.

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

We first explore the **algorithmic dimension** (ellipses in Figure 1), by formally establishing the correctness of the FASTTRACK algorithm [**?**]. 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.

[1] If this paper is accepted, we plan to submit our Coq proofs for artifact evaluation. If the reviewers are interested in having access to the Coq sources, please contact the program chair.

Our verification efforts have revealed a small issue in the paper proof that, while fixable, illustrates the potential dangers that stem from best-effort proofs. We have repaired this issue in our proof of FASTTRACK, establishing that the algorithm is correct. We also verify a variant of the base vector-clock algorithm for handling multiple joins to the same thread.

We next explore the **implementation dimension** (rectangles in Figure 1), by 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. While constructing our implementation, we consulted the existing implementation of FASTTRACK for guidance, and our verification process brought to light an example of extraneous work performed by the current FASTTRACK implementation. This issue is unlikely to have come to light otherwise, but when proving our implementation equivalent to the abstract algorithm such discrepancies were quickly revealed. This again demonstrates the value of formal verification over best-effort implementation, ensuring that an implementation does no less, and no more, than is necessary for correctness.

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 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 is critical to help 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 through to its implementation.
- We give the first verified proofs of correctness of the vector-clock and FASTTRACK data race detection algorithms.
- We give the first verified implementation of vector-clock race detection on a simple, imperative multithreaded language.
- We uncover issues in the paper proof of correctness for FASTTRACK and in its current implementation that are unlikely to have been revealed without our verification efforts. We repair these issues in our own proofs and implementation.

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 6, and evaluate our results and describe future work in Section 7.

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 allows us to separate concerns and avoid duplicating 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, e.g., vacuously correct).

Our approach to verifying instrumentation has three major steps. First, we describe our race detection algorithm abstractly, separately from the details of any programming language. We verify this algorithm against a high-level specification of the property we want to guarantee (in this case, soundness and completeness). Secondly, we define the semantics of a target language, labeled with the abstract operations produced by each step. This means that from each execution of an uninstrumented program, we can use the algorithm to determine the behavior we *would have observed* if the program was correctly instrumented. Thirdly, 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 [**?**], 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 simulation between the bigger-step semantics and "would have observed" semantics of the uninstrumented program, and a proof that the bigger-step semantics completely captures the possible behaviors of any instru-

mented program. This three-step approach has 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 [**?** **?**], which we call VECTORCLOCK, and briefly describing its verification in Coq. We then turn to the FASTTRACK [**?**] algorithm and its verification both by simulation 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 [**?**]. 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 thread t releasing a lock m, and b is thread u subsequently acquiring it; or (3) (a, b) is in the transitive closure of program order and synchronization order. Synchronization order is also commonly considered to include thread operations such as fork and join. Two events not ordered by happens-before are *concurrent*. Two memory accesses to the same address form a *data race* if they are concurrent and at least one is a write.

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 [**?** **?**]. 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 $\forall t.\ V_a(t) \leq V_b(t)$. Finally, *increment* increases a single component of a vector clock, defined as $inc_t(V) = \lambda u.\ \text{if } u = t \text{ then } V(u) + 1 \text{ else } V(u)$.

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 address x.
- Vector clock W_x stores the time of each thread's last write to address 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 2. 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. 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 $\sigma \not\xrightarrow{o}$.) 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 [**?**] (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

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

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

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

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

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

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

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

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

Figure 2. VECTORCLOCK operational semantics

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 [**?**] leverages the observation that, by the definition of a data race, all writes to an address 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 $c \leq V(t)$. $\bot_e$ denotes 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 read vector clock or 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 3 (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 Simulation

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 simulates 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 simulation 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 it emulates V for C_u for all u and L_m for all m.*

In FASTTRACK, when write metadata W_x is collapsed to an epoch $c@t$, it is precisely because $W_x(t) = c$ and $c@t$ emulates W_x. Figure 4 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 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

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

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

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

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

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

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

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

Figure 3. FASTTRACK operational semantics. The FORK, JOIN, ACQUIRE and RELEASE rules are identical to VECTORCLOCK, and omitted here for clarity.

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 4. 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]	$\bot_e$
write x	”	[1,0]	$\bot_e$	$1@t_0$

Figure 5. An example of how FASTTRACK's write epoch emulates a conventional read vector clock.

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 5 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, FASTTRACK's write epoch emulates the conventional R_x vector clock because the reads happen before t_0's write, so 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

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

some 0 values in places where reads have in fact been performed. 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 6 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 in the following definition.

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 it partially emulates V for C_u for all u and L_m for all m.

We can now state the full simulation relation between vector clock 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 $\mathcal{R}'_x = \bot_e$, 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, so by Theorem **??** the trace is race-free. Conversely, for each race-free trace, there is a successful execution of VECTORCLOCK, so there is also a successful execution of 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. We expect that the arithmetic involved could be further automated, decreasing the burden of verifying related algorithms.

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 states the following: *Suppose σ is well-formed and $\sigma \stackrel{\alpha}{\Rightarrow} \sigma'$ 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 $\sigma \stackrel{\alpha}{\Rightarrow} \sigma'$ 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 simulation. 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 simulation 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 2, 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 at the same time, the updates to C_u will race. (See Section 4.3 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 simultaneous 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 operation $exit(t)$ 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 exits, and only be the target of a $join$ after it $exit$s. With these changes, we obtain a new vector clock race detection algorithm suitable for multiple-join approaches to concurrency.

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. However, for dynamic race detection to be useful, it must be implemented as 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. For example, we found that the implementation of FASTTRACK in the RoadRunner framework uses a different version of the RELEASE rule from that shown in Figure 2: 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 (though it does represent unnecessary computation), but the fact that the algorithm implemented is different from the one described weakens the guarantee provided by the paper proof. 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 define a simple multithreaded language that is just complicated enough to have races and implement race detection instrumentation. The instructions of the language are recursively defined as follows, where n is a natural number, a is a local variable, $e, e1, e2$ are expressions, x is a memory location, l is a lock, t is a numeric thread id, and li is a list of $instr$s:

$$expr ::= n \mid a \mid e_1 + e_2 \mid \texttt{max}(e_1, e_2)$$

$$\begin{aligned} instr ::= {} & a \texttt{ := } e \mid a \texttt{ := load } x \mid \texttt{store } e\ x \mid \texttt{lock } l \\ & \mid \texttt{unlock } l \mid \texttt{spawn } t\ li \mid \texttt{wait } t \\ & \mid \texttt{assert(}e_1 \texttt{ <= } e_2\texttt{)} \end{aligned}$$

A *program* is simply a list of instructions, considered to be the body of the main thread with id 0. The dynamic state of a program contains a collection P of thread states, where a thread is a triple (t, li, ρ) of an id, a list of instructions, 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 `asserts`). 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 have no effect on the program's execution, but allow us to 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 pull out some thread and execute the next instruction in that thread, 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 an optional race detection operation.

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

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

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

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

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

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

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

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

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

Figure 7. Semantics of the simple language

The semantics of the language is shown in Figure 7. Note that we implement locks as memory locations in which we store 0 if the lock is unheld 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

[2] While our semantics is phrased in terms of memory states and updates, the Coq implementation uses an approach based on that of Mansky et al. [**?**], 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.

$\mathsf{move}(p, q) \triangleq$
```
tmp1 := load p;
store tmp1 q;
```

$\mathsf{set}(a, b) \triangleq$ // $V := V'$
```
move((a,0),(b,0));
...
move((a,z-1),(b,z-1));
```

$\mathsf{inc}(b, o) \triangleq$ // $C = C[t := inc_t(C_t)]$
```
tmp1 := load (b,o);
tmp1 := tmp1 + 1;
store tmp1 (b,o);
```

$\mathsf{max}(p, q) \triangleq$
```
tmp1 := load p;
tmp2 := load q;
tmp2 := max tmp1 tmp2;
store tmp2 p;
```

$\mathsf{merge}(a, b) \triangleq$ // $C = C \sqcup C'$
```
max((a,0),(b,0));
...
max((a,z-1),(b,z-1));
```

$\mathsf{lea}(p, q) \triangleq$
```
tmp1 := load p;
tmp2 := load q;
assert(tmp1 <= tmp2);
```

$\mathsf{hb\_check}(a, b) \triangleq$ // $C \sqsubseteq C'$
```
lea((a,0),(b,0));
...
lea((a,z-1),(b,z-1));
```

Figure 8. Helper macros

have executed to completion; *err* is also considered final. A *result* of a program *prog* is a final state R such that $((0, prog, G_0), m_0) \xrightarrow{\vec{\sigma}}{}^* R$; depending on the order in which threads are interleaved, a program (even a race-free one) may have many possible results.

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 8. We designate two local variables unused in the base program, `tmp1` and `tmp2`, as temporaries for use by the instrumentation. Let z be 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 be dynamically resized in memory when this bound is exceeded.)

$[\![a$ `:= load` $x]\!]_t \triangleq$
```
hb_check(W[x],C[t]);
move((C[t],t),(R[b],t));
a := load x;
```

$[\![$`store` $e\ x]\!]_t \triangleq$
```
hb_check(W[x],C[t]);
hb_check(R[b],C[t]);
move((C[t],t),(W[x],t));
store e x;
```

$[\![$`lock` $m]\!]_t \triangleq$
```
lock m;
merge(L[m],C[t]);
```

$[\![$`unlock` $m]\!]_t \triangleq$
```
set(C[t],L[m]);
inc(t,C[t]);
unlock m;
```

$[\![$`spawn` $u\ li]\!]_t \triangleq$
```
merge(C[t],C[u]);
inc(t,C[t]);
spawn u ([[li]]_u);
```

$[\![$`wait` $u]\!]_t \triangleq$
```
wait u;
merge(C[u],C[t]);
inc(u,C[u]);
```

Figure 9. Instrumentation

With these macros as building blocks, we can straightforwardly translate the rules of VECTORCLOCK from Figure 2 into code. For this purpose, we use a block-offset memory model; 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 timing 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 a program *prog* is then simply $\mathsf{instrument}(prog) \triangleq [\![prog]\!]_0$, the result of applying the instrumentation function to each instruction in *prog*.

4.3 Necessary Synchronization

The instrumentation of the previous section cannot implement sound and complete race detection for one important reason: when a race does occur, the corresponding instrumentation also races. In instrumented programs such as that

//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 10. Races in instrumentation

shown in Figure 10, depending on the order in which the updates to metadata locations occur, 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, locks prevent races on their associated metadata, and a thread cannot race with the thread that spawns it or waits for it to terminate.) 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. We then add locking to the `load` and `store` instrumentation:

$$
\begin{aligned}
[\![a \texttt{ := load } x]\!]_t \triangleq\ & \texttt{lock } X[x]; \\
& \texttt{hb\_check}(W[x], C[t]); \\
& \texttt{move}((C[t], t), (R[x], t)); \\
& a \texttt{ := load } x; \\
& \texttt{unlock } X[x]; \\
[\![\texttt{store } e\ x]\!]_t \triangleq\ & \texttt{lock } X[x]; \\
& \texttt{hb\_check}(W[x], C[t]); \\
& \texttt{hb\_check}(R[x], C[t]); \\
& \texttt{move}((C[t], t), (W[x], t)); \\
& \texttt{store } e\ x; \\
& \texttt{unlock } X[x];
\end{aligned}
$$

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 verify the correctness of 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 simulation 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 simulation 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 11. 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 some checks 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 application of the relevant small-step rules. □

We would also like to prove the correspondence in the other direction, but this is much more difficult. 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 in the process of executing as many different instrumentation sections as there are threads. We resolve this problem by showing that we can "reorder" 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.

The reordering proceeds inductively: given an execution, we can always find the first complete instrumentation section, move its steps to the front of the execution, and then continue on the remaining steps. 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 a state P_0 if P_0 and P contain the same threads and for each thread t, $P(t)$ is obtained by dropping some prefix*

[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.

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

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

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

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

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

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

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

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

Figure 11. Bigger-step semantics of instrumented programs

of the instrumentation of the first instruction from $[\![P_0(t)]\!]_t$ (or is empty if $P_0(t)$ is empty).

Steps by a 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. For our purposes, 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 $(P, m) \rightarrow_t (P_1, m_1) \rightarrow_t ... \rightarrow_t R$, and $(P, m) \rightarrow_{\neg t}^* R$ to mean that there is a (possibly empty) sequence of steps $(P, m) \rightarrow_a (P_1, m_1) \rightarrow_b ... \rightarrow_z R$ where $a, b, ..., z \neq t$.

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 counterpart. 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 $P_3(t) = i'; ...; li$. 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 the instruction i. In the cases of assignment and `assert` statements, no locations are accessed. For `load` and `store` instructions on a location x, the lock $X[x]$ protects all associated memory locations, and so no other thread can access them until the lock is released. For `lock` and `unlock` instructions on a lock m, m itself protects its associated metadata, and likewise no other thread can access until the instrumentation is finished and the lock released. For `spawn` instructions, the thread is not spawned until the end of the instrumentation, and so no other thread can interact with it or its metadata. Likewise, for `wait` instructions, the instrumentation cannot begin to execute until the target of the `wait` has terminated, and at that point no other thread will access its metadata (we assume that only one thread `wait`s for each thread; see the note in Section 3.4). □

We can then prove our main noninterference lemma.

Lemma 5 (Noninterference)**.** *Let P_0 be a well-formed uninstrumented state and P its instrumented counterpart. 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 $(P_1, m_1) \rightarrow_{\neg t}^* (P_2, m_2)$. Since P_2 is an instrumented suffix of P_0, we know that no instrumentation section is completed in this section of the execution. Thus, each step by a thread $u \neq t$ is either the first step of an instrumentation section, or somewhere in the middle of an instrumentation section. In the former case, we know from Lemma 3 that the remaining operations by threads other than u do not overlap with the operations by u.

In the latter case, there must have previously existed a first step by u that began executing the instrumentation section, and again we know from Lemma 3 that all remaining operations by threads other than u do not overlap with operations by u. We can conclude that the operations by each thread from m_1 to m_3 are to disjoint locations. Since none of the operations from m_1 to m_2 are by t, the desired result follows immediately. □

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 4 to reorder steps and obtain similar memories.

Lemma 6. *Let P_0 be a well-formed uninstrumented state and P its instrumented counterpart. Suppose that $(P, m) \to^* (P_1, m_1) \to_t (P_2, m_2)$, where P_1 is an instrumented suffix of P_0 and the step from P_1 to P_2 completes an instrumentation section. Then there exists a state P' such that $(P, m) \Rightarrow_t (P', m') \to^* (P_2, m_2')$ and $m_2' \sim m_2$.*

Proof. By induction on the derivation of $(P, m) \to^* (P_1, m_1)$. We use Lemma 4 to justify moving each step by t before all steps by threads other than t. This gives us an execution of the form $(P, m) \to_t^* (P', m') \to^* (P_2, m_2')$ in which the steps from P to P' execute a complete instrumentation section (and $m_2' \sim m_2$). This allows us to conclude that $(P, m) \Rightarrow_t (P', m') \to^* (P_2, m_2')$. □

This lemma allows us to reorder the first completed instrumentation section to the front of an execution. The next lemma allows us to identify the first completed instrumentation section if one exists, and forms the basis for all our reordering reasoning henceforth.

Lemma 7. *Let P_0 be a well-formed uninstrumented state and P its instrumented counterpart. If $(P, m) \to^* (P', m')$, then either P' is an instrumented suffix of P_0, or there are some P_1 and t such that $(P, m) \Rightarrow_t (P_1, m_1) \to^* (P', m'')$ and $m'' \sim m'$.*

Proof. By induction on the derivation of $(P, m) \to^* (P', m')$. In particular, we must consider the case in which P' is an instrumented suffix of P_0, but steps to some P_2' that is not an instrumented suffix of P_0. In this case, the thread t that advanced in this step must have completed the instrumentation section for some instruction. By Lemma 5, we can reorder the steps by t from P to P_2 to the front, and obtain a P_1 such that $(P, m) \Rightarrow_t (P_1, m_1) \to^* (P_2', m'')$ as desired. □

We can then inductively apply this reordering to all the completed instrumentation sections in an execution.

Lemma 8. *Let P_0 be a well-formed uninstrumented state and P its instrumented counterpart. If $(P, m) \to^* (P', m')$, then there is some uninstrumented state P_1 such that P' is an instrumented suffix of P_1, $(P, m) \Rightarrow^* (P_1', m_1') \to^* (P', m'')$, and $m'' \sim m'$.*

Proof. By induction on the size of P. From Lemma 6, we know that either P' is an instrumented suffix of P_0 or we can reorder some instrumentation to the front of the execution. In the former case, we can choose $P_1 = P_0$ and the rest follows immediately. In the latter case, $(P, m) \Rightarrow (P_2, m_2) \to^* (P', m')$ and P_2 is smaller than P, so by the inductive hypothesis $(P_2, m_2) \Rightarrow^* (P_1', m_1') \to^* (P', m'')$ and $m'' \sim m'$. By combining the big steps, we get the desired execution. □

In an execution in which every thread terminates, we can reorder the entire execution into successful handlers:

Lemma 9. *Let P_0 be a well-formed uninstrumented state and P its instrumented counterpart. If $(P, m) \to^* (P', m')$ where P' is some final state, then $(P, m) \Rightarrow^* (P', m'')$ such that $m'' \sim m'$.*

Proof. By Lemma 7, there is some P_1 such that $(P, m) \Rightarrow^* (P_1', m_1') \to^* (P', m'')$ and P' is an instrumented suffix of P_1. But a final state is only the instrumented suffix of a final state, and so P_1 and hence P_1' must also be final. Since P_1' is final, $P_1' = P'$ and $m_1' = m''$, completing the proof. □

We must also consider the case in which the instrumented program fails an assertion before reaching a final state. In this case, other threads may be in the middle of executing instrumentation sections when the execution terminates, so the last non-error state of the instrumented program does not necessarily correspond to a state of the original program.

Lemma 10. *Let P_0 be a well-formed uninstrumented state and P its instrumented counterpart. If $(P, m) \to^* err$, then there exists some P' such that $(P, m) \Rightarrow^* (P', m') \Rightarrow err$.*

Proof. There must be some last good state P'' such that $(P, m) \to^* (P_2, m_2) \to err$. Then by Lemma 7, there is some P_1 such that P_2 is an instrumented suffix of P_1 and $(P, m) \Rightarrow^* (P_1, m_1) \to^* (P_2, m_2')$. Adding the failing step to the end of this execution yields the desired result. □

5.2 Simulation

Using Lemmas 2, 8, and 9, we can reason entirely in terms of the bigger-step relation in which instrumentation executes atomically. At this point, the correctness of the instrumentation becomes a matter of simulation: we need only prove that there is a bisimulation between states of the uninstrumented and the instrumented program such that they mirror each other's behavior. We begin by defining the relationship between states of the abstract algorithm and memory configurations.

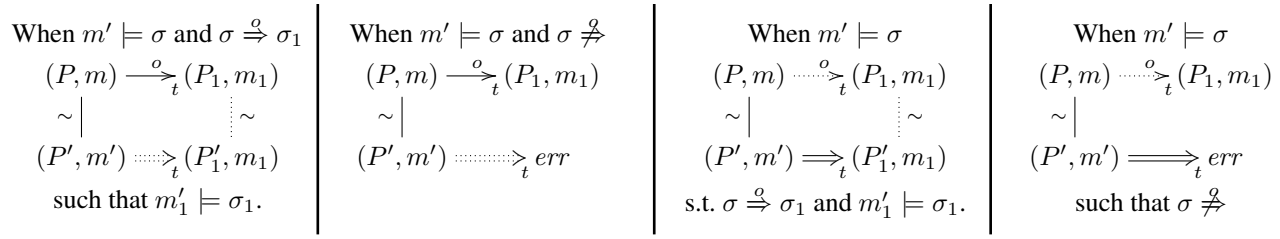

Figure 12. Bisimulation lemmas needed to prove Theorems 3 and 4. Solid arrows denote assumptions that imply the existence of the dotted arrows, assuming that P is a well-formed initial state.

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. *The relation $\sim$ relates an uninstrumented state P to an instrumented state 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. We write $(P, m) \sim (P', m')$ 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 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 given in Figure **??**.

The 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 2, $\sigma_0 \stackrel{\alpha}{\Rightarrow} \sigma$, and $\sigma \stackrel{o}{\not\Rightarrow}$.*

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 3 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 4 guarantees that for each racy execution of the original program, there is a corresponding instrumented execution that executes successfully 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. 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 [2] and the RockSalt system for software fault isolation on x86 [**?**]. Neither of these systems support instrumentation of multithreaded code, which is the main source of complexity in our verification.

There is a rich history of systems that provide dynamic data race detection, dating back to the original proposals of the vector-clock algorithm [**? ? ?**]. Subsequent systems have implemented vector clocks, with various optimizations, using dynamic analysis to provide race detection for C/C++ [**? ?**] and Java [**? ? ? ?**] programs. These systems have relied on paper proofs to demonstrate correctness, but these proofs have not been mechanically verified until our work. 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.

Lockset-based race detection [**? ?**], an alternative to the traditional vector-clock race detection algorithm, reports false races on some common programming idioms like privatization but can also detect in a single execution some races that would require multiple executions to detect with vector clocks. Other work has generalized vector-clock race detection to detect more races from a single execution [**? ? ?**] at the cost of decreased performance.

Several forms of sampling-based dynamic race detection have been proposed. Such schemes trade soundness [**? ? ? ? ?**] for reduced performance overheads.

There have been several proposals for static race detection [**? ?**] or static analysis [**? ?**] to prune race detection instrumentation and metadata at compile time, which can serve as a complement to our dynamic approach. Others have proposed type systems [**? ?**] and implicitly parallel languages [**? ?**] that eliminate data races by construction, though these systems sacrifice expressiveness to obtain race-freedom guarantees.

Several systems exist for detecting data races in structured parallel programs such as fork-join programs [**? ? ? ? ?**] or programs with asynchronous callbacks [**? ? ? ? ?**]. Structured parallelism admits more time- and space-efficient data race detection than the general multithreaded programs we support. None of these prior algorithms or implementations have been formally verified, however, so they represent a potentially fruitful area for future work.

7. Conclusions and Future Work

We have presented the first machine-verified proofs of correctness of the VECTORCLOCK and FASTTRACK race detection algorithms, as well as 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 an issue in the original paper proof of correctness for FASTTRACK, and an instance of unnecessary computation in the FASTTRACK implementation. Our work places dynamic data race detection on a formally verified foundation for the first time.

We intend to expand our approach to verified race detection along three main lines. First, our approach should generalize easily to verifying more sophisticated algorithms and instrumentation passes, such as an implementation of FASTTRACK [**?**] or the THREADSANITIZER algorithm used in LLVM [**?**]. The second and greater challenge will be to apply the same approach to more realistic languages, and ultimately to integrate it into high-assurance compilation frameworks like Vellvm [2] or CompCert [1]. This would involve taking into account a wider range of program instructions and potential complications, including variable-size accesses and low-level atomics. Finally, our proofs thus far assume sequential consistency as the concurrent memory model; an interesting challenge would be to prove that the same instrumentation suffices under the relaxed memory models used in languages like C [**?**] and Java [**?**]. By extending the reach of verified race detection, we aim to make it easier to design and justify increasingly sophisticated race detection algorithms and implementations.

References

[1] Martin Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems*, 28(2):207–255, March 2006.

[2] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 133–144, New York, NY, USA, 2004. ACM.

[3] Pavol Bielik, Veselin Raychev, and Martin Vechev. Scalable race detection for android applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 332–348, New York, NY, USA, 2015. ACM.

[4] Robert Bocchino, Mohsen Vakilian, Vikram Adve, Danny Dig, Sarita Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, and Hyojin Sung. A Type and Effect System for Deterministic Parallel Java. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications - OOPSLA '09*, page 97, Orlando, Florida, USA, 2009.

[5] Hans-J. Boehm and Sarita V. Adve. Foundations of the c++ concurrency memory model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 68–78, New York, NY, USA, 2008. ACM.

[6] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. PACER: Proportional Detection of Data Races. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation - PLDI '10*, page 255, Toronto, Ontario, Canada, 2010.

[7] Feng Chen and Grigore Rou. Parametric and Sliced Causality. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 240–253, Berlin, Heidelberg, 2007. Springer-Verlag.

[8] Mark Christiaens and Koenraad De Bosschere. TRaDe: Data Race Detection for Java. In *Proceedings of the International Conference on Computational Science-Part II*, ICCS '01, pages 761–770, London, UK, UK, 2001. Springer-Verlag.

[9] Madan Das, Gabriel Southern, and Jose Renau. Section-based program analysis to reduce overhead of detecting unsynchronized thread communication. *ACM Trans. Archit. Code Optim.*, 12(2):23:23:1–23:23:26, June 2015.

[10] Anne Dinning and Edith Schonberg. Detecting access anomalies in programs with critical sections. In *Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, PADD '91, pages 85–96, New York, NY, USA, 1991. ACM.

[11] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J. Boehm. IFRit: interference-free regions for dynamic data-race detection. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 467–

484, New York, NY, USA, 2012. ACM.

[12] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: a race and transaction-aware java runtime. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 245–255, June 2007.

[13] Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 237–252, New York, NY, USA, 2003. ACM.

[14] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[15] Colin Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, August 1991.

[16] Cormac Flanagan and Stephen N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 121–133, New York, NY, USA, 2009. ACM.

[17] Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. *Communications of the ACM*, 53(11):93–101, November 2010.

[18] Cormac Flanagan and Stephen N. Freund. RedCard: Redundant Check Elimination for Dynamic Race Detectors. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 255–280, Berlin, Heidelberg, 2013. Springer-Verlag.

[19] Joseph L. Greathouse, Zhiqiang Ma, Matthew I. Frank, Ramesh Peri, and Todd Austin. Demand-driven Software Race Detection Using Hardware Performance Counters. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 165–176, New York, NY, USA, 2011. ACM.

[20] Guy Blelloch. NESL: A Nested Data-Parallel Language. Technical Report CMU-CS-92-103, Carnegie Mellon University, Pittsburgh, PA, 1992.

[21] Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L. Pereira, Gilles A. Pokam, Peter M. Chen, and Jason Flinn. Race Detection for Event-driven Mobile Applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 326–336, New York, NY, USA, 2014. ACM.

[22] Ayal Itzkovitz, Assaf Schuster, and Oren Zeev-Ben-Mordehai. Toward integration of data race detection in dsm systems. *J. Parallel Distrib. Comput.*, 59(2):180–203, November 1999.

[23] John Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing - Supercomputing '91*, pages 24–33, Albuquerque, New Mexico, United States, 1991.

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

[25] Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, December 2009.

[26] Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal. GRace: a low-overhead mechanism for detecting data races in GPU programs. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, New York, NY, 2011.

[27] William Mansky, Dmitri Garbuzov, and Steve Zdancewic. An axiomatic specification for sequential memory models. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pages 413–428, 2015.

[28] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 378–391, New York, NY, USA, 2005. ACM.

[29] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. LiteRace: Effective Sampling for Lightweight Data-Race Detection. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation - PLDI '09*, page 134, Dublin, Ireland, 2009.

[30] Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226, 1989.

[31] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 245–258, New York, NY, USA, 2009. ACM.

[32] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 308–319, New York, NY, USA, 2006. ACM.

[33] Boris Petrov, Martin Vechev, Manu Sridharan, and Julian Dolby. Race detection for web applications. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 251–262, New York, NY, USA, 2012. ACM.

[34] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '03, pages 179–190, New York, NY, USA, 2003. ACM.

[35] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 531–542, New York, NY, USA, 2012. ACM.

[36] Veselin Raychev, Martin Vechev, and Manu Sridharan. Effective Race Detection for Event-driven Programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Appli-*

cations, OOPSLA '13, pages 151–166, New York, NY, USA, 2013. ACM.

[37] Martin C. Rinard and Monica S. Lam. The design, implementation, and evaluation of Jade. *ACM Transactions on Programming Languages and Systems*, 20(3):483–545, May 1998.

[38] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

[39] Koushik Sen, Grigore Rou, and Gul Agha. Detecting Errors in Multithreaded Programs by Generalized Predictive Analysis of Executions. In *Proceedings of the 7th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems*, FMOODS'05, pages 211–226, Berlin, Heidelberg, 2005. Springer-Verlag.

[40] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM.

[41] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. Sound predictive race detection in polynomial time. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 387–400, New York, NY, USA, 2012. ACM.

[42] The Coq Development Team. The coq proof assistant reference manual (version 8.5), 2016.

[43] James Wilcox, Parker Finch, Cormac Flanagan, and Stephen N. Freund. Array shadow state compression for precise dynamic race detection. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, 2015.

[44] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. *SIGPLAN Not.*, 47(1):427–440, January 2012.