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 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.

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.

[TODO: figure showing these two dimensions?]

We first explore the **algorithmic dimension**, 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**, 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 bigger-step semantics is analogous to that used in SoftBound [**?**], 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 (as in SoftBound) 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

The *happens-before* relation $<_{\mathrm{hb}}$ is 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.*, a lock release $rel_t(m)$ and subsequent acquire $acq_u(m)$ implies $rel_t(m) <_{\mathrm{sw}} acq_u(m)$; or (3) (a, b) is in the transitive closure of program order and synchronization order. Two events not ordered by happens-before are *concurrent*. Two memory

accesses to the same address form a *data race* if they are concurrent and at least one is a write.

3.2 Vector-Clock Race Detection

Vector clocks can track the happens-before relation during execution [**? ?**]. A vector clock v stores one integer logical clock per thread. 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. $\forall t \,.\, v_3(t) = \max(v_1(t), v_2(t))$). *Compare* 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* is defined as $inc_t(V) = \lambda u \,.\, \text{if } u = t \text{ then } V(u) + 1 \text{ else } V(u)$

A vector-clock race detector keeps four kinds of state:

- 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 v_0, where $\forall t \,.\, v_0(t) = 0$. Each thread t's initial vector clock is C_t, where $C_t(t) = 1$ and $\forall u \neq t \,.\, C_t(u) = 0$. **[TODO: would probably be good to render these as operational semantics instead]** On a lock acquire $acq_t(m)$, 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. On a lock release $rel_t(m)$, 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. On a read $rd_t(x)$, we first 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. Otherwise, we set t's entry in R_x to t's current logical clock, $C_t(t)$. On a write $wr_t(x)$, we check if $W_x \sqsubseteq C_t$ and $R_x \sqsubseteq C_t$. If this check fails, there is a previous access to x that did not happen before this write, so there is a data race. Otherwise, we clear all last reads, setting R_x to v_0, and replace the last write with the current write, such that $W_x(t) = C_t(t)$ and $\forall u \neq t \,.\, W_x(u) = 0$.

3.2.1 Correctness

The correctness of vector clock race detection follows from the fact that the $\sqsubseteq$ relation between vector clocks precisely models the happens-before relation. Our 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 $C_t(t) \sqsubseteq C_u(t)$, that is, each thread always has a higher timestamp for itself than any other thread has for it.

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 $c@t$ is a reduced vector clock that 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. 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 $c@t \preceq V$ when $c \leq V(t)$ **[TODO: iff other entries of V are zero, right?]**, and $c@t \preceq c'@u$ when $t = u$ and $c \leq c'$. **[TODO: jld: defn doesn't handle 0@t0 ¡= 1@t0. Should specify this precisely.]** $\bot_e$ denotes a minimal epoch $0@t0$, though other minimal epochs exist, *e.g.*, $0@t1$.

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 a write epoch, for the location x. 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.

Intuitively, $C_u(t)$ records the last timestamp at which the thread t synchronized with u, and $L_m(t)$, $R_x(t)$, and $W_x(t)$ record the last timestamps at which t accessed the lock m, read from the location x, and wrote to the location x, respectively. The write metadata for a location is always an epoch; the read metadata may be either an epoch or a full vector clock, depending on whether the location is being read concurrently by multiple threads. We write *EpochPair* for the set of location metadata pairs (R, W) in which both R and W are epochs, and *EpochPlusVC* for the set of pairs in which R is a vector clock and W is an epoch. The FastTrack operational semantics are presented in Figure 1.

[TODO: are these necessary here? Would be good to avoid them if possible] We have made two small departures from the FastTrack semantics as originally presented. We extend the FastTrack WriteExclusive rule (Figure 1) to reset the read epoch $M'^R_x := \bot_e$. This extension is correct because $M'^W_x := E(t)$ and $M^R_x \preceq E(t)$. Any subsequent race-free access to x must be well-ordered with respect to M'^W_x and thus must also be well-ordered with M^R_x. Any subsequent racing access to x that is concurrent with M^R_x will be concurrent with M'^W_x as well, so the read epoch is extraneous and can be cleared. Finally, we have added a ReadSharedSameEpoch rule which is used when a thread t reads *EpochPlusVC* metadata multiple times within a given epoch. This lets us distinguish the case where *Epoch-*

PlusVC updates are necessary from that where updates are not.

3.3.1 Proof by Simulation

Beyond guaranteeing the correctness of FastTrack, we want to test the simulation approach described in Section 2 as a tool for verifying refinements of the vector clock algorithm. 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 (which directly relates the algorithm to the definition of race).

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 *encapsulation* relation:

Definition 1. *An epoch* e encapsulates *a vector clock* V *for another vector clock* V' *if* $e \preceq V'$ *implies that* $V \sqsubseteq V'$. *An epoch* e *encapsulates* V *in a state* (C, L, R, W) *if it summarizes* 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 $c@t$ encapsulates W_x. The relation for read metadata is more complicated, for one particular reason: the Write Shared rule resets read metadata to an empty epoch. 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 encapsulates the original read vector clock. Second, 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 encapsulation relation, in which one component encapsulates all the relationships on zeroed values.

Definition 2. *A vector clock* V_0 partially encapsulates *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 encapsulates a vector clock* V *in a state* (C, L, R, W) *if it partially encapsulates* 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 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* $c@t$ *encapsulates* W_x
- $R'_x(t) \leq R_x(t)$ *for all* t
- *if* $R'_x = \bot_e$, *then* W'_x *encapsulates* R_x
- *if* $R'_x = c@t$, *then* $R_x(t) = c$ *and* $c@t$ *encapsulates* R_x
- *if* $R'_x = V$, *then* V *partially encapsulates* 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. □

While the simulation relation is complicated by the encapsulation relations, 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

Flanagan and Freund justify the optimizations of FastTrack by proving that the $\sqsubseteq$ relation on vector clocks still precisely captures the happens-before relation. As a baseline for comparison, we formalized this proof in Coq as well, and discovered an error in the paper proof in the process. FastTrack's Lemma 4 states the following: *Suppose* σ *is well-formed and* $\sigma \Rightarrow^{\alpha} \sigma'$ *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 α that successfully executes to a state σ'. Because of this constraint, Lemma 4 in fact does not apply. 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* σ *is well-formed and* $\sigma \Rightarrow^{\alpha} \sigma'$ *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 $K^b(t)$ to refer to the C_u component of σ' updated as appropriate for a blocking operation. With this statement of the lemma, the completeness proof follows as outlined in the paper.

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

$$\textsc{ReadSameEpoch}\ \frac{M_x^R = E(t)}{(C,L,M) \Rightarrow^{rd(t,x)} (C,L,M)}$$

$$\textsc{ReadSharedSameEpoch}\ \frac{\begin{array}{c}M_x \in \textit{EpochPlusVC} \\ M_x^W \preceq C_t \\ M_x^R[t] = C_t(t)\end{array}}{(C,L,M) \Rightarrow^{rd(t,x)} (C,L,M)}$$

$$\textsc{ReadShared}\ \frac{\begin{array}{c}M_x \in \textit{EpochPlusVC} \\ M_x^W \preceq C_t \\ M' = M[x := (M_x^R[t := C_t(t)], M_x^W)]\end{array}}{(C,L,M) \Rightarrow^{rd(t,x)} (C,L,M')}$$

$$\textsc{ReadInflate}\ \frac{\begin{array}{c}M_x \in \textit{EpochPair} \\ M_x^W \preceq C_t \\ M_x^R = c@u \\ V = \bot_V[t := C_t(t), u := c] \\ M' = M[x := (V, M_x^W)]\end{array}}{(C,L,M) \Rightarrow^{rd(t,x)} (C,L,M')}$$

$$\textsc{ReadExclusive}\ \frac{\begin{array}{c}M_x \in \textit{EpochPair} \\ M_x^R \preceq C_t \qquad M_x^W \preceq C_t \\ M' = M[x := (E(t), M_x^W)]\end{array}}{(C,L,M) \Rightarrow^{rd(t,x)} (C,L,M')}$$

$$\textsc{WriteSameEpoch}\ \frac{M_x^W = E(t)}{(C,L,M) \Rightarrow^{wr(t,x)} (C,L,M)}$$

$$\textsc{WriteExclusive}\ \frac{\begin{array}{c}M_x \in \textit{EpochPair} \\ M_x^R \preceq C_t \qquad M_x^W \preceq C_t \\ M' = M[x := (\bot_e, E(t))]\end{array}}{(C,L,M) \Rightarrow^{wr(t,x)} (C,L,M')}$$

$$\textsc{WriteShared}\ \frac{\begin{array}{c}M_x \in \textit{EpochPlusVC} \\ M_x^R \sqsubseteq C_t \qquad M_x^W \preceq C_t \\ M' = M[x := (\bot_e, E(t))]\end{array}}{(C,L,M) \Rightarrow^{wr(t,x)} (C,L,M')}$$

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

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

$$\textsc{Acquire}\ \frac{C' = C[t := (C_t \sqcup L_m)]}{(C,L,M) \Rightarrow^{acq(t,m)} (C,L,M)}$$

$$\textsc{Release}\ \frac{\begin{array}{c}L' = L[m := C_t] \\ C' = C[t := inc_t(C_t)]\end{array}}{(C,L,M) \Rightarrow^{rel(t,m)} (C',L',M)}$$

Figure 1. FastTrack operational semantics. **[TODO: update these to R,W instead of M]**

t_0	t_1	VC W_x	FT W_x	C_0	C_1
$wr_0(x)$		[3,0]	t0@3	[3,2]	[2,4]
$rel_0(m)$				[4,2]	”
	$acq_1(m)$			”	[4,5]
	$wr_1(x)$	[3,5]	t1@5	”	”

Figure 2. An example of how FastTrack's write epoch encapsulates the conventional write vector clock.

t_0
$rd_0(x)$
$wr_0(x)$

Figure 3. An example of how FastTrack's write epoch encapsulates a conventional read vector clock.

t_0	t_1	t_2
$rd_0(x)$		
$rel_0(m)$		
	$acq_1(m)$	
	$rd_1(x)$	
		$rd_2(x)$

Figure 4. An example of how FastTrack's read vector clock partially encapsulates the conventional read vector clock.

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.

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, p is a pointer, l is a lock, t is a numeric thread id, and i_j are instructions:

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

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

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 of threads, where a thread is a pair (t, li) of an id and a list of instructions, and a family of local environments G_t, one for each thread, assigning values to local variables. We give semantics to the language via a transition system of the form $(P, G) \xrightarrow{o,c} (P', G')$, where P, P' are collections of threads, G, G' are environments, o is a race detection operation (if one is produced by the step), and c is a concurrent memory operation (likewise). We define a function eval that takes a local environment and an expression and yields the value of the expression. The semantics of the language are shown in Figure 5. **[TODO: I'm not satisfied with this presentation; can we distinguish memory and RD ops better?]**

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

Figure 5. Semantics of the simple language

A state P is *final* if all of its threads have executed to completion.

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.)

With these macros as building blocks, we can straightforwardly translate the rules of vector clock race detection **[TODO: cite figure]** into code. 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 the block at $L + m$ in memory contains the vector clock L_m. 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 $\mathsf{instrument}(P) \triangleq \mathsf{instrument}(0, P)$. 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.

[TODO: formatting]

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

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

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

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

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

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

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

Figure 6. Helper macros

$\mathsf{instrument}(t, a$ `:= load` $(b, o)) \triangleq \mathsf{hb\_check}(W+b, C+t)$; $\mathsf{move}((C+t, t), (R+b, t))$; a `:= load` (b, o)

$\mathsf{instrument}(t,$ `store` $e\ (b, o)) \triangleq \mathsf{hb\_check}(W+b, C+t)$; $\mathsf{hb\_check}(R+b, C+t)$; $\mathsf{move}((C+t, t), (W+b, t))$; `store` $e\ (b, o)$

$\mathsf{instrument}(t,$ `lock` $m) \triangleq$ `lock` m; $\mathsf{merge}(L+m, C+t)$

$\mathsf{instrument}(t,$ `unlock` $m) \triangleq \mathsf{set}(C+t, L+m)$; $\mathsf{inc}(t, C+t)$; `unlock` m

$\mathsf{instrument}(t,$ `spawn` $u\ li) \triangleq \mathsf{merge}(C+t, C+u)$; $\mathsf{inc}(t, C+t)$; `spawn` u $(\mathsf{instrument}(u, li))$

$\mathsf{instrument}(t,$ `wait` $u) \triangleq$ `wait` u; $\mathsf{merge}(C+u, C+t)$; $\mathsf{inc}(u, C+u)$

Figure 7. Instrumentation

4.3 Necessary Synchronization

`tmp1 := load` $(W+b, 0)$	`tmp1 := load` $(W+b, 0)$
...	...
	`tmp1 := load` $(R+b, 0)$
...	...
`store tmp1` $(R+b, 0)$	`store tmp1` $(W+b, 0)$
...	...
`a := load` (b, o)	`store 2` (b, o)

Figure 8. 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 $X + b$ holds the lock protecting the metadata for block b. We then add locking to the `load` and `store` instrumentation:

$$\begin{aligned}&\mathsf{instrument}(t, a \texttt{ := load } (b, o)) \triangleq \texttt{lock } X + x\texttt{;}\\ &\quad \mathsf{hb\_check}(W + b, C + t)\texttt{;}\mathsf{move}((C + t, t), (R + b, t))\texttt{;}\\ &\quad a \texttt{ := load } (b, o)\texttt{;unlock } X + x\end{aligned}$$

$$\begin{aligned}&\mathsf{instrument}(t, \texttt{store } e\ (b, o)) \triangleq \texttt{lock } X + x\texttt{;}\\ &\quad \mathsf{hb\_check}(W + b, C + t)\texttt{;}\mathsf{hb\_check}(R + b, C + t)\texttt{;}\\ &\quad \mathsf{move}((C + t, t), (W + b, t))\texttt{;store } e\ (b, o)\texttt{;unlock } X + x\end{aligned}$$

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.

The semantics of our language allow one more case in which instrumentation may race. Since a terminated thread is not removed from the state when another thread `waits` for it, two threads may `wait` for the same thread, and the subsequent race detection operations may conflict. While this is a fairly obscure case, on closer examination, it is also preventable. The merge operation only reads the vector clock of the terminated thread, and simultaneous reads to the same location do not constitute races. The only write to $C + u$ is in the inc macro, which is performed to maintain the invariant that a thread always has a higher timestamp for itself than any other thread has for it. While this simplifies reasoning, the invariant is not actually necessary for a thread that will perform no further operations, which must be the case for the target of a `wait`. If we remove the inc from the `wait` instrumentation, we get an equally correct race detection algorithm that allows for multiple `waits` on a single thread. **[TODO: outline proof?]**

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 **[TODO: ?]**

Lemma 2. *If* $(P, G) \xrightarrow{\vec{a},\vec{b}}_t (P', G')$, *then* $(P, G) \xrightarrow{\vec{a},\vec{b}}_t (P', G')$.

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 of the instrumentation of the first instruction from* $\mathsf{instrument}(t, P_0(t))$ *(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 $P \rightarrow_t^* P'$ to mean that there is a (possibly empty) sequence of steps $P \rightarrow_t P_1 \rightarrow_t \ldots \rightarrow_t P'$, and $P \rightarrow_{t'}^* P'$ to mean that there is a (possibly empty) sequence of steps $P \rightarrow_a P_1 \rightarrow_b \ldots \rightarrow_z P'$ where $a, b, \ldots, z \neq t$.

Lemma 3. *Let* P_0 *be a well-formed uninstrumented program and* P_0' *its instrumented counterpart. Suppose we have some state* P' *and instruction* i *such that* $P_0' \rightarrow^* P'$ *and* $P'(t) = \mathsf{instrument}(t, i); li$, *and* $P' \xrightarrow{o1,c1}_t P_1 \xrightarrow{\vec{a},\vec{b}}{}^* P_2 \xrightarrow{o2,c2}_u P_3$ *such that* $P_2(t) = i'; \ldots; li$. *Then the operations performed by* t *in* $c1; \vec{b}; c2$ *do not overlap with the operations performed by threads other than* t *in* $c1; \vec{b}; c2$.

Proof. By case analysis on the instruction i. In the cases of assignment and `assert` statements, no memory operations are produced. 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 $P \xrightarrow{\vec{a},\vec{b}}{}^*{}_t\ P_1 \xrightarrow{\vec{c},\vec{d}}{}^*{}_{t'}\ P_2 \xrightarrow{o,c}{}_t\ P_3$ such that P_2 is an instrumented suffix of P_0. Then c does not overlap with the operations in $\vec{d}$.*

Proof. By induction on the derivation of $P_1 \xrightarrow{\vec{c},\vec{d}}{}^*{}_{t'}\ P_2$. 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 following operations in $\vec{d}; c$ 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 operations in $\vec{d}; c$ by threads other than u do not overlap with operations by u. We can conclude that the operations by each thread in $\vec{d}; c$ are to disjoint locations. Since none of the operations in $\vec{d}$ are by t, this means that c does not overlap with any of the operations in $\vec{d}$. □

Lemma 5. *Let P_0 be a well-formed uninstrumented program and P its instrumented counterpart. Suppose that $P \xrightarrow{\vec{a},\vec{b}}{}^*\ P_1 \xrightarrow{o,c}{}_t\ P_2$, 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 $P \xRightarrow{\vec{c},\vec{d}} P' \xrightarrow{\vec{e},\vec{f}}{}^*\ P_2$ and $\vec{d}; \vec{f}$ is equivalent to $\vec{b}$.*

Proof. By induction on the derivation of $P \xrightarrow{\vec{a},\vec{b}}{}^*\ P_1$. 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 $P \xrightarrow{\vec{c},\vec{d}}{}^*{}_t\ P' \xrightarrow{\vec{e},\vec{f}}{}^*\ P_2$ in which the steps from P to P' execute a complete instrumentation section. This allows us to conclude that $P \xRightarrow{\vec{c},\vec{d}}{}_t\ P' \xrightarrow{\vec{e},\vec{f}}{}^*\ P_2$. □

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 $P \xrightarrow{\vec{a},\vec{b}}{}^*\ P'$, then either P' is an instrumented suffix of P_0, or there are some P'' and t such that $P \xRightarrow{\vec{c},\vec{d}}{}_t\ P'' \xrightarrow{\vec{e},\vec{f}}{}^*\ P'$ and $\vec{b}$ is equivalent to $\vec{d}; \vec{f}$. (What about consistency?)*

Proof. By induction on the derivation of $P \xrightarrow{\vec{a},\vec{b}}{}^*\ P'$. 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'' such that $P \xRightarrow{\vec{c},\vec{d}}{}_t\ P'' \xrightarrow{\vec{e},\vec{f}}{}^*\ P_2'$ 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 $P \xrightarrow{\vec{a},\vec{b}}{}^*\ P'$, then there is some uninstrumented program P_1 such that P' is an instrumented suffix of P_1, $P \xRightarrow{\vec{c},\vec{d}}{}^*\ P_1' \xrightarrow{\vec{e},\vec{f}}{}^*\ P'$, and $\vec{b}$ is equivalent to $\vec{d}; \vec{f}$.*

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 $\vec{c}, \vec{d} = \cdot$ and the rest follows immediately. In the latter case, $P \xRightarrow{\vec{c},\vec{d}}{}_t\ P'' \xrightarrow{\vec{e},\vec{f}}{}^*$ P' and P'' is smaller than P, so by the inductive hypothesis $P'' \xRightarrow{\vec{g},\vec{h}}{}^*\ P_1' \xrightarrow{\vec{i},\vec{j}}{}^*\ P'$. 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 $P \xrightarrow{\vec{a},\vec{b}}{}^*\ P'$ where P' is some final state, then $P \xRightarrow{\vec{c},\vec{d}}{}^*\ P'$ and $\vec{b}$ is equivalent to $\vec{d}$.*

Proof. By Lemma 7, there is some P_1 such that $P \xRightarrow{\vec{c},\vec{d}}{}^*$ $P_1' \xrightarrow{\vec{e},\vec{f}}{}^*\ P'$ 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 the tail end of small steps must be empty, 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 ?. Then we can prove the following lemma:

Lemma 9. *Let P_0 be a well-formed uninstrumented program and P its instrumented counterpart. If $P \xrightarrow{\vec{a},\vec{b}}{}^* \ err$, then there exists some P' such that $P \overset{\vec{c},\vec{d}}{\Longrightarrow}{}^* P' \overset{\vec{e},\vec{f}}{\Longrightarrow} err$.*

Proof. There must be some last good state P'' such that $P \xrightarrow{\vec{a},\vec{b}}{}^* P'' \rightarrow err$. Then by Lemma 7, there is some P_1 such that P'' is an instrumented suffix of P_1 and $P \overset{\vec{c},\vec{d}}{\Longrightarrow}{}^* P_1 \xrightarrow{\vec{e},\vec{f}}{}^* P''$. 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 (C, L, R, W) is modeled 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 configuration (P, G, m) to an instrumented configuration (P', G', m') if:*

- $P' = \mathsf{instrument}(0, P)$
- *for all t and all a other than `tmp1` and `tmp2`, $G'_t(a) = G_t(a)$*
- *m and m' hold the same values at all non-metadata locations*

Lemma 10. *If P is a well-formed program, $(P, G, m) \sim (P', G', m')$, $m' \models s$, $(P, G, m) \xrightarrow{o,c}_t (P_2, G_2, m_2)$, and $s \xrightarrow{o} s'$, then $(P', G', m') \overset{\vec{o'},\vec{c'}}{\Longrightarrow}_t (P_2', G_2', m_2')$ such that $(P_2, G_2, m_2) \sim (P_2', G_2', m_2')$ and $m_2' \models s'$.*

Lemma 11. *If P is a well-formed program, $(P, G, m) \sim (P', G', m')$, $m' \models s$, $(P, G, m) \xrightarrow{o,c}_t (P_2, G_2, m_2)$, and $s \not\xrightarrow{o}$, then $(P', G', m') \overset{\vec{o'},\vec{c'}}{\Longrightarrow}_t err$.*

Lemma 12. *If P is a well-formed program, $(P, G, m) \sim (P', G', m')$, $m' \models s$, and $(P', G', m') \overset{\vec{o},\vec{c}}{\Longrightarrow}_t (P_2', G_2', m_2')$, then $m_2' \models s'$ and $(P, G, m) \xrightarrow{o',c'}_t (P_2, G_2, m_2)$ such that $(P_2, G_2, m_2) \sim (P_2', G_2', m_2')$ and $s \xrightarrow{o'} s'$.*

Lemma 13. *If P is a well-formed program, $(P, G, m) \sim (P', G', m')$, $m \models s$, and $(P, G, m) \overset{\vec{o},\vec{c}}{\Longrightarrow}_t err$, then $(P, G, m) \xrightarrow{o',c'}_t (P_2, G_2, m_2)$ and $s \not\xrightarrow{o'}$.*

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

Theorem 2. *For all well-formed programs P, $(\mathsf{instrument}(P), G_0) \xrightarrow{\vec{a}',\vec{b}'} (P_f', G_f')$ for some final state P_f iff $(P, G_0) \xrightarrow{\vec{a},\vec{b}} (P_f', G_f')$, $\vec{a}'$ is race-free, and $(P_f, G_f, m_0; \vec{b}) \sim (P_f', G_f', m_0; \vec{b}')$.*

Theorem 3. *For all well-formed programs P, $(\mathsf{instrument}(P), G_0) \xrightarrow{\vec{a}',\vec{b}'} (P_1', G_1') \rightarrow err$ iff $(P, G_0) \xrightarrow{\vec{a},\vec{b}} (P_1, G_1) \xrightarrow{o,c}_t (P_2, G_2)$, $(P_1, G_1, m_0; \vec{b}) \sim (P_1', G_1', m_0; \vec{b}')$, $\sigma_0 \xrightarrow{\vec{a}} \sigma$, and $\sigma \not\xrightarrow{o}$.*

Since $(P, G, m) \sim (P', G', m')$ implies that G and G', and likewise m and m', 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 [**?** **?** **?** **?**], async-finish 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. 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 **[TODO: cite memory models?]**. 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.