# Verifying Dynamic Race Detection (Draft)

## 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 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.

## 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 or races, the program's behavior is unchanged.

In this paper, we seek to rectify these concerns and place dynamic race detection on a provably correct foundation for the first time. We have begun 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.

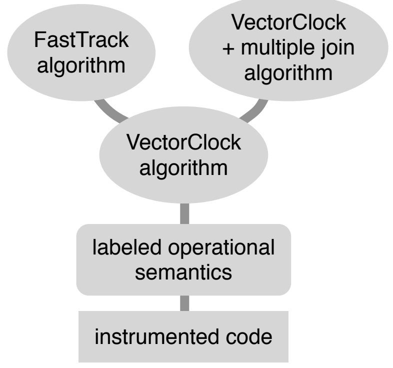
FastTrack algorithm
VectorClock + multiple join algorithm
VectorClock algorithm
labeled operational semantics
instrumented code

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

We first explore the **algorithmic dimension** (ellipses in **??**), 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. Our verification efforts have revealed a small issue in the paper proof from [**?** ] 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 next explore the **implementation dimension** (rectangles in **??**), 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 the program to perform vector clock race detection. 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 two 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. (diagram?) 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 proof 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 instrumented 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

This section reviews the classic vector-clock race detection algorithm [? ? ], and how we verified its correctness in Coq. Then we turn to the FastTrack [? ] algorithm and its verification both by simulation of the vector-clock algorithm and by direct proof.

### 3.1 Defining Data Races

Data races are defined in terms of a *happens-before* relation $<_{\text{hb}}$, 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 $a <_{\text{hb}} b$, if: (1) $a$ precedes $b$ in program order in the same thread; or (2) $a$ precedes $b$ in synchronization order $<_{\text{sw}}$, *e.g.*, if $a$ is thread $t$ releasing a lock $m$, and $b$ is thread $u$ subsequently acquiring it, then $a <_{\text{sw}} b$; 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

**[ TODO: Should "vector-clock race detection" be hyphenated everywhere?]** 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 $inc_t(V) = \lambda u.\;$ 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 $W_x$ stores the time of each thread's last write to address $x$.
- Vector clock $R_x$ stores the time of each thread's last read of address $x$ *since the last write by any thread*. If thread $t$ has not read $x$ since this write, then $R_x(t) = 0$.

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 $\forall u \neq t \,.\, C_t(u) = 0$.

The vector clock algorithm's operational semantics are presented in **??**. 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 a thread $u$'s $C_u(u)$ 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 $\sigma$ 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, where 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 unordered 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. The soundness and completeness of vector clock race detection 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 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

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

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

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

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

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

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

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

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

**Figure 2.** Conventional vector clock race detection operational semantics

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, which is equivalent to $\sqsubseteq$. We say $c@t \preceq V$ (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 $0@t_0$.

The FASTTRACK analysis state is a tuple $(C, L, R, W)$ just as in the vector clock algorithm, except that $R_x$ may be either a read vector clock or epoch, and $W_x$ 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 1. The READ and WRITE rules are split into several cases, as $R_x$ transitions between an epoch and a full vector clock. When a write occurs to a location $x$ for which $R_x$ is a vector clock, $R_x$ is set to an empty epoch $\bot_e$; conversely, when multiple unordered reads to $x$ occur, $R_x$ 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 took a different approach, proving that the transition system of FastTrack simulates that of the base vector clock algorithm. Since we have already verified the base algorithm, 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 $W_x$. **??** shows a simple program with two threads that illustrates write epoch emulation. VC $W_x$ retains information about both threads' writes, though 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 even when $R_x$ is a vector clock, it may not contain all the information about the most recent reads. The WRITE-SHARED rule resets read metadata to an empty epoch, erasing all previous values, and, as long as $R_x$ 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. **??** shows an example program illustrating this case. Assuming the location $x$ had been 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 $R_x$ metadata, losing track of the reads. However, FASTTRACK's

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

$$\textsc{ReadShared}\;\frac{\begin{array}{c} R_x \in \textit{Vector Clock} \\ W_x \preceq C_t \\ R' = R[x := R_x[t := C_t(t)]] \end{array}}{(C, L, R, W) \xRightarrow{rd(t,x)} (C, L, R', W)}$$

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

$$\textsc{ReadShare}\;\frac{\begin{array}{c} R_x \in \textit{Epoch} \\ W_x \preceq C_t \\ R_x = c@u \\ V = \bot_V[t := C_t(t), u := c] \\ R' = R[x := V] \end{array}}{(C, L, R, W) \xRightarrow{rd(t,x)} (C, L, R', W)}$$

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

$$\textsc{WriteExclusive}\;\frac{\begin{array}{c} R_x \in \textit{Epoch} \\ R_x \preceq C_t \qquad W_x \preceq C_t \\ W' = W[x := E(t)] \end{array}}{(C, L, R, W) \xRightarrow{wr(t,x)} (C, L, R, W')}$$

$$\textsc{WriteShared}\;\frac{\begin{array}{c} R_x \in \textit{Vector Clock} \\ R_x \sqsubseteq C_t \qquad W_x \preceq C_t \\ W' = W[x := E(t)] \\ R' = R[x := \bot_e] \end{array}}{(C, L, R, W) \xRightarrow{wr(t,x)} (C, L, R', W')}$$

**Figure 3.** FASTTRACK operational semantics. The FORK, JOIN, ACQUIRE and RELEASE rules are identical to the vector clock algorithm, and omitted here for clarity.

| $t_0$ | $t_1$ | VC $W_x$ | FT $W_x$ |
|---|---|---|---|
| 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 | |
|---|---|---|---|---|
| $t_0$ | $R_x$ | $W_x$ | $R_x$ | $W_x$ |
| 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.

| $t_0$ | $t_1$ | $t_2$ | VC $R_x$ | FT $R_x$ |
|---|---|---|---|---|
| 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.

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.

**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 $m$.*

The need for partial emulation arises because, even when the FASTTRACK state holds a vector clock in $R_x$, that vector clock may contain 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. **??** 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 can now state the full simulation relation between vector clock and FastTrack states.

**Definition 3.** *A vector clock state $(C, L, R, W)$ and a FastTrack state $(C', L', R', W')$ are in the relation $\sim$ when $C' = C$, $L' = L$, and for every location $x$:*

- *if $W'_x = c@t$, then $W_x(t) = c$ and $c@t$ emulates $W_x$*
- *$R'_x(t) \leq R_x(t)$ for all $t$*
- *if $R'_x = \bot_e$, then $W'_x$ emulates $R_x$*

- *if $R'_x = c@t$, then $R_x(t) = c$ and $c@t$ emulates $R_x$*
- *if $R'_x = V$, then $V$ partially emulates $R_x$*

**Theorem 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 the base algorithm on the same trace, so the trace is race-free. Conversely, for each race-free trace, there is a successful execution of the base algorithm, 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 $<_{\mathrm{hb}}$ relation. In the process, we discovered an error in the paper proof. FastTrack's Lemma 4 states the following: *Suppose $\sigma$ 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 $K^a(t) \sqsubseteq K^b(t)$.* 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 $K^a(t) \not\sqsubseteq K^b(t)$. However, the premise of Lemma 4 requires that $a$ and $b$ not be stuck, since they are in a trace $\alpha$ that successfully executes to a state $\sigma'$. 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 1.** *Suppose $\sigma$ 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 $K^a(t) \sqsubseteq K^b(u)$.*

where by abuse of notation we use $K^b(u)$ to refer to the $C_u$ component of $\sigma'$ updated as appropriate for a blocking operation. With this statement of the lemma, the completeness proof follows as outlined in the paper.

**[ TODO: SAZ: we didn't say anything specific about the Coq proofs of the simulation relation, or of the original correctness, so this comparison is a little bit baseless; we should say more about the Coq proofs somewhere. Perhaps collect all of that into a discussion section? ]** Compared to the proof by simulation, this proof is about 130 lines longer, reflecting duplication of effort from proving similar theorems about the two systems. It also involves 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 the base algorithm, the proofs have largely the same structure; 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 rules in Figure **??**, 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 in some sense unnecessary, since $u$ will perform no more operations and does not require race detection, but it appears harmless. However, if we consider what would be involved in 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 **??** 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 $\mathsf{exit}(t)$ produced when a thread terminates, and to extend the state with an additional component, a set $X$ of threads that have exited. A thread can then only perform operations before it exits, and only be the target of a $\mathsf{join}$ after it exits. With these changes, we obtain a new vector clock race detection algorithm suitable for multiple-join approaches to concurrency, and the new algorithm is also sound and complete. We have verified this modified algorithm in Coq by modifying the proofs of the base vector clock algorithm; note that since it accepts traces that are rejected by the base algorithm (in the original presentation of the vector clock algorithm, 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 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 highly error-prone **[ TODO: mention differences between FastTrack paper and implementation?]**. In this and the following sections, we present an instrumentation pass in a simple language that is formally proven to implement the simple vector clock race detection 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 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 $i_j$ are instructions:

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

$$\begin{aligned}\mathit{instr} ::= {} & a \texttt{ := } e \mid a \texttt{ := load } x \mid \texttt{store } e\ x \mid \texttt{lock } l \\ & \mid \texttt{unlock } l \mid \texttt{spawn } t\ i_1, ..., i_n \mid \texttt{wait } t \\ & \mid \texttt{assert(}e_1 \texttt{ <= } e_2\texttt{)}\end{aligned}$$

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 $(t, li, \rho)$ of an id, a list of instructions, and an environment mapping local variables to values, as well as a memory state[1] $m$. 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 any race detection operation it produces. These operations have no effect on the program's execution, but allow us to track the (purely hypothetical) behavior of the vector clock algorithm 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. The semantics of the language are shown in Figure 5.

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 $t$; 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; $err$ is also considered final.

[1] 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.

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

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

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

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

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

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

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

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

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

**Figure 7.** Semantics of the simple language

The result of a program $P$ is the configuration $R$ such that $((0, P, G_0), m_0) \xrightarrow{\vec{o}}{}^{*} R$.

### 4.2 Instrumentation

For each rule of the vector clock algorithm, we provide a code snippet that can be added to the associated instruction to implement the rule. We begin by defining macros for the basic mathematical operations of the algorithm, as shown in Figure 6. 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.)

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

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

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

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

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

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

                    // $C \sqsubseteq C'$
hb_check($a, b$)  $\triangleq$  lea($(a, 0), (b, 0)$);
                    ...
                    lea($(a, z-1), (b, z-1)$);
```

**Figure 8.** Helper macros

```
$[\![a$ := 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 vector clock race detection from Figure **??** into code. For this purpose, we use a block-offset memory model (normal variables 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.

The instrumented version of a program $P$ is then simply instrument($P$) $\triangleq [\![P]\!]_0$. Because in our language the bodies of future threads are embedded in the `spawn` instructions, we instrument these threads by recursively calling instrument on their bodies.

### 4.3 Necessary Synchronization

| | |
|---|---|
| tmp1 := load $(W[x], 0)$ | tmp1 := load $(W[x], 0)$ |
| ... | ... |
| | tmp1 := load $(R[x], 0)$ |
| ... | ... |
| store tmp1 $(R[x], 0)$ | store tmp1 $(W[x], 0)$ |
| ... | ... |
| a := load $x$ | store 2 $x$ |

**Figure 10.** Races in instrumentation

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 shown in Figure 8, 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:

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

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. **[ TODO: If this takes up too much space, we can put the locks directly into Figure 7.]**

## 5. Verifying Instrumentation

We verify the correctness of the instrumentation by showing that the instrumentation records the same information and performs the same checks as the vector clock algorithm would perform on the input program.

Key to our correctness proof is the idea that the instrumented program can be thought of as executing under a bigger-step semantics in which each instruction and its instrumentation execute in a single step. This semantics is shown in Figure **??**. 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.

**Lemma 2.** *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. $\square$

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.

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

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

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

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

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

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

**[ TODO: Which rules should we show? Also, some omitted "iterate over o"s]**

**Figure 11.** Bigger-step semantics of instrumented programs

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 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 $(P, m) \to_t^* R$ to mean that there is a (possibly empty) sequence of steps $(P, m) \to_t (P_1, m_1) \to_t \dots \to_t R$, and $(P, m) \to_{\neg t}^* R$ to mean that there is a (possibly empty) sequence of steps $(P, m) \to_a (P_1, m_1) \to_b \dots \to_z R$ where $a, b, \dots, z \neq t$.

Before proving noninterference, we first need a technical lemma that says

**Lemma 3.** *Let $P_0$ be a well-formed uninstrumented program and $P_0'$ its instrumented counterpart. Suppose we have some state $P_1'$ and instruction $i$ such that*

$$(P_0', m_0) \to^* (P_1', m_1) \text{ where } P_1'(t) = [\![i]\!]_t; li \text{ and}$$

$$(P_1', m_1) \xrightarrow{o1}_t (P_2, m_2) \xrightarrow{\vec{b}}{}^* (P_3, m_3) \xrightarrow{o4}_u (P_4, m_4)$$

*such that $P_3(t) = i'; \dots; li$. 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 $m_4$.*

**[ TODO: SAZ: I adjusted the indices in the variable names of this lemma statement. We should double check that they're correct.]**

*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 $(b, o)$, the lock $X[b]$ 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 4.3). □

This lemma lets us gather up the steps by a thread $t$ that make up an instrumentation section, and reorder them into a consecutive sequence.

**Lemma 4** (Noninterference). *Let $P_0$ be a well-formed uninstrumented program and $P$ its instrumented counterpart. Suppose that $(P, m) \to_t^* (P_1, m_1) \to_{\neg t}^* (P_2, m_2) \to_t (P_3, m_3)$ 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 $(P_1, m_1) \to_{\neg t}^* (P_2, m_2)$. 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 5.** *Let $P_0$ be a well-formed uninstrumented program 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 6.** *Let $P_0$ be a well-formed uninstrumented program 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 7.** *Let $P_0$ be a well-formed uninstrumented program and $P$ its instrumented counterpart. If $(P, m) \rightarrow^* (P', m')$, then there is some uninstrumented program $P_1$ such that $P'$ is an instrumented suffix of $P_1$, $(P, m) \Rightarrow^* (P_1', m_1') \rightarrow^* (P', m'')$, 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, $(P, m) \Rightarrow (P_2, m_2) \xrightarrow{(P', m')}$ and $P_2$ is smaller than $P$, so by the inductive hypothesis $(P_2, m_2) \Rightarrow^* (P_1', m_1') \rightarrow^* (P', m'')$ 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 8.** *Let $P_0$ be a well-formed uninstrumented program and $P$ its instrumented counterpart. If $(P, m) \rightarrow^* (P', m')$ 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 $(P, m) \Rightarrow^* (P_1', m_1') \rightarrow^* (P', m'')$ 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. We begin by defining a relation in which an instrumentation section fails in one step, as shown in Figure **[ TODO: ?]**. Then we can prove the following lemma:

**Lemma 9.** *Let $P_0$ be a well-formed uninstrumented program and $P$ its instrumented counterpart. If $(P, m) \rightarrow^*$ 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 $(P, m) \rightarrow^* (P_2, m_2) \rightarrow$ *err*. Then by Lemma 7, there is some $P_1$ such that $P_2$ is an instrumented suffix of $P_1$ and $(P, m) \Rightarrow^* (P_1, m_1) \rightarrow^* (P_2, m_2')$. Adding the failing step to the end of this execution yields the desired result. □

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 relation relating states of the uninstrumented and the instrumented program such that they mirror each other's behavior. 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). We begin by defining 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 vector clock 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 $R[b]$ and $W[b]$ encode $R_b$ and $W_b$ respectively for every $b$.*

**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 $m \sim m'$.*

**Lemma 10.** *If $P$ is a well-formed program, $(P, m) \sim (P', m')$, $m' \models \sigma$, $(P, m) \xrightarrow{o}_t (P_2, m_2)$, 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 $m_2' \models \sigma'$.*

**Lemma 11.** *If $P$ is a well-formed program, $(P, m) \sim (P', m')$, $m' \models \sigma$, $(P, m) \xrightarrow{o}_t (P_2, m_2)$, and $\sigma \overset{o}{\not\Rightarrow}$, then $(P', m') \Rightarrow_t$ err.*

**Lemma 12.** *If $P$ is a well-formed program, $(P, m) \sim (P', m')$, $m' \models \sigma$, 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 $m_2' \models \sigma'$.*

**Lemma 13.** *If $P$ is a well-formed program, $(P, m) \sim (P', m')$, $m' \models \sigma$, and $(P, m) \Rightarrow_t$ err, then $(P, m) \xrightarrow{o}_t (P_2, m_2)$ and $s \overset{o}{\not\Rightarrow}$.*

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

**Theorem 3** (Race-free bisimulation)**.** *For all well-formed programs $P$, $(\mathsf{instrument}(P), m_0) \rightarrow^* (P_f', m_f')$ for some final state $P_f'$ iff $(P, m_0) \xrightarrow{\alpha}^* (P_f, m_f)$, $\alpha$ is race-free, and $(P_f, m_f) \sim (P_f', m_f')$.*

**Theorem 4** (Race-detected bisimulation)**.** *For all well-formed programs $P$,*

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

*where $(P_1, m_1) \sim (P_1', m_1')$, and, according to the vector clock semantics of Figure* **??**, *$\sigma_0 \overset{\alpha}{\Rightarrow} \sigma$, 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 2 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 3 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

**[ TODO: discuss verified instrumentation like SoftBound, RockSalt SFI? I don't think either of these support multithreading.]**

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 data race detection algorithm, reports false races on some common programming idioms like privatization but can also detect with 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 verified proofs of correctness of the vector clock and FastTrack race detection algorithms, and a verified implementation of vector clock race detection for a simple multithreaded language. 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. **[ TODO: Where do we mention the unnecessary work in FastTrack? We should make sure that it's called out explicitly.]** 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 on 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. Thirdly, 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] X. Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, Dec. 2009. ISSN 0168-7433. `DOI:10.1007/s10817-009-9155-4`. URL `http://dx.doi.org/10.1007/s10817-009-9155-4`.

[2] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. *SIGPLAN Not.*, 47(1):427–440, Jan. 2012. ISSN 0362-1340. `DOI:10.1145/2103621.2103709`. URL `http://doi.acm.org/10.1145/2103621.2103709`.