Sound Predictive Race Detection in Polynomial Time

Yannis Smaragdakis
University of Athens and
University of Massachusetts, Amherst
smaragd@di.uoa.gr

Jacob M. Evans
University of Massachusetts, Amherst
jmevans@cs.umass.edu

Caitlin Sadowski Jaeheon Yi
Cormac Flanagan
University of California at Santa Cruz
{supertri,jaeheon,cormac}@cs.ucsc.edu

Abstract

Data races are among the most reliable indicators of programming errors in concurrent software. For at least two decades, Lamport's happens-before (HB) relation has served as the standard test for detecting races—other techniques, such as lockset-based approaches, fail to be sound, as they may falsely warn of races. This work introduces a new relation, causally-precedes (CP), which generalizes happens-before to observe more races without sacrificing soundness. Intuitively, CP tries to capture the concept of happens-before ordered events that must occur in the observed order for the program to observe the same values. What distinguishes CP from past predictive race detection approaches (which also generalize an observed execution to detect races in other plausible executions) is that CP-based race detection is both sound and of polynomial complexity.

We demonstrate that the unique aspects of CP result in practical benefit. Applying CP to real-world programs, we successfully analyze server-level applications (e.g., Apache FtpServer) and show that traces longer than in past predictive race analyses can be analyzed in mere seconds to a few minutes. For these programs, CP race detection uncovers races that are hard to detect by repeated execution and HB race detection: a single run of CP race detection produces several races not discovered by 10 separate rounds of happens-before race detection.

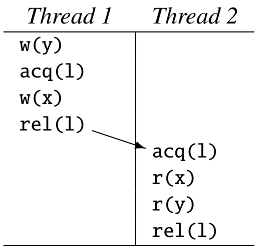
Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]; D.2.4 [*Software/Program Verification*]

General Terms Languages, Reliability, Verification

1. Introduction

Data races are the most common symptom of a programming error in the increasingly central field of concurrent programming. Two memory accesses are "conflicting" if they are performed by different threads, they access the same memory location, and at least one of them is a write. A data race is then typically defined as two *concurrent* (or *unordered* or *not happens-before-ordered*) conflicting accesses [7, 21, 26, 36].

Typical modern high-performance dynamic race detectors are based on one of two principles: *happens-before* (HB) ordering or *lockset* computation. Lockset-based race detectors follow an idea popularized by Eraser [42] and attempt to detect inconsistent use of locks for access to the same memory location by different threads. This approach has the advantage of detecting many races, even if these are not observed in the execution being monitored. For instance, if the detector records two write accesses by different threads to the same memory word without holding a common lock, it will report a possible race. The drawback of lockset-based race detectors is that they are *unsound*. The reported races are often spurious since the two suspicious events may be well-ordered via other thread communication (e.g., the prior reading of a well-synchronized flag indicating that the thread can now freely read shared data without synchronization). Consider, for instance, the example execution in Figure 1. (Our visual convention is that events occur top-to-bottom in the total order of the observed execution. We use the standard syntax `acq(l)`/`rel(l)` for the acquisition/release of lock `l`, and `w(x)`/`r(x)` for the write/read of variable `x`.)



Thread 1	*Thread 2*
w(y)	
acq(l)	
w(x)	
rel(l)	
	acq(l)
	r(x)
	r(y)
	rel(l)

Figure 1. Example of no race on variable `y`.

Although the two accesses to shared variable `y` do not occur with the same lock held, they are well-ordered—the second access only occurs after Thread 2 has observed a value written by Thread 1. It is quite possible that, if the two critical sections over lock `l` had been swapped, Thread 2 would not have attempted to read `y` since its read of `x` would have yielded a different value.

In our work, we focus on sound race detection: races are extremely hard to debug and reporting false positives to the user severely reduces the usability of a race detection tool. Sound race detection is a hallmark feature of happens-before based approaches. HB race detectors attempt to discern when there has been inter-thread communication that effectively orders the two conflicting events. Unordered events are reported as a race, since there is no reason why they could not have occurred at exactly the same instance. In its simplest form, happens-before is a partial order that generalizes the observed total order of a multithreaded program's execution by:

- ordering all events by a single thread in the order they were actually observed
- ordering lock releases and subsequent acquisitions of the same lock in the order they were observed.

POPL'12, January 25–27, 2012, Philadelphia, PA, USA.
Copyright © 2012 ACM 978-1-4503-1083-3/12/01. . . $10.00

All events unordered by happens-before are then considered to be potentially performed simultaneously. Under the assumption that threads can only communicate via mechanisms represented in the HB order[1] this approach is sound. Consider again the above execution trace. The inter-thread HB edge shown as an arrow, together with the transitivity of the HB partial order, ensure that the two writes to y are HB-ordered, thus no HB race is reported.

The problem with plain happens-before race detection is that it can miss many races due to accidental HB edges. The original definition of happens-before by Lamport [30] was in the context of distributed systems, with an HB edge introduced for explicit inter-process communication. Lock synchronization does not induce the same hard ordering as explicit communication, however. A lock-based critical section can often be reordered with others, as long as lock semantics (mutual exclusion) is preserved. Consider the code example shown in Figure 2. In this example, the PolarCoord class has two fields, radius and angle, protected by the object lock this. The count field tallies the number of accesses to radius and angle, and the main method forks two concurrent threads. This program has a race condition on count; unfortunately, precise race detectors such as FastTrack [21] or djit^{+} [38] fail to detect this race condition on 94% of test runs.

Figure 3(A) illustrates the essence of the problem by showing the trace that the HotSpot JVM typically generates for the program of Figure 2, with no overlap between the executions of the two threads. For this trace, a happens-before race detector would not find a race on count, since the lock release by Thread 1 *happens-before* the lock acquire of Thread 2, thereby masking the lack of synchronization between the accesses to count. In contrast, trace B presents a different scheduling where there is clearly a race on count. By inspection, we are able to predict from trace A that a race condition *could* occur as in trace B; we say that trace A has a *predictable* race.

Our work consists of defining a new relation, *causally-precedes* (CP), by analogy to happens-before, to detect such races. A race occurs if two conflicting actions are not CP-ordered. Unlike prior precise race detectors, a CP-based race detector can detect predictable race conditions as in Figure 3. The essence of detecting this predictable race is that the critical section of Thread 2 has received no information that can reveal whether the critical section of Thread 1 has already executed or not. More precisely, reordering events as in trace B (thus exposing an HB race) maintains the property that *all read operations return exactly the same values as in the original execution*—we call this a *correct reordering* of the observed behavior. A correctly reordered execution is just as feasible as the observed one.

Some previous work has addressed the problem of latent races in the context of happens-before race detection [10, 11, 45, 47]. This work typically comes under the label of *predictive* race analysis, and is a subset of the general area of predictive and generalizing concurrency analysis [18, 20, 28, 29, 48, 49, 57–59]. The main idea of this body of work is to consider which of the correct reorderings of critical sections would have triggered a race in an HB detector. The problem with a reorderings-based approach is that it requires exploring all reorderings of critical sections to determine which ones are correct and produce a race. This exploration is an expensive process: the space of possible reorderings is exponential and executions with races are often hard to discover. When applied to dynamic (i.e., run-time) race detection "[the] predictive runtime analysis technique can be understood as a hybrid of testing and model checking" [9].

Figure 2: Example Program PolarCoord

```
class PolarCoord {
  int radius, angle;
  int count; // counts accesses

  static PolarCoord pc = new PolarCoord();

  void setRadius(int r) {
    count++;
    synchronized(this) { radius = r; }
  }

  int getAngle() {
    int t;
    synchronized(this) { t = angle; }
    count++;
    return t;
  }

  public static void main(String[] args){
    fork { pc.setRadius(10); }
    fork { pc.getAngle(); }
  }
}
```

Figure 3: Example Traces for the PolarCoord Program.

```
Thread 1        Thread 2
r(count)
w(count)
acq(this)
w(radius)
rel(this)

                acq(this)
                r(angle)
                rel(this)
                r(count)
                w(count)
```

Tool	Report
Happens-Before:	"no race"
Causally-Precedes:	"race"

(A) predictable race condition

```
Thread 1        Thread 2
                acq(this)
                r(angle)
                rel(this)

r(count)
w(count)
                r(count)
                w(count)
acq(this)
w(radius)
rel(this)
```

Tool	Report
Happens-Before:	"race"
Causally-Precedes:	"race"

(B) happens-before race

In contrast, our work offers the first sound yet scalable technique for predictive race detection. Specifically, CP weakens the HB order while still maintaining soundness. CP is guaranteed to have a polynomial cost of evaluation, and our efficient implementation allows turning this into a linear cost, by limiting the size of the reordering window. CP race detection results are not complete: examining all correct reorderings of the original trace would necessitate an exponential search. Consequently, our detector may miss some races. Nevertheless, it is guaranteed to detect a superset of the observed happens-before races and to only give warnings for true races. (More accurately, the soundness theorem we prove is more subtle: if our detector issues a warning, there is either a correct reordering of the observed execution that exhibits an HB race, or a reordering that exhibits a deadlock. Thus, our race soundness guarantee only applies to deadlock-free programs, yet in practice our sound warning of a possible deadlock is just as valuable as a warning of a race.)

[1] Although this simplistic definition only covers locks, other inter-thread communication can be captured as happens-before edges.

Specifically our work makes the following contributions:

- We present causally-precedes, a weaker relation than happens-before, yet offering the same desirable features: CP leads to sound race detection, and can be evaluated efficiently (in polynomial time). It is worth emphasizing that multiple researchers have fruitlessly pursued such a weakening of HB in the past. We demonstrate with numerous examples why it is not easy to weaken HB while remaining sound. (Both the definition of CP and our proof of soundness are results of multi-year collaborative work, with several intermediate failed attempts.)
- We present an efficient implementation of CP. Although the relation is polynomial, practical race detection can hardly afford even quadratic complexity: an $\Omega(n^2)$ algorithm (with n being the number of observed events) is unscalable in practice, with event counts in the millions. We implement our algorithm in Datalog, for declarative logical reasoning, and apply successive optimizations, first to make the algorithm's complexity depend only on synchronization events, and then to derive a family of linear algorithms by allowing finite reordering windows.
- Our experiments showcase the advantages of CP and our implementation. The extra races detected are among the hardest to discover with plain HB analysis. A single CP-based race detection run discovers several new races, unexposed by 10 independent runs of plain HB race detection. Our implementation avoids past scalability pitfalls. If we examine races appearing within a finite reordering window (e.g., 1,000 non-redundant shared memory events) we can achieve linear runtime costs for our analysis, even for real-world programs and workloads.

The rest of the paper introduces our causally-precedes relation (Section 2), illustrates the kind of reasoning it supports and its soundness properties (Section 3), describes our implementation (Section 4), presents experimental results (Section 5), and discusses related work (Section 6) before concluding (Section 7).

2. Causally-Precedes

We next introduce our new relation, causally-precedes, and subsequently illustrate it via examples.

2.1 Definitions

We consider a standard single-observer model for assigning semantics to a concurrent execution. That is, all events are observed in a total order and we define our concepts relative to the observed order. Events have the form $[t : a]_i$, where t is the thread performing the event, and i is the event's index in the total order. The action, a performed by an event is of the form $w(x)$, $r(x)$, $acq(l)$ and $rel(l)$. Thread creation and joining can be added straightforwardly, as explicit causally-precedes edges; for simplicity, we do not discuss these events in the examples.

Definition 1 (Happens-before)**.** In this framework the happens-before ($\ll_{HB}$) relation is defined simply as the smallest relation that satisfies the following (we use underscores as a "don't care" value):

- Events by the same thread are ordered as they appear. This partial order of events in a trace is also called Program Order (PO).
  ($[t : \_]_{i_1} \ll_{HB} [t : \_]_{i_2}$ if $i_1 \leq i_2$)
- Releases and acquisitions of the same lock are ordered as they appear.
  ($[t_1 : rel(l)]_{i_{rel}} \ll_{HB} [t_2 : acq(l)]_{i_{acq}}$ if $i_{rel} < i_{acq}$)
- $\ll_{HB}$ is closed under composition with itself.
  ($\ll_{HB} = (\ll_{HB} \circ \ll_{HB})$)

We also assume two helper relations:

- the binary relation $\asymp$ (read *conflicts*). Two events by different threads conflict if they both access the same variable and one of the actions is a write.
- the function $RL(e)$ that maps a lock acquisition event to the corresponding lock release. (I.e., $RL([t_1 : acq(l)]_{i_{acq}}) = [t_1 : rel(l)]_{i_{rel}}$ if there is no $[t_1 : rel(l)]_{i'_{rel}}$ with $i_{acq} < i'_{rel} < i_{rel}$.) In this case, we say that $[t_1 : acq(l)]_{i_{acq}}$ *is paired with* $[t_1 : rel(l)]_{i_{rel}}$. $RL(e)$ is a one-to-one function for a well-formed execution, so we also consider its inverse, $RL^{-1}(e)$.[2]

Definition 2 (Causally Precedes)**.** Causally precedes ($\ll_{CP}$) is then the smallest relation such that:

a) $\ll_{CP}$ has a release-acquire edge between critical sections over the same lock that contain conflicting events.
($[t_1 : rel(l)]_{i_{rel}} \ll_{CP} [t_2 : acq(l)]_{j_{acq}}$ if there are
$[t_1 : \_]_{k_1} \asymp [t_2 : \_]_{k_2}$ such that
$i_{acq} < k_1 < i_{rel} < j_{acq} < k_2 < j_{rel}$, where
$RL^{-1}([t_1 : rel(l)]_{i_{rel}}) = [t_1 : acq(l)]_{i_{acq}}$ and
$RL([t_2 : acq(l)]_{j_{acq}}) = [t_2 : rel(l)]_{j_{rel}}$)

b) $\ll_{CP}$ has a release-acquire edge between critical sections over the same lock that contain CP-ordered events. (These events can be lock acquisition or release events, and not necessarily internal events in the critical section.)
Because of Rule (c), below, this condition turns out to be equivalent to the seemingly weaker "releases and acquisitions of the same lock are ordered if the beginning of one critical section is CP-ordered with the end of the other"—since there is an HB order between the start of a critical section and every internal event, as well as all internal events and the end of a critical section. ($[t_1 : rel(l)]_{i_{rel}} \ll_{CP} [t_2 : acq(l)]_{j_{acq}}$, if
$RL^{-1}([t_1 : rel(l)]_{i_{rel}}) \ll_{CP} RL([t_2 : acq(l)]_{j_{acq}})$)

c) CP is closed under left and right composition with HB.
($\ll_{CP} = (\ll_{HB} \circ \ll_{CP}) = (\ll_{CP} \circ \ll_{HB})$)

Note that $\ll_{CP}$ is a subset of the happens-before relation. Inspecting the three cases of the $\ll_{CP}$ definition, we see that all $\ll_{CP}$ edges produced by the first two rules are release-acquire edges on the same lock and so are also $\ll_{HB}$ edges. The third rule then states that CP is closed under composition with HB, which still produces a subset of the HB edges, since HB is transitively closed. It is similarly easy to see that CP is transitive.

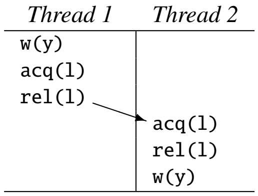
Definition 3 (CP-Race)**.** We define a *race* (or *CP-race* when we need to distinguish from happens-before races) to be a pair of conflicting events that are not CP-ordered in either direction.

2.2 Illustration

There are a few aspects of the definition of CP that should be emphasized for clarity. Probably most important among them is that CP is not a reflexive relation. (If it were, Rule (c) would make CP equal to HB.) Consequently, CP is also not a partial order.

Recall that we want CP to retain only some of the HB edges (i.e., ordering dependencies) in a way that captures which conflicting events could have happened simultaneously in a reordered but certain-to-be-feasible execution. Consider again the example of Figure 3 from Section 1 and observe that the shown HB edge is not a CP edge. None of the events shown are CP-ordered—i.e., they constitute a CP-race (and a predictable race). The same occurs in the execution of Figure 4.

[2] In this section we try to strike a balance between precise presentation and avoiding tedious definitions for pre-existing, well-understood concepts. We thus try to be more formal when defining elements unique to our approach and otherwise rely on intuition—e.g., the preconditions for a well-formed trace (such as pairing of lock acquisitions and releases, well-nesting of critical sections) are omitted here but stated fully in our proof of soundness.



Thread 1	Thread 2
w(y)	
acq(l)	
rel(l)	
	acq(l)
	rel(l)
	w(y)

Figure 4. Another example of a certain race not reported by HB.

In both cases, the critical sections do not contain conflicting events, which would order them per Rule (a) of the CP definition. In contrast, Figure 1 in Section 1 incurs no CP race report: the HB release-acquire edge is also a CP edge (per Rule (a)) and Rule (c) can then be used to CP-order the two operations on variable y.

Indeed, Rule (a) of the CP definition is almost inevitable. To see this consider what constitutes a feasible execution.

Definition 4 (Correctly Reorders)**.** We say that an execution ex' correctly reorders (CR) another execution ex (also written $ex' =_{CR} ex$) iff ex' is a total order over a subset of the events of ex that:

- contains a prefix of the events of every thread in ex and respects program order, i.e., if an event e in ex appears in ex' then all events by the same thread that precede e in ex also appear in ex' and precede e.
- for every read event that appears in ex', the most recent write event of the same variable in ex' is the same as the most recent write event of the same variable in ex.

The definition of CR matches the intuition for feasible alternative executions: if every value read is the same as in the observed execution, then the alternative is certainly also feasible.[3]

In this light, Rule (a) from the definition of CP is intuitively clear: as far as later events are concerned, two conflicting events have to occur in the same order in every correctly-reordered execution. Thus, conflicting events induce a hard ordering dependency, if it is certain that they do not constitute a race (in this case, because they are protected by a common lock). Since two critical sections over the same lock have to be ordered in their entirety (i.e., all events of one have to precede all events of the other) the ordering constraint on conflicting events becomes an ordering constraint on the entire critical section containing them. The same reasoning applies to Rule (b) of the CP definition: if two events internal to respective critical sections are CP-ordered, then the entire critical sections are also CP-ordered.

Rule (c) is the most interesting aspect of the CP definition. The rule is both very conservative and surprisingly weak, in different ways. Intuitively, Rule (c) is directly responsible for the soundness of CP: once some evidence of inevitable event ordering is found, all earlier and later events in an HB order automatically maintain their relative order. This conservative aspect of the Rule is necessary for ensuring that a CP race truly indicates that the events could have been concurrent. At the same time, Rule (c) is quite weak. Consider three events e_1, e_2, e_3. It could be that $e_1 \ll_{CP} e_2$, $e_2 \ll_{HB} e_3$, and consequently $e_1 \ll_{CP} e_3$. This still does *not* mean that $e_2 \ll_{CP} e_3$, even though the HB order between e_2 and e_3 is what allows e_1 to CP e_3. This aspect is what prevents CP from missing the possibility of predictable races (as in e_2-does-not-CP-e_3) even when it assumes conservatively that certain reorderings are not possible, in order to maintain soundness.

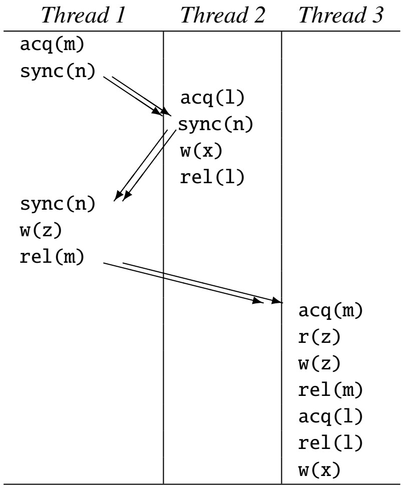
3. Complications and Soundness

We discuss the subtleties of CP through examples of hard-to-reason-about executions. Such examples naturally lead to the statement of our soundness theorem and its proof.

3.1 Example Reasoning

For each example, it is interesting for the reader to consider independently whether there is a predictable race or not. Reasoning about concurrent executions is quite hard: even concise examples require exhaustive examination of a large number of possible schedulings or complex formal reasoning to establish ordering properties. In the following examples, we expressed the constraints as symbolic inequalities with disjunctions (e.g. "this event is either before that or after the other") and proved manually they were unsatisfiable.

Figure 5 contains a first example that suggests why sound predictive race detection is hard in the presence of many threads and nested locks. We use the shorthand sync(*lock*) for a sequence of events that induces an inevitable ordering with other identical sync sequences. (E.g., sync(n) can be short for acq(n) r(nVar) w(nVar) rel(n).) For ease of reference, CP edges produced by Rule (a) of the CP definition are shown in the figure, as twin arrows.



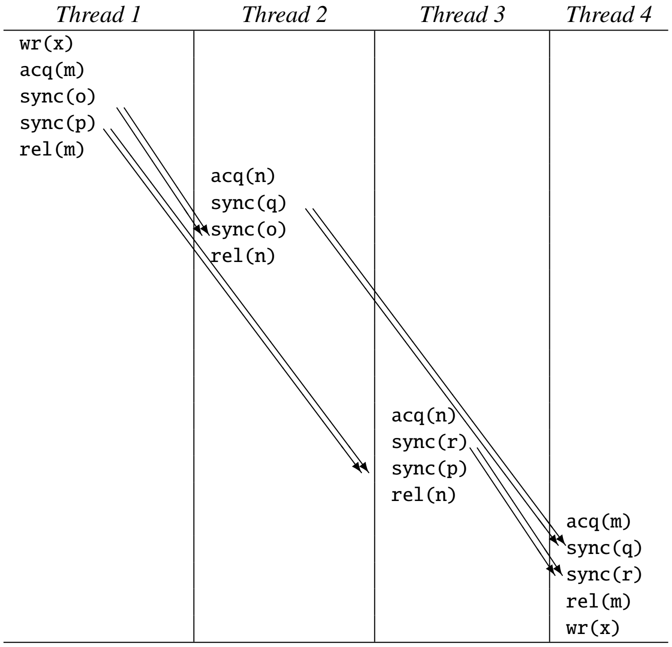
Thread 1	Thread 2	Thread 3
acq(m)		
sync(n)		
	acq(l)	
	sync(n)	
	w(x)	
	rel(l)	
sync(n)		
w(z)		
rel(m)		
		acq(m)
		r(z)
		w(z)
		rel(m)
		acq(l)
		rel(l)
		w(x)

Figure 5. No race between the two writes to x in any correctly reordered execution. Twin arrows show the "hard" CP constraints, i.e., CP order because of Rule (a) of the CP definition. sync(n) can be thought of as acq(n) r(nVar) w(nVar) rel(n).

There is no predictable race between the two writes to x in the above example: any correct reordering of the execution will have the three sync sequences and the critical sections over lock m ordered in the way they were observed, resulting in an ordering of the two writes. (Interestingly, *the empty critical section over l in Thread 3 is necessary*, or there would be a predictable race.) The CP definition captures this reasoning accurately. Rule (b) is essential in establishing that the two critical sections over lock l are CP-ordered: because of rule (b), the end of the critical section over l in Thread 2 is CP-ordered relative to the (empty) critical section over l in Thread 3. (Since CP composes with HB to yield CP, the sync(n) event in Thread 2 is CP-ordered with acq(l) in Thread 3, thus triggering rule (b).)

For another interesting example, consider Figure 6.
There is no predictable race on x in this example, but establishing this fact requires case-based reasoning involving both the hard ordering constraints induced by sync sequences and the semantics

[3] This pairing of a read with the most recent write event implicitly introduces sequential consistency as an assumption. Nevertheless, this is not a constraint: Every HB race is a CP race. In case no HB races are observed for an execution, a relaxed memory model yields sequentially consistent behavior, thus our assumption is valid.

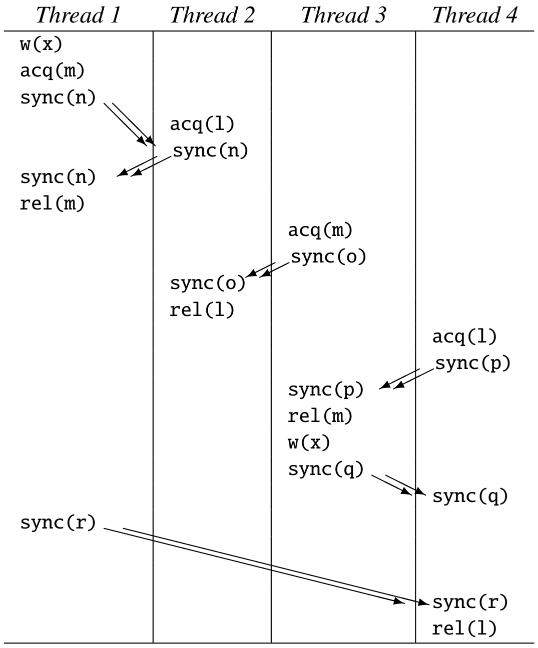


Thread 1	Thread 2	Thread 3	Thread 4
wr(x)			
acq(m)			
sync(o)			
sync(p)			
rel(m)			
	acq(n)		
	sync(q)		
	sync(o)		
	rel(n)		
		acq(n)	
		sync(r)	
		sync(p)	
		rel(n)	
			acq(m)
			sync(q)
			sync(r)
			rel(m)
			wr(x)

Figure 6. Trace with no predictable HB-race on x.

and identity of locks (e.g., the fact that the critical sections on n cannot overlap). CP avoids such reasoning but gives an accurate result. The two critical sections on n are not CP-ordered, and also do not necessarily occur in the order shown in a correctly reordered execution. The two critical sections on m, however, are CP-ordered and also necessarily in the order observed. Again, we see the interesting aspects of Rule (c) of the CP definition: even though the HB order between the critical sections on n is what enables the CP order between the critical section on m, the former does not get upgraded to a CP order.

Figure 7 presents another hard-to-reason-about example.



Thread 1	Thread 2	Thread 3	Thread 4
w(x)			
acq(m)			
sync(n)			
	acq(l)		
	sync(n)		
sync(n)			
rel(m)			
		acq(m)	
		sync(o)	
	sync(o)		
	rel(l)		
			acq(l)
			sync(p)
		sync(p)	
		rel(m)	
		w(x)	
		sync(q)	
			sync(q)
sync(r)			
			sync(r)
			rel(l)

Figure 7. No predictable race between the two writes to x.

This execution does not have a predictable race on variable x. Nevertheless, the reasoning required to establish this fact can be quite complex. Removing events can easily result in a racy execution. For instance, removing the sync(r) edge allows a race by moving the entire set of actions by Thread 3 and Thread 4 before those of Thread 1 and Thread 2.

Detecting predictable races can often require complex reorderings of events. The value of a polynomial but sound predictive race detector is that it avoids exploring all such reorderings. Consider the case of Figure 8. There is a CP-race between the two writes to variable x. There is also a predictable race. It is not possible to expose this, however, without thread scheduling that breaks up the outer critical sections, as shown on the right part of the figure. CP does not need to reproduce the schedule in order to warn of a possible race.

Thread 1	*Thread 2*
acq(m)	
sync(o)	
w(x)	
acq(n)	
rel(n)	
rel(m)	
	acq(n)
	acq(m)
	rel(m)
	w(x)
	sync(o)
	rel(n)

Thread 1	*Thread 2*
	acq(n)
	acq(m)
	rel(m)
acq(m)	
sync(o)	
w(x)	
	w(x)
	sync(o)
	rel(n)
acq(n)	
rel(n)	
rel(m)	

Figure 8. Exposing the HB race on x (execution on the left) requires a complex reordering of events (shown on the right).

At the same time, however, CP often avoids complex nested lock reasoning by not distinguishing between a predictable race and a deadlock. Our soundness theorem has an interesting form: a CP-race is a sound indication of either a race or a deadlock in a correctly reordered execution. The deadlock is immediately apparent: there is a cycle in the lock-blocking graph. To see this consider the example of Figure 9.

Thread 1	*Thread 2*
acq(m)	
acq(l)	
rel(l)	
w(x)	
rel(m)	
	acq(l)
	acq(m)
	rel(m)
	w(x)
	rel(l)

Thread 1	*Thread 2*
acq(m)	
	acq(l)
acq(l)	
	acq(m)

Figure 9. The observed execution (left) has a CP-race between the two writes to x. There is no predictable race, however! Instead, it is easy to reorder events to expose a deadlock (see right).

There is a CP-race between the accesses to variable x in this example. Yet there is no predictable HB race: no correctly reordered execution can have the two write events without synchronization between them. The CP soundness theorem states that this is only possible when a reordering can expose a deadlock due to threads acquiring locks in a way that introduces a cycle in the acquisition dependencies.

The above examples offer the reader a glimpse of the complexities of defining CP and proving its soundness. The difficulty of reasoning about event order highlights the challenge of defining a relation that weakens the observed ordering much more than happens-before, yet at the same time is guaranteed to be sound. A race analysis has to either perform heavy reasoning or to conservatively assume ordering every time events *may* be ordered. Rule (c) of the CP definition plays this role but it is still hard to prove that it is conservative enough. The ultimate conservative ordering is of course HB: all critical sections are assumed to always be precisely in the order they were observed.

3.2 Soundness

We now state more fully our assumptions on the execution form, as well as our soundness theorem. Our assumptions include the well-nesting of locks, as well as standard lock semantics.

Definition 5 (Trace)**.** A *trace* is a total order of events such that

1. Acquisition of a lock is not followed by another acquisition of the same lock without an intervening paired lock release.
2. Critical sections are well-nested. More explicitly, if an acquisition of lock l_2 is performed after an acquisition of lock l_1 by the same thread and before l_1's paired release then the paired release of l_1 cannot appear before the paired release of l_2 does.

Our main soundness theorem has a simple statement:

Theorem 1 (CP is Sound)**.** Given a trace *tr* with a CP-race, we can produce a $tr'' =_{CR} tr$ with either an HB-race or a deadlock.

The full proof can be found in the Appendix.

Note that the statement of the theorem applies only to one CP race. (And, since $\ll_{CP}$ is a subset of $\ll_{HB}$, every HB-race is also a CP-race.) That is, the theorem proof establishes that either the “first” CP-race of a trace is an HB-race in some correct reordering of the trace or we can produce a correct reordering with a deadlock. (The idea of stating the soundness guarantee so that it applies to the first error reported is standard [21, 23].) The first race is the one that *finishes* earliest in the total order of the trace, i.e., a CP-race between events $e_1 = [t_1 : u_1]_i$ and $e_2 = [t_2 : u_2]_j$, with $i < j$, such that that there is no CP-race between two events both of which appear before e_2, as well as no race between events e_3-e_2, with e_3 appearing after e_1 and before e_2.

Although the theorem's guarantee applies to only one race, we can conservatively maintain soundness when reporting multiple races, at the cost of potentially missing some. Specifically, once a CP-race (which may be merely an HB-race) is discovered (and reported), the rest of the trace can be treated as if the CP-race were a CP edge, thus hard-ordering the two racy events. This means that the soundness guarantee of the theorem then applies to the *next* CP-race reported: any correct reordering of a restricted trace (i.e., one with extra CP edges) is a correct reordering of the original trace. The drawback is that some CP-race nearby another CP race may not be reported due to our conservative treatment.

The form of the soundness theorem is quite interesting. Although a CP race implies a predictable race only in deadlock-free programs, this is hardly a disadvantage. Pointing out a potential deadlock is at least as important as pointing out races. Furthermore, deadlocks are arguably an easier problem to dynamically detect, or statically eliminate. Therefore we expect CP warnings to be almost always (correct) race warnings.

4. Implementation

CP reasoning, based on definition 2, is highly recursive. Notably, Rule (c) can feed into Rule (b), which can feed back into Rule (c). As a result, we have not implemented CP using techniques such as vector clocks, nor have we yet discovered a full CP implementation that only does online reasoning (i.e., never needs to “look back” in the execution trace); these remain challenging questions for future work. However, CP has an easy polynomial algorithm, derived directly from the definition. We express this algorithm in the Datalog language. (Datalog programs have guaranteed polynomial complexity, and Datalog can express any polynomial algorithm.) Consider a trace of execution expressed via the input relations of Figure 10. We report errors of the form “instruction X accessed memory location Y in a way that races with instruction Z”.

```
TraceNext[e1] = e2.    // what is the next event after e1
ThreadForEvent[e] = t. // what is the thread performing e
AcqEvent[e] = m.       // e is a lock acquire for mutex m
RelEvent[e] = m.       // e is a lock release for mutex m
WriteEvent[e] = x.     // e is a write for variable x
ReadEvent[e] = x.      // e is a read for variable x
```

Figure 10. Input relations for our Datalog algorithm.

(Datalog code notation: A relation in the form “`Relation[arg] = value`” is a function; the left arrow symbol, `<-`, is the implication used for inference, i.e., if the right hand side is true, the left hand side fact is inferred; `;` is the logical “or” operator.)

We can then straightforwardly define concepts such as “critical sections”, “matching lock/unlock” operations, etc. These are captured by relations “`InSameCriticalSec(e,eAcq)`” (*event `e` is in the critical section starting at `eAcq`*) and “`MatchingCriticalSecBoundary[eRel] = eAcq`” (*event `eRel` is the lock release paired with lock acquisition `eAcq`*). This allows defining HB and eventually CP as in Figure 11.

```
CP(e1Rel,e2Acq) <-
       MatchingCriticalSecBoundary[e1Rel] = e1Acq,
       RelEvent[e1Rel] = AcqEvent[e2Acq],
       InSameCriticalSec(e1,e1Acq),
       InSameCriticalSec(e2,e2Acq),
       (WriteEvent[e1] = WriteEvent[e2];
        WriteEvent[e1] = ReadEvent[e2];
        ReadEvent[e1] = WriteEvent[e2]).

CP(e1Rel,e2Acq) <-
       MatchingCriticalSecBoundary[e1Rel] = e1Acq,
       RelEvent[e1Rel] = AcqEvent[e2Acq],
       InSameCriticalSec(e1,e1Acq),
       InSameCriticalSec(e2,e2Acq),
       CP(e1,e2).

CP(e1,e2) <- CP(e1,e3), HB(e3,e2).
CP(e1,e2) <- HB(e1,e3), CP(e3,e2).
```

Figure 11. Straightforward CP logic in Datalog.

Having a polynomial algorithm is not sufficient for scalability, however. Realistic executions can have millions of “significant” shared memory events. (Significant events are those remaining after elimination of events that will not affect the reporting of a race—e.g., repeat accesses to the same shared memory variable by the same thread without intervening synchronization operations.) Defining relations such as HB or CP on a cross-product of events is prohibitive. In our implementation we address this challenge in two ways:

- We compute quadratic relations, such as HB or CP, only on lock acquisition and release events. The HB and CP order on regular memory events is then computed on-demand based on the HB and CP order of preceding/following synchronization events. This is already shown in the code of Figure 11.

- We do not fully compute HB or CP, but instead maintain a finite window K of allowed distance for event reorderings (typically 500 or 1000 significant shared memory events). `HB(e1,e2)` and `CP(e1,e2)` are guaranteed to be conservatively computed for endpoints `e1` and `e2` less than K significant events apart. This windowing strategy effectively makes our race detection often be of practically linear complexity: we only relate every event with at most K others.[4] (Our implementation does have some remaining $\Omega(n^2)$ or higher asymptotic complexity parts, but for relations that are expected to be small. A notable exception in practice is long critical sections, for which we have to relate all events to all others, thus suffering quadratic complexity.) The windowing strategy also means that our algorithm will only detect and report races between events less than K significant events apart.

Our computation of CP for only a finite window is implemented by computing a relation `FollowingNearbyEvent` and various specializations of it (for nearby events by the same thread, nearby synchronization actions, etc.). These relations are then threaded throughout the implementation to restrict computations that require examination of every pair of events. Importantly, this means that our main implementation logic differs from the simple form of Figure 11. In particular, all rules need to both have a filter (so that they trigger accurate computation only when events are within a distance of K) as well as a conservative closure (so that a sound overapproximation of CP is computed when there is a possibility of missing an ordering due to the finite window). This is well illustrated by looking at the form of the second rule of the CP definition, which is also the second rule in Figure 11. In the implementation the rule is broken up in two:

```
// the version for when CP is accurate
CP(e1Rel,e2Acq) <-
        MatchingCriticalSecBoundary[e1Rel] = e1Acq,
        RelEvent[e1Rel] = AcqEvent[e2Acq],
        InSameCriticalSec(e1,e1Acq),
        InSameCriticalSec(e2,e2Acq),
        CP(e1,e2),
        FollowingNearbySynchronizationEvent(e1,e2).

// the version for conservative CP when the
// base CP information may be inaccurate
CP(e1Rel,e2Acq) <-
        MatchingCriticalSecBoundary[e1Rel] = e1Acq,
        NearbyCriticalSecOnSameLock(e2Acq,e1Rel),
        InSameCriticalSec(e1,e1Acq),
        InSameCriticalSec(e2,e2Acq),
        SynchronizationEvent(e1),
        SynchronizationEvent(e2),
        !FollowingNearbySynchronizationEvent(e1,e2).
```

This approach allows us to maintain soundness while achieving scalability. Additional optimizations in our implementation include manual indexing for efficient relational joins, as well as join order optimizations.

5. Experiments

We evaluated the performance and prediction capability of our CP race detector on a collection of multithreaded Java benchmarks, mostly from previous studies [12, 17, 54]).

The most substantial of our benchmarks are:

- Jigsaw, W3C's web server [53], coupled with a stress test harness.
- FtpServer, a high-performance FTP server implementation from The Apache Foundation [51], coupled with a JMeter workload [52].
- StaticBucketMap, a part of the Apache Commons project, offering a thread-safe implementation of the Java Map interface. The code size of this benchmark is small, but its driver exercises it thoroughly, resulting in a long trace.

To perform the CP analysis on the benchmarks, we used the RoadRunner framework [22] to dynamically instrument the bytecode of each benchmark at load time. The instrumentation code creates a stream of events for field and array accesses, synchronization operations, thread fork/joins, etc.

We used this infrastructure to perform an inexpensive happens-before race analysis and to also produce a trace subsequently used for the CP analysis. The CP analysis was thus explicitly coded to report races if they were not also HB races (since the latter were discovered and reported already). We conservatively translate accesses to volatile variables and thread creation/join events into pseudo-lock accesses. The traces produced are quite sizable even though stack-variable references are filtered out. The traces are then reduced to only maintain events concerning shared memory locations, and to eliminate re-accesses to the same variable by the same thread without intervening synchronization. The reduced trace is then imported into a database and analyzed using our Datalog implementation.

The first columns of Table 1 show the main metrics for our benchmarks. The benchmark size in LoC is not entirely representative of its complexity for our analysis: much of the code in a program's directory is library code, not exercised at all. Conversely, much of the code actually exercised is Java library code, never shown in the benchmark size. (Library code is still valuable to analyze: the code may not contain races, but may be used in an unsafe way, exposing a race in client code.) StaticBucketMap is the most extreme example: if we were to report its code size uniformly with the other benchmarks, it would come to 110KLoC. The directory contains the entire Apache Commons Collections project, however. The main StaticBucketMap class and test driver file are just 807 LoC. Neither of the two sizes is representative of the code actually exercised, though the second is closer. Similar caveats apply to the report of thread counts. This metric lists the total number of thread created, which can be higher than the number of threads active simultaneously.

Table 1 collects the results of our experiments. We use a high-performance commercial Datalog engine by LogicBlox Inc., on academic license. All analysis was done on a machine with a Dual Six Core Intel Xeon X5650, 2.66GHz Processor and 24GB of RAM. (The CP analysis implementation is single-threaded, hence only one core was active at a time.) We performed CP analysis with the reordering window $K = 500$ significant events, as described in Section 4. We report the races found in a single HB run, in 10 HB runs, as well as the races found by CP in its single run but never found in the 10 HB detector runs. Races are reported per-variable (i.e., dynamic race instances are collapsed based on which data words they occur on). Still, multiple races may have the same underlying cause (e.g., a single missing lock/unlock may fix more than one race). Furthermore, recall that our soundness guarantee only applies to the first race and ensuring that all race reports are sound requires post-processing (which we currently perform manually and requires multiple re-runs of our analysis). For these reasons, the number of races and running times should be viewed

[4] One may argue that even an exponential search algorithm can become linear by limiting its search space to a finite window, but the constants (space and time overhead) would be prohibitive in that case. Our limiting the accurate-CP-computation window is practically feasible exactly because computing CP has a reasonable asymptotic complexity in the first place.

		Trace Size (#events)			Race Conditions Detected			
Programs	Size (LoC)	Original	Reduced	Thread Count	HB (1 run)	HB (10 runs)	CP	Time
banking	145	762	522	10	1	1	+0	10s
elevator	1.4k	25k	16k	5	0	0	+0	38s
FtpServer	39k	992k	543k	11	21	27	+7	20m 06s
hedc	25k	102k	1.4k	6	5	5	+0	9s
Jigsaw	49k	1,992k	42k	77	18	33	+3	17s
philo	86	669	382	6	0	0	+0	10s
pool1.2	8.4k	692	526	8	0	0	+0	10s
pool1.3	24k	841	683	8	0	0	+0	12s
StaticBucketMap	see text	265k	133k	5	7	7	+0	1m 35s
stringBuffer	1.4k	223	178	8	0	0	+0	9s
tsp	706	328k	381	4	0	0	+0	9s
vector	26k	325	270	15	1	1	+0	9s

Table 1. Benchmark metrics and number of races detected by our CP analysis but not found in 10 HB runs, as well as time taken for CP analysis. Running times of 9sec are effectively zero: this is the overhead of initializing the Datalog engine and importing data from text files.

only as an imperfect proxy of the utility of our approach. A likely way our analysis would be used by a programmer in practice would be not to report all races, but by analyzing a trace, fixing the first race in the code, then re-analyzing.

The results are representative of the position of our approach relative to others. As can be seen in Table 1, 8 of these benchmarks (i.e., two-out-of-three) produce tiny reduced traces (fewer than 2,000 events, with 7 of them having fewer than 1,000). These brief executions offer little opportunity to detect HB races and virtually no opportunity to correctly reorder events: For all of the tiny traces, CP does not discover any races that are not HB races. Nevertheless, a lockset-based approach would report races even for these executions, since it is easy to observe when a shared variable is accessed by a different thread with a disjoint set of locks held. This illustrates quite well how unsound, lockset-based race detectors tackle a fundamentally different problem. Unsound predictive analyses highlight *possible* race conditions, whereas our work focuses on *proving the presence* of race conditions.

Elevator is another benchmark that is mostly uninteresting. Although its trace is longer than the tiny ones, it is still fairly short (16k events), and has neither HB nor CP races. The jPredictor study [12] and the Said et al. work [40] showed no races on elevator either, despite using an unsound lockset-based analysis and an SMT solver that explores all valid trace reorderings, respectively.

The real benchmarks for our approach are the three longer traces: FtpServer, Jigsaw, and StaticBucketMap. It is instructive to compare results with HB. Clearly HB race detection and CP race detection are different in nature. The cost of our current CP analysis is at least one-to-two orders of magnitude higher than the cost of HB race detection. Nevertheless, the goal of CP is to perform a deep analysis, implicitly considering event reorderings, and to find "hard" races, which a plain HB analysis would not normally detect. For both FtpServer and Jigsaw, CP-based race detection uncovers several races that were not exposed by any of the 10 happens-before race detector runs. This ranges from 3 to 7, or an average of 17% new races for these two benchmarks, confirming that CP race detection can target deeper races than typical HB detectors. Examining the intermediate results of our analysis indicates that CP is a much weaker relation than HB: the number of CP edges computed by our analysis (among synchronization operations) is typically as low as 10% to 20% of the number of HB edges for these benchmarks. Thus, happens-before race detection often considers events to be ordered when there is no semantic reason why they should be.

StaticBucketMap is our final benchmark. It has a long execution, covering very little code, thus CP only reports races also detected by HB. This is the inverse problem from that of the tiny traces: CP does not win over HB not because of insufficient coverage but because of too-thorough coverage of the possible executions. It would be interesting in the future to perform a sensitivity analysis and create runs of several sizes, both for StaticBufferMap, as well as for the 9 programs with smaller traces, to see at what point CP starts detecting more races than HB and when HB catches up. This requires extensive knowledge of the benchmark programs, however.

The table also shows that the time required for CP analysis is quite low, although there is certainly room for further improvement of our CP implementation. For small traces, CP analysis is almost instantaneous. For traces with long critical sections, many threads, and deep lock nesting, the analysis time grows. Still, the overall scalability of CP analysis far exceeds reported numbers for past sound predictive concurrency analyses in the literature. For instance, Said et al.'s SMT-solver-based technique [40] takes 57sec to analyze a 1.4k event hedc trace. (The longest trace analyzed by Said et al. is much smaller than ours, at 45k events long. Yet even this size may be misleading, since the trace is for tsp, which performs a tiny number of thread-shared events relative to the original trace length.) We are not aware of a sound predictive analysis that can scale to executions at the level of our reactive applications Jigsaw and FtpServer with a running time in the low minutes.

6. Related Work

Our approach is distinguished from other related work by (1) maintaining precision while generalizing beyond an observed trace and (2) guaranteeing polynomial complexity and achieving scalability. We discuss other precise predictive and other dynamic approaches in some depth, and briefly overview other race detection techniques.

6.1 Predictive Approaches

The most relevant work to our precise race prediction technique [10–12, 40, 46] was briefly discussed in Section 1 and Section 5. Such work is typically a hybrid of testing and model checking and does not achieve the polynomial complexity and scalability of causally-precedes race detection. For instance, in the recent work by Said et al. [40] scalability relies on a modern SMT solver and an efficient encoding of the problem. Another interesting idea that past work [10, 11] has explored is *sliced causality*, which makes use of *a priori* control- and data-flow dependence information to obtain a reduced slice of the happens-before partial ordering for a particular observed trace. Race conditions are predicted with sliced causality by logging only the relevant operations in a program trace and model-checking all feasible trace permutations in a post-mortem analysis phase. These permutations are sound because one may infer and ignore irrelevant operations via static dependence information; the soundness of the predictions follow from the soundness of the trace permutations generated. Sliced causality requires two static analysis phases for prediction. This approach can detect races that our approach does not: recall that our correctly-reordered execution always respects intra-thread event order. However, our definition of causality is explicitly geared towards mechanisms that do not perform static program analysis, and, thus, cannot tell whether a value read by a thread influences subsequent values produced by the same thread. Adapting our causally-precedes order to also take such dependences into account (instead of assuming that every value read by a thread affects all subsequent thread actions) is orthogonal to the core properties of the definition.

Among this work, the jPredictor tool [12] is distinguished by explicitly producing a polynomial algorithm for race detection. Nevertheless, in order to do so, jPredictor abandons the general soundness guarantees of the theory that underlies it. The main soundness theorem of the jPredictor work (which applies to more than race detection) states that every produced "consistent permutation" corresponds to a possible program execution. Nevertheless, "generating all the consistent permutations of a partial order is a #P-complete problem" [12]. To avoid an exponential search, jPredictor employs two shortcuts for the case of race detection. The first is avoiding a search of permutations: events are processed following the order of the original execution. For predictive power, jPredictor relies on an unsound definition of what constitutes a race (Def. 5 of [12]). The definition adapts (to sliced causality) the lockset criterion for race detection: two conflicting, sliced causality-unordered accesses that occur without holding a common lock are considered to race. The examples of Section 3 (assuming a program text that causes the events in the order shown, under the sliced causality criterion) result in false race reports under this definition. The second shortcut is in the implementation, which performs a single slicing traversal of the trace, also resulting in unsoundness.

Another interesting predictive approach consists of reordering models that are more permissive than our correctly-reorders relation on executions. Past work [20, 27] assumes that threads only communicate via holding locks, and not by writing to shared memory locations. Any trace that holds the same locks in the same order of nesting is considered an appropriate generalization of the observed behavior in that predictive model. The Penelope [49] system then adds soundness back by trying to reproduce the predicted atomicity violation through changes to the scheduling of a real execution. The published experiment numbers imply that their approach does not scale at or near the level of CP race detection and can suffer from high costs in small yet complex executions. Nevertheless, there cannot be a valid comparison since Penelope is an atomicity violation detector and not a race detector (thus addressing an inherently harder problem). Combinations of approaches along these lines should be interesting future work.

6.2 Dynamic Analysis

Happens-Before Approaches Numerous tools are based on Lamport's happens-before relation [14, 21, 37, 43]. These tools are more precise than lockset-based race detectors, but are often less efficient. TRaDE [14] uses accordion clocks along with dynamic escape analysis to boost performance. Banerjee et al. [5] provide an alternative algorithm to the simple happens-before analysis that uses a linear amount of storage. Happens-before race detectors have also been applied to nested fork-join parallelism [32]. The DJIT$^+$ [37] algorithm is an efficient happens-before vector clock algorithm that uses the epoch optimization for a 2-3x performance improvement. FastTrack [21] improves upon DJIT$^+$ by using an adaptive lightweight representation for the happens-before relation and by introducing optimized constant-time fast paths that account for approximately 96% of operations encountered in a trace, and provides a 2.3x performance improvement over DJIT$^+$. Pacer [7] improves on the performance further by employing sampling techniques.

Lockset Approaches A *lockset* for a shared variable is the set of locks that protect access to that variable. Locksets were introduced [15] as an alternative representation to the happens-before analysis. Later, locksets were used alone as an efficient technique for data race detection in Eraser [42].

Every CP race (and hence every HB race) is also a lockset race, since the absence of a CP edge between conflicting accesses to a location means that there is no consistently held protecting lock for that location. Consequently, a lockset-based race detector would detect all races detected by CP,[5] and may also report many additional warnings. In practice, many of these extra warnings are false alarms that do not correspond to actual races.

Various refinements and extensions for Eraser have been proposed. Static escape analysis can improve performance [35, 55]. Reasoning about races at the object level instead of the memory word level [55] improves performance but leads to more false alarms. The lockset technique was also extended with timing *thread segments* [25] to reduce false positives caused by data not being protected by a lock during an initialization phase. Further performance enhancements use whole-program static analysis to reduce the amount of instrumentation necessary [13] or involve type inference [2]. Eraser's algorithm has also been extended for the Java Memory Model [31] and implemented with AspectJ [6].

Hybrid Techniques Recent work on dynamic data race detection focuses on combining locksets and happens-before analysis. O'Callahan and Choi [36] developed a two-phase localization scheme; a first pass lockset analysis filters out problematic fields for a second pass hybrid analysis. RaceTrack [61] uses a happens-before analysis to estimate whether threads can concurrently access a memory location so as to reduce false positives caused by empty locksets. MultiRace [38] presents improved versions of happens-before and lockset algorithms. Locksets enable happens-before approaches to report additional warnings and reduce the number of vector clock comparisons needed in the happens-before analysis. Goldilocks [17] combines locksets and happens-before in an unusual way by using locksets to efficiently track the happens-before relation. This precise, complicated analysis is embedded in a Java virtual machine to enable continuous monitoring of race conditions.

6.3 Other Approaches

Several *static* approaches exist to deal with data races. Type systems [1, 8, 24, 41] and languages [4] have been proposed to prevent races in programs. Other static approaches include Warlock [50] and Locksmith [39] for ANSI C programs, scalable whole-program analyses [34, 56] and dataflow-analysis-based approaches [16, 19]. Aiken and Gay [3] also investigate static race detection focusing on SPMD programs.

Scheduler-driven approaches, such as model checking [33], involve running the target program with many different thread schedules, either concretely or symbolically. RaceFuzzer [44] takes potentially racing access pairs and uses a randomized scheduler to

[5] Note that the Eraser algorithm, which incorporates lockset-based reasoning, is also slightly incomplete in how it reasons about thread-local data, and so may miss some real HB or CP races.

drive the execution to exhibit an actual race condition. It would be interesting to try to combine such techniques with our approach, which is explicitly geared towards discovering "hard" races without exploring many complex interleavings.

7. Conclusions

Precise race detectors are important tools for developing reliable multithreaded programs, while avoiding the costs associated with false alarms. We showed how to extend the traditional notion of race detection to also support *race prediction*, without sacrificing scalability or soundness. Our work introduces the novel concept of the causally-precedes relation, which significantly weakens HB. We prove that CP race detection is sound and demonstrate its practical value. Defining and proving the soundness of CP was far from trivial: both tasks became possible only after several failed attempts and significant collaborative work.

We believe that our research can open several avenues for further work. First, it is quite possible that a more efficient implementation of CP can be derived, to push performance further by at least an order of magnitude and even bring it to levels comparable to HB. Such a development may, for instance, be based on a novel summarization of CP information using vector clocks, or on the use of Binary Decision Diagrams to represent Datalog relations. (We could try the bddbddb engine [60] for this purpose.) A second possibility is that of defining other relations that weaken HB in a sound yet efficient way. Finally, we suspect the underlying notion of trace prediction, based on the causally-precedes relation, may also provide benefits when checking properties such as deadlock-freedom, atomicity, and determinism.

Acknowledgments

We would like to thank Aarti Gupta and the anonymous reviewers for their valuable comments that helped strengthen the paper. This work was funded by the National Science Foundation under grants CCF-0917774, CCF-0934631, and CCF-1116883. We thank LogicBlox Inc. for providing the platform used for our implementation and for continuous support.

References

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):207–255, 2006.

[2] R. Agarwal, A. Sasturkar, L. Wang, and S. D. Stoller. Optimized runtime race detection and atomicity checking using partial discovered types. In *International Conference on Automated Software Engineering (ASE)*, 2005.

[3] A. Aiken and D. Gay. Barrier inference. In *Symposium on Principles of Programming Languages (POPL)*, 1998.

[4] D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: a dialect of Java without data races. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2000.

[5] U. Banerjee, B. Bliss, Z. Ma, and P. Petersen. A theory of data race detection. In *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD)*, 2006.

[6] E. Bodden and K. Havelund. Racer: effective race detection using AspectJ. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2008.

[7] M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: proportional detection of data races. In *Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[8] C. Boyapati, R. Lee, and M. Rinard. A type system for preventing data races and deadlocks in Java programs. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2002.

[9] F. Chen and G. Roşu. Predicting concurrency errors at runtime using sliced causality. Technical Report UIUCDCS-R-2006-2965, Department of Computer Science, University of Illinois at Urbana-Champaign, 2006.

[10] F. Chen and G. Roşu. Parametric and Sliced Causality. In *Computer Aided Verification (CAV)*, 2007.

[11] F. Chen, T. F. Şerbănuţă, and G. Roşu. Effective predictive runtime analysis using sliced causality and atomicity. Technical Report UIUCDCS-R-2007-2905, University of Illinois at Urbana-Champaign, Department of Computer Science, October 2007.

[12] F. Chen, T. F. Şerbănuţă, and G. Roşu. jPredictor: a predictive runtime analysis tool for Java. In *International Conference on Software Engineering (ICSE)*, 2008.

[13] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridhara. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[14] M. Christiaens and K. D. Bosschere. TRaDe: Data Race Detection for Java. In *International Conference on Computational Science*, 2001.

[15] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. *SIGPLAN Notices*, 26(12):85–96, 1991.

[16] M. B. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *International Symposium on Foundations of Software Engineering (FSE)*, 1994.

[17] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware Java runtime. In *Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[18] M. Emmi, S. Qadeer, and Z. Rakamarić. Delay-bounded scheduling. In *Symposium on Principles of Programming Languages (POPL)*, 2011.

[19] D. R. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[20] A. Farzan, P. Madhusudan, and F. Sorrentino. Meta-analysis for atomicity violations under nested locking. In *Computer Aided Verification (CAV)*, 2009.

[21] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[22] C. Flanagan and S. N. Freund. The roadrunner dynamic analysis framework for concurrent programs. In *PASTE*, pages 1–8, 2010.

[23] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[24] D. Grossman. Type-safe multithreading in Cyclone. In *Workshop on Types in Language Design and Implementation (TLDI)*, 2003.

[25] J. J. Harrow. Runtime checking of multithreaded applications with visual threads. In *International SPIN Workshop on Model Checking of Software*, 2000.

[26] D. P. Helmbold, C. E. McDowell, and J. zhong Wang. Detecting data races by analyzing sequential traces. In *HICCS-24, Hawaii Intl. Conference on System Sciences (HICCS-24)*, 1990.

[27] V. Kahlon, F. Ivančić, and A. Gupta. Reasoning about threads communicating via locks. In *Computer Aided Verification (CAV)*, 2005.

[28] V. Kahlon and C. Wang. Universal causality graphs: A precise happens-before model for detecting bugs in concurrent programs. In *Computer Aided Verification (CAV)*, 2010.

[29] S. K. Lahiri, S. Qadeer, and Z. Rakamarić. Static and precise detection of concurrency errors in systems code using smt solvers. In *Computer Aided Verification (CAV)*, 2009.

[30] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, July 1978.

[31] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[32] J. M. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Supercomputing*, 1991.

[33] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent pro-

grams. In *Operating Systems Design and Implementation (OSDI)*, 2008.

[34] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[35] H. Nishiyama. Detecting data races using dynamic escape analysis based on read barrier. In *Virtual Machine Research and Technology Symposium (VM)*, 2004.

[36] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2003.

[37] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multihreaded C++ programs. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2003.

[38] E. Pozniansky and A. Schuster. Multirace: efficient on-the-fly data race detection in multithreaded C++ programs. *Concurrency and Computation: Practice and Experience*, 19(3):327–340, 2007.

[39] P. Pratikakis, J. S. Foster, and M. Hicks. Context-sensitive correlation analysis for detecting races. In *Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[40] M. Said, C. Wang, Z. Yang, and K. Sakallah. Generating data race witnesses by an smt-based analysis. In *NASA Formal Methods Symposium*, pages 313–327. Springer, 2011.

[41] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated type-based analysis of data races and atomicity. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2005.

[42] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multi-threaded programs. In *ACM Symposium on Operating Systems Principles (SOSP)*, 1997.

[43] E. Schonberg. On-the-fly detection of access anomalies. In *Conference on Programming Language Design and Implementation (PLDI)*, 1989.

[44] K. Sen. Race directed random testing of concurrent programs. In *Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[45] K. Sen and G. Agha. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *IIFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, 2005.

[46] K. Sen, G. Rosu, and G. Agha. Online efficient predictive safety analysis of multithreaded programs. *International Journal on Software Technology and Tools Transfer (STTT)*, 8(3):248–260, 2006.

[47] T. F. Şerbănuţă, F. Chen, and G. Roşu. Maximal causal models for multithreaded systems. Technical Report UIUCDCS-R-2008-3017, University of Illinois at Urbana-Champaign, Department of Computer Science, December 2008.

[48] N. Sinha and C. Wang. On interference abstractions. In *Symposium on Principles of Programming Languages (POPL)*, 2011.

[49] F. Sorrentino, A. Farzan, and P. Madhusudan. PENELOPE: weaving threads to expose atomicity violations. In *International Symposium on Foundations of Software Engineering (FSE)*, 2010.

[50] N. Sterling. Warlock: A static data race analysis tool. In *USENIX Winter Technical Conference*, 1993.

[51] The Apache Software Foundation. Apache FtpServer. Available at `http://mina.apache.org/ftpserver/`, 2009.

[52] The Apache Software Foundation. Apache JMeter. Available at `http://jakarta.apache.org/jmeter/`, 2009.

[53] The World Wide Web Consortium. Jigsaw Web Server. Available from `http://www.w3.org/Jigsaw/`, 2009.

[54] C. von Praun and T. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[55] C. von Praun and T. R. Gross. Object race detection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2001.

[56] J. W. Voung, R. Jhala, and S. Lerner. Relay: static race detection on millions of lines of code. In *International Symposium on Foundations of Software Engineering (FSE)*, 2007.

[57] C. Wang, S. Kundu, M. Ganai, and A. Gupta. Symbolic predictive analysis for concurrent programs. In *World Congress on Formal Methods (FM)*, 2009.

[58] C. Wang, R. Limaye, M. Ganai, and A. Gupta. Trace-based symbolic analysis for atomicity violations. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2010.

[59] L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2006.

[60] J. Whaley. *Context-Sensitive Pointer Analysis using Binary Decision Diagrams*. PhD thesis, Stanford University, Mar. 2007.

[61] Y. Yu. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

Appendix: Soundness Proof

Theorem (CP is Sound)**.** Given a trace tr with a CP-race, we can produce a $tr'' =_{CR} tr$ with either an HB-race or a deadlock.

Proof. Let the first race of trace tr be between events e_1 and e_2, with e_1 appearing before e_2 in the trace. Being the first race means that there is no CP-race between two events both of which appear before e_2, as well as no race between events e_3-e_2, with e_3 appearing after e_1 and before e_2.

Consider a trace tr' such that:

- $tr' =_{CR} tr$ and tr' has the same first CP-race as tr, i.e., between events $e_1 = [t_1 : u_1]_i$ and $e_2 = [t_2 : u_2]_j$, with $i < j$. (With i and j the indices of the events in trace tr'.)
- Among traces satisfying the above property, tr' has minimal distance $j - i$ between events e_1 and e_2 of the first CP-race. (Intuitively, this means that all irrelevant events between e_1 and e_2 are correctly-reordered out of the e_1-e_2 segment in trace tr'.)
- Among traces that satisfy the above properties, tr' is one that also minimizes the distance between e_2 and every beginning of a critical section containing e_1, from innermost to outermost. That is, among traces that have the same (minimal) distance between e_2 and the innermost acquisition event k_1 for a critical section containing e_1, tr' minimizes the distance between e_2 and the second such lock acquire, among those satisfying all the above tr' minimizes the distance between e_2 and the third such lock acquire, k_3, and so on, all the way to the outermost critical section containing e_1.

We will refer to the last two requirements as *the minimality property*. For such a trace tr', we get important lemmas:

Lemma 1. All events e between e_1 and e_2 are such that

(a) $e_1 \ll_{HB} e$ and $e \ll_{HB} e_2$
(b) $e_1 \not\ll_{CP} e$ and $e \not\ll_{CP} e_2$.

Proof. We prove each case separately.

(a) • Assume $e \not\ll_{HB} e_2$. Then we can move e and all events e' that occur between e and e_2 such that $e \ll_{HB} e'$ to the point right after e_2. The result of the move maintains program order. (Note that thread t_2 is not affected by the move at all, since we have already assumed that $e \not\ll_{HB} e_2$, therefore e cannot happen-before any previous event in t_2.) If this move (or any move in later proofs) is not possible it is because of one of two reasons:

- It causes the result to not be a trace because the pairing of lock acquisition/releases becomes invalid (i.e., a lock is acquired while held).

- The result is a valid trace t but $t \neq_{CR} tr'$ because a read now sees a different written value.

The former means that the moved events have a common lock with some non-moved event (say, f) that occurs before e_2—an impossibility since in this case $e \ll_{HB} f$, hence f would be a moved event. The latter reason is also an impossibility since then a moved event would conflict with a non-moved event, f. If the two events were CP-ordered, then they would also be HB ordered, hence f would have moved. If the events were not CP-ordered then e_1-e_2 would not have been the first race of tr'. Hence the move is possible, which violates the first minimality property, therefore our assumption was false and $e \ll_{HB} e_2$.

- If $e_1 \not\ll_{HB} e$, then we can move e and all events e' that occur between e_1 and e such that $e' \ll_{HB} e$ to right before e_1. Again, with similar reasoning as above (or as in Lemma 2, below) we get a contradiction.

(b) If either $e_1 \ll_{CP} e$ or $e \ll_{CP} e_2$, then we would have had $e_1 \ll_{CP} e_2$, per part (a) and the definition of CP: a contradiction. □

Lemma 2. Consider any lock acquire event a_1 for a critical section containing e_1. All events e between a_1 (inclusive) and e_1 have $a_1 \ll_{HB} e$ and if $e1 \ll_{HB} e2$ then $e \ll_{HB} e_2$.

Proof. Assume $a_1 \not\ll_{HB} e$. Let E be the set of operations made up of e and all e' that occur between a_1 and e such that $e' \ll_{HB} e$. Note that set E cannot contain any events from thread t_1, or else $a_1 \ll_{HB} e$. Try to move all operations in E to right before a_1. If the move is not possible, it is either because these moved events have a common lock with some non-moved event, f (a contradiction, since then $f \ll_{HB} e$, and f would be moved) or that the moved events conflict with a non-moved event (also a contradiction since it would imply a race before e_1-e_2 or a CP relation, which violates the assumption of no-HB between a_1 and the moved events). Therefore, moving E before a_1 is possible, and the result of the move maintains intra-thread order. However, moving E violates the minimality property of tr'.

The fact that ($e \ll_{HB} e_2$) follows from similar reasoning as in Lemma 1, but uses the assumption that $e1 \ll_{HB} e2$ to establish that $e1$ is not among the moved events. □

Lemma 3. Any conflicting events that both occur before e_2, or with one being e_2 and the other occurring after e_1 and before e_2, have to be CP-ordered.

Proof. Otherwise we trivially have a CP-race earlier than e_1-e_2. □

Lemma 4. Consider any lock acquire event a_1 for a critical section containing e_1. If a critical section $c...c'$ starts before event a_1 and ends after a_1 and before e_1 then $a_1 \ll_{CP} c'$.

Proof. By induction.

Base case: Consider the $c...c'$ that ends the soonest after a_1 (among all such $c...c'$ that satisfy the stated conditions). Assume that $a_1 \not\ll_{CP} c'$. By the well-nesting of lock operations, such a critical section $c...c'$ cannot be performed by thread t_1.

Let d be the first event after a_1 in this critical section. We will try to move $d...c'$ to the point right before a_1, respecting intra-thread order. If the move is successful it violates the minimality of tr', hence the move must be illegal because it violates some property of CR or of the definition of a trace. Therefore, the move must be illegal for either of the usual two reasons:

1. It causes the result to not be a trace because the pairing of lock acquisition/releases becomes invalid (i.e., a lock is acquired while held).
2. The result is a valid trace t but $t \neq_{CR} tr'$ because a read now sees a different written value.

For case (1), the moved events cannot be acquiring a lock held by thread t_1 at position a_1, since that lock would not be released before e_1. If the lock were held by a thread other than t_1, we have a critical section with the stated properties for $c...c'$ that ends before the currently considered $c...c'$, which is impossible. Case (2) means that the moved events conflict with some non-moved event that occurs after a_1. This non-moved event e'' has to CP the moved event it conflicts with (by Lemma 3). But from Lemma 2 we have $a_1 \ll_{HB} e''$ and therefore $a_1 \ll_{CP} c'$.

Inductive case: the argument is identical to the base case, except in case (1) when we consider the possibility that a lock that needs to be acquired by the moved events is held by a thread other than t_1. In this case, we have an earlier critical section $g...g'$ with the stated properties, and therefore $a_1 \ll_{CP} g'$, by the inductive assumption. But since our $c...c'$ acquires the same lock, we get the desired $a_1 \ll_{CP} c'$. □

Lemma 5. There cannot be a critical section by a thread other than t_2 that starts after event e_1 and before e_2, and ends after event e_2.

Proof. Assume that such critical sections exist. Among them pick the $c...c'$ that starts last, i.e., closest to e_2. Let d be the last event before e_2 of this critical section. We have two cases:

1. If $c...d$ does not contains nested critical sections inside it, we can move all events $c...d$ to the point right after e_2. The proof is similar to that of Lemma 4. The move respects intra-thread ordering. Also the moved events cannot be acquiring a lock held at point e_2. (There are no nested critical sections in the moved events, and the lock acquired by event c is still held at e_2.) Furthermore, the resulting trace is a correct reordering of the original because if it were not we would then have a conflict between events whose relative position changes, i.e., between the moved events and non-moved events. But in that case there would be a CP edge originating in $c...d$ to an event before e_2 (by Lemma 3) and since c is between e_1 and e_2 we would get (using Lemma 1) $e_1 \ll_{CP} e_2$ (a contradiction).
2. If $c...d$ does contain critical sections, let $g...g'$ be the one ending last before e_2. Consider an event sequence produced as follows:
   - we drop all events starting from (and including) g of that thread
   - we drop all events after e_2 by all other threads.

   Clearly the result is a prefix of tr'. If it is a legal trace that correctly reorders tr' then we are violating the minimality of tr'. In the resulting event sequence there cannot be an event acquiring a lock already held: the only dropped lock release events are either after e_2 (in which case subsequent lock acquisitions are also dropped), or are dropped together with their lock acquisition event (in the case of $g...g'$ events). Note that if there is a critical section inside $c...c'$ that starts before $g...g'$, it has to also end before g, or it would violate the definition of either $g...g'$ or $c...c'$. Also, no read can see a different write, or this would imply a conflict between a dropped event after g and another before e_2. In such a case we would have a CP ordering, per Lemma 3 and $e_1 \ll_{CP} e_2$, as before. We conclude that the resulting trace correctly reorders tr' and violates its minimality assumption: a contradiction.

□

Armed with these lemmas about trace tr' we can now attempt to prove the soundness theorem. We will show that tr' either has an HB race (in fact, e_1 and e_2 have to be adjacent in this minimal trace) or, if not, the trace exposes a deadlock which can be caused by a slightly reordered trace.

Clearly, if e_1-e_2 is an HB race in tr' then we are done. Assume it is not. We will try to CR-reorder tr' so that one of the minimality properties is violated (which is a contradiction). Consider the first event f such that:

- f is performed by a thread other than t_1
- f occurs after e_1 and before e_2.

Such an event needs to exist if e_1-e_2 are HB-ordered. Furthermore, by Lemma 1, $e_1 \ll_{HB} f$, and since f is the first such event in any thread other than t_1, it needs to be a lock acquire. Consider then the critical section $f...f'$. There are two cases:

1. f' occurs before e_2. Let $f...f'$ be over lock l. We get two subcases:
   (a) l is not held by t_1 during e_1.
   Since $e_1 \ll_{HB} f$ and f is the first such event outside t_1, there must be a critical section over l after e_1 and before f. Let g be the lock acquire event of that critical section. g has to be an event by thread t_1, otherwise the definition of f would be violated (there would be another "first" event). Also, g has to be after e_1, by our assumption that l is not held during e_1. Consider a move of $f...f'$ to right before point g. The move respects intra-thread order. Also, if a read sees a different write then a moved event must conflict with one of the non-moved events after g, hence (Lemma 3) we have some e'' such that $e'' \ll_{CP} f'$. But we have $e_1 \ll_{HB} e''$ (by Lemma 1), $e'' \ll_{CP} f'$, and $f' \ll_{HB} e_2$ (by Lemma 1), hence $e_1 \ll_{CP} e_2$: a contradiction.
   Finally, the move may cause a lock, m, to be acquired while being held: this means a critical section acquiring and releasing that lock is inside $f...f'$. (Assume w.l.o.g. that m is the first such lock.) If m is held at point g by a thread t_3, other than t_1, then it has to be released before f, violating the definition of f (since there is a different first event after e_1 by a thread other than t_1).
   A more interesting case is when lock m is held at point g by thread t_1. In that case, lock l is nested inside lock m in thread t_1 (because l is acquired at position g with m held) and lock m is nested inside lock l in thread t_2. We can cause a deadlock by moving a prefix of the $f...f'$ critical section (up until the lock m acquisition) to point g. Therefore the move of $f...f'$ to point g either produces a legal trace t such that $t =_{CR} tr'$, or exposes a deadlock. The move can be repeated until there are no more critical sections over lock l between e_1 and $f...f'$. At that point, we can just move event f to right before e_1. This would produce a correctly reordered trace that violates the minimality of tr': a contradiction.
   We conclude that if Case 1(a) occurs, there is always a deadlock in a correct reordering of trace tr'.
   (b) l is held by t_1 during e_1.

   Let a_2 be the last lock acquisition event of lock l before e_1. Consider a move of $f...f'$ and all previous events by the same thread after a_2 to right before point a_2. Let a'_2 be the lock release paired with a_2. The move respects intra-thread order. Also, if a read sees a different write then a moved event must conflict with one of the non-moved events after a_2, hence (by Lemma 3) we have some e'' such that $e'' \ll_{CP} e'$, where e' is a moved event, and therefore $e'' \ll_{CP} f'$ (since all the moved events precede f' and are by the same thread). But (by Lemma 2) we have $a_2 \ll_{HB} e''$ and therefore $a_2 \ll_{CP} f'$. Since, however, the critical sections starting at a_2 and ending at f' are over the same lock l, we get that $a'_2 \ll_{CP} f'$, because of the second rule in the CP definition. This implies $e_1 \ll_{CP} e_2$ (since e_1 is before a'_2, by assumption of case 1(b), and $f' \ll_{HB} e_2$, by Lemma 1): a contradiction.
   Finally, we consider the case of the move being illegal because it causes a lock, m, to be acquired while being held. If such an m is held by a thread t_3, other than t_1, at point a_2, then it has to be released before e_1 (otherwise the release event would violate the definition of f, since it would come before it, after e_1 and by a thread other than t_1). This means that Lemma 4 applies to the critical section of that thread. Hence, we have that $a_2 \ll_{CP} h'$, where h' is m's release event in t_3. But since h' happens-before some moved event (since the moved events acquire lock m), we get $e_1 \ll_{CP} f'$ (again, all moved events are program-ordered with f') and consequently (via Lemma 1) $e_1 \ll_{CP} e_2$: a contradiction.
   If lock m is held at position a_2 by thread t_1, then l is nested inside m in thread t_1, while m is nested inside l in the thread performing $f...f'$. (The lock acquisition of m by that thread cannot be before f since the lock is released after e_1 and f is the first event after e_1 by a thread other than t_1. Therefore m is acquired and released inside critical section $f...f'$.) As before, we can cause a deadlock by moving a prefix of the $f...f'$ critical section (and any earlier events after a_2 by the same thread) to a_2.
   Therefore, this case again implies a deadlock in a reordering of trace tr'.
2. f' occurs after e_2.
   We then have by Lemma 5 that $f...f'$ has to be performed by thread t_2. Let $f...f'$ be over lock l. Lock l cannot be held by t_1 during event e_1, or $e_1 \ll_{CP} e_2$. Therefore, there must be some critical section $g...g'$ over l, performed by thread t_1 after e_1, such that $g' \ll_{HB} f$. (Recall that f is the first event after e_1 by a thread other than t_1.) Assume w.l.o.g. that $g...g'$ is the last such critical section.
   Consider an event sequence produced as follows:
   - we drop g and all events e' after g by a thread other than t_2.
   - we drop all events after e_2 by all threads.

   Clearly the result is a prefix of tr'. If it is a legal trace that correctly reorders tr' then we are violating the minimality of tr'. Therefore the result of this event drop has to be illegal. The drop respects intra-thread order. Also, if a read sees a different write then a dropped event must conflict with one of the non-dropped events before e_2. By Lemma 3 we get a CP edge between events after e_1 and before e_2 in tr', and by Lemma 1 and CP properties we have $e_1 \ll_{CP} e_2$: a contradiction.
   Thus, the event sequence cannot be a trace: a lock has to be acquired while held. Such a lock, m, has to be acquired before one of the dropped events, with its release among the dropped events. The lock is then re-acquired by one of the non-dropped events, i.e., by thread t_2. The acquisition of m has to be in thread t_1 (otherwise f would not be the first event between e_1 and e_2 by a thread other than t_1). In thread t_1, for trace tr', lock l has to be nested inside m (since dropping every event after g, which is an acquisition of l, caused the drop of the release but not the acquisition of m). However, lock m is nested inside l in thread t_2, since it is acquired after l's acquisition (point f) and before l's release (which occurs after e_2). We can again cause a deadlock with an event move (of a prefix of $f...e_2$).

This concludes the proof of the theorem: any CP race implies either an HB race in the minimal trace tr', or a deadlock in a reordered trace. □