Maximal Sound Predictive Race Detection with Control Flow Abstraction

Jeff Huang Patrick O'Neil Meredith Grigore Rosu

Department of Computer Science, University of Illinois at Urbana-Champaign

{smhuang, pmeredit, grosu}@illinois.edu

Abstract

Despite the numerous static and dynamic program analysis techniques in the literature, data races remain one of the most common bugs in modern concurrent software. Further, the techniques that do exist either have limited detection capability or are unsound, meaning that they report false positives. We present a sound race detection technique that achieves a provably higher detection capability than existing sound techniques. A key insight of our technique is the inclusion of abstracted control flow information into the execution model, which increases the space of the causal model permitted by classical happens-before or causally-precedes based detectors. By encoding the control flow and a minimal set of feasibility constraints as a group of first-order logic formulae, we formulate race detection as a constraint solving problem. Moreover, we formally prove that our formulation achieves the maximal possible detection capability for any sound dynamic race detector with respect to the same input trace under the sequential consistency memory model. We demonstrate via extensive experimentation that our technique detects more races than the other state-of-the-art sound race detection techniques, and that it is scalable to executions of real world concurrent applications with tens of millions of critical events. These experiments also revealed several previously unknown races in real systems (e.g., Eclipse) that have been confirmed or fixed by the developers. Our tool is also adopted by Eclipse developers.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Diagnostics; Debugging aids

General Terms Algorithms, Design, Theory

Keywords Maximal Sound, Data Race, Prediction, Control Flow

1. Introduction

Some of the worst concurrency problems in multithreaded systems today are due to data races, which occur when there are unordered conflicting accesses in the program without proper synchrornization. Data races are particularly problematic because they manifest non-deterministically, often appearing only on very rare executions, making them notoriously difficult to test and debug. We shall refer to data races simply as races in this paper.

PLDI '14, June 09–11, 2014, Edinburgh, United Kingdom.

http://dx.doi.org/10.1145/2594291.2594315

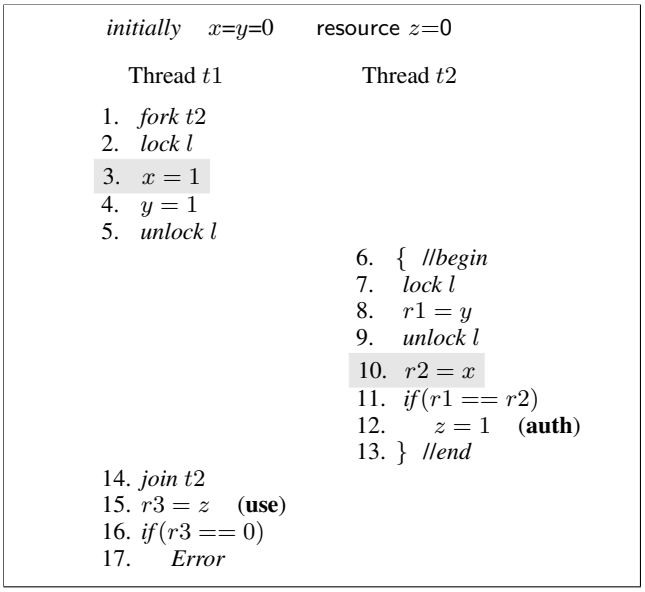


Figure 1. An example program with a race (3,10).

Although researchers have proposed a wide spectrum of techniques [7, 10, 13, 14, 19, 21, 24, 28, 30, 31, 35] to combat races, existing techniques are either unsound (by *soundness* we mean *no-false-positive* in this paper) or have a limited detection capability. The school of *lockset*-based techniques [14, 21, 28, 31] popularized by Eraser [31] is known to be unsound, whereas the *happens-before* (HB) based approaches [7, 13, 19, 24] are often very limited in detecting races, due to extra overly conservative HB edges. Even though a recent development, causally-precedes (CP) [35], improves the detection power by soundly relaxing the HB edges between critical sections that have no conflicting accesses, it can still miss many races. Consider, for instance, an execution of the program in Figure 1. The program contains a race between lines (3,10) that may cause an authentication failure of resource z at line 12, which in consequence causes an error to occur when z is used at line 15. Supposing the execution follows an order denoted by the line numbers, however, CP cannot detect this race because line 3 causally-precedes line 10, for the reason that the two lock regions contain conflicting accesses to y. PECAN [21], another representative technique that uses a hybrid algorithm combining lockset and a weaker form of HB, is able to detect this race by ignoring the HB edges between critical regions. However, the hybrid algorithm is unsound in general. For instance, if we switch lines 1 and 2, (3,10) is no longer a race (because then line 10 will always happen-after line 3), but PECAN will still report it.

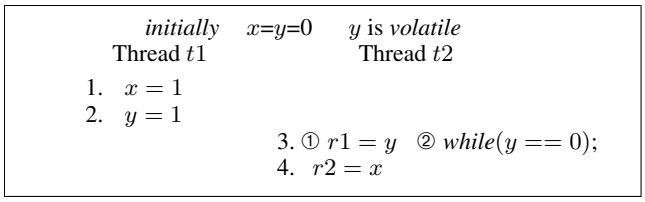


Figure 2. The two cases ① and ② produce the same read/write trace. However, (1,4) is a race in case ① but not in case ②.

In this work, we present a sound dynamic race detection technique that achieves a much higher detection capability than existing techniques. Our key observation is that the control flow information between events in the execution (which is often ignored by existing techniques) can help significantly improve the race detection ability. Consider the simple scenario in Figure 2 where y is volatile and line 3 has two cases: ① $r1 = y$ and ② *while*$(y == 0)$. For case ①, (1,4) is a race on x; while for case ②, it is not, because line 4 is control-dependent on the while loop at line 3. However, without considering the control dependence between operations, the dynamic execution traces for these two cases are identical (both following lines 1-2-3-4). Hence, a sound technique must conservatively assume that a value read by a thread influences all subsequent values produced by the same thread, which, in consequence, creates a HB edge from line 2 to line 3 and misses the race in case ①. However, with the control flow information, we can tell that, in case ①, line 4 is not control-dependent on line 3. In other words, regardless of what value line 3 reads, line 4 will always be executed. Therefore, we can safely drop the HB edge from line 2 to line 3, which enables detecting the race (1,4). Similarly, we are able to detect the race (3,10) in Figure 1 by dropping the HB edge from line 4 to line 8, because there is no control flow from line 8 to line 10 and hence no need to ensure line 8 should read value 1 (written by line 4).

Our first contribution is to add a new type of events—*branch*—into the execution model. Observing branch events is cheap at runtime, however, it provides an abstract view of the control flow information between events that enables a higher race detection power. Moreover, inspired by the theoretical maximal causal model [33], we develop a weaker maximal causal model that incorporates control flow information under the sequential consistency memory model. Underpinned by the new model, we design a maximal race detection algorithm that encodes all the valid trace reorderings allowed by the model as a set of first-order logical constraints, and uses an SMT solver to find races. By formulating race detection as a constraint solving problem, our technique is both *sound* and *maximal*: *every race it detects is real*, accompanying with a valid trace that can manifest it, and *it detects all the possible races that can be detected by any sound technique based on the same trace*.

Our race detection algorithm is inspired by the work of Said *et al.* [30], which also uses SMT-based analysis to detect races. Unlike our approach, [30] does not consider control flow and it is non-maximal. To ensure soundness, [30] requires the *whole trace read-write consistency*: every read returns the same value as that in the original trace (the value may be written by a different write, though). However, this requirement limits the race exploration to only a subset of the feasible traces. For example, [30] cannot detect the race (1,4) in case ①, because it requires line 3 to read the value 1 on y written by line 2, which rules out the incomplete trace 3-1-4 that can manifest the race (1,4). Similarly, [30] cannot detect the the race (3,10) in Figure 1, as line 10 can only read value 1 on x written by line 3. Instead, our technique is concerned with the read-write consistency from the perspective of control dependence, and generates only the constraints with respect to the events that have control flow to the race related operations. Hence, our technique is able to detect races in all feasible incomplete traces as well.

We have implemented our technique for Java and conducted extensive evaluation and comparison with the state-of-the-art sound race detection techniques—HB, CP, and Said *et al.* [30]—on a wide range of popular multithreaded benchmarks as well as real world large concurrent systems. Our results show that our technique detects significantly more races than the other approaches, demonstrating the theoretically higher race detection capability of our approach with the control flow abstraction. Moreover, our technique is practical: it has been applied to real complex executions with tens of millions of critical events and is highly effective in detecting real races. For the seven real systems, our tool detected 299 real races in total. Comparatively, HB, CP, and Said *et al.* only detected 68, 76, and 158 races, respectively. Our experiments also revealed 11 previously unknown races in these real systems that have been confirmed or fixed by the developers. Because of our bug reports in Eclipse [1, 2], the developers have adopted our tool on the codebase of Eclipse Virgo [3].

The following summarizes our contributions:

- We present a sound and maximal causal model incorporating the control flow information for general multithreaded programs under the sequential consistency memory model. This new causal model forms a foundation for maximal dynamic concurrency error detection with control flow. (§2)
- We present a maximal sound dynamic race detection technique based on this new model. We formally prove that our technique is able to detect all races by any sound race detector based on the same execution trace. (§3)
- We present an efficient implementation (§4) and extensive evaluation of our technique, demonstrating the practicality and race detection capability in real world concurrent systems. (§5)

2. Maximal Causal Model With Control Flow

In this section we present our main theoretical contribution, the maximal causal model with control flow, following the axiomatic approach pioneered in [33] (there without control flow). This model paves the theoretical foundation for maximal dynamic concurrency error detectors, such as our race detection technique.

Multithreaded programs $\mathcal{P}$ are abstracted as the prefix-closed sets of finite traces of events that they can produce when completely or partially executed, called $\mathcal{P}$*-feasible* traces. Such sets of traces can be constructed for each $\mathcal{P}$ using, for example, a formal semantics of the target programming language. Regardless of the programming language and of how they are defined, the sets of $\mathcal{P}$-feasible traces must obey some basic consistency axioms. We only consider sequential consistency in this paper. The axioms allow us to associate a *sound* and *maximal* causal model *feasible*(τ) to any consistent trace τ, which comprises precisely the traces that can be generated by all programs that can generate τ. As shown in [33], conventional happens-before causal models consisting of all the legal interleavings of τ and their prefixes are *not* maximal. The maximal causal model allows us to define a maximal notion of race: trace τ has a race iff there is some $\tau' \in$ *feasible*(τ) which contains two consecutive events by different threads that access the same location, at least one of them corresponding to a write.

2.1 Events

The execution environment contains a set of *concurrent objects* (shared locations, locks, etc.), which are accessed by arbitrarily many *threads* to share data and synchronize. A concurrent object is behaviorally defined through a set of atomic operations and a serial specification of its legal behavior in isolation [20]. For example, a *shared memory location* is a concurrent object with *read* and *write* operations, whose serial specification states that each read yields

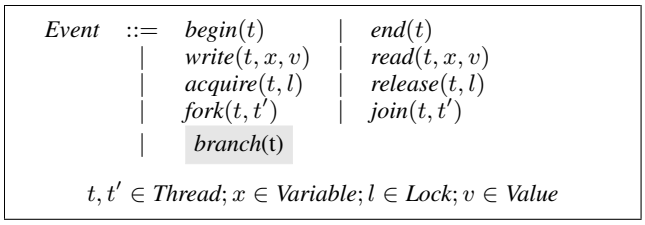


Figure 3. Event types in a multithreaded execution.

the same value as the one of the previous write. A (non-reentrant) *lock* is an object with *acquire* and *release* operations, whose serial specification consists of operation sequences in which the difference between the number of *acquire* and *release* operations is either 0 or 1 for each prefix, and all consecutive pairs of *acquire-release* share the same thread. Virtually all concurrency constructs fall under this simple and general notion of concurrent object [20] (including reentrant locks, fork/join, wait/notify, etc.).

Events are operations performed by threads on concurrent objects, abstracted as tuples of *attribute-value* pairs. For example, $(thread = t_1, op = write, target = x, data = 1)$ is a write event by thread t_1 to memory location x with value 1. When there is no confusion, we take the freedom to use any other meaningful notation for events; for example $write(t_1, x, 1)$. For any event e and attribute *attr*, $attr(e)$ denotes the value corresponding to the attribute *attr* in e, and $e[v/attr]$ denotes the event obtained from e by replacing the value of attribute *attr* by v. The theoretical results in this section hold for any types of events corresponding to any concurrent objects whose behaviors can be described with serial specifications. However, for clarity, we instantiate our subsequent notions and results to the following common event types:

- $begin(t)$/$end(t)$: the first/last event of thread t;
- $read(t, x, v)$/$write(t, x, v)$: read/write a value v on a variable x;
- $acquire(t, l)$/$release(t, l)$: acquire/release a lock l;
- $fork(t, t')$: fork a new thread t';
- $join(t, t')$: block until thread t' terminates;

In addition to the usual events above that have been extensively studied in previous work, we consider a new branch event:

- $branch(t)$: jump to a new operation.

The semantics of this new type of event cannot be given as a serial specification. In fact, branch events can appear anywhere in the trace. Their formal semantics will be defined within the local determinism axiom shortly. To state briefly, the branch event serves as a guard of a possible control flow change, which determines the next operation to execute in a thread. The choice depends on some computation local to the thread, for example the result of an expression in a conditional statement, that is unknown in the event and is not visible to other threads. Hence, conservatively, we assume that the choice of $branch(t)$ depends on all the previous $read(t, x, v)$ operations executed by the same thread.

Figure 3 depicts all the event types discussed above and considered in the rest of the paper, highlighting the novel branch event.

2.2 Traces

An *execution trace* is abstracted as a sequence of events. Given a trace τ and any set S of concurrent objects, threads, or event types, we let $\tau\restriction_S$ denote the restriction of τ to events involving one or more of the elements in S. For example, if o is a concurrent object then $\tau\restriction_o$ is the restriction of τ to events involving o; if t is a thread then $\tau\restriction_t$ contains only the projection of τ to events by thread t; $\tau\restriction_{t,o}$ is the projection of τ to events by thread t involving object o; $\tau\restriction_{t,read}$ the projection to read events by thread t; etc. If e is an event in trace τ then let τ_e denote the prefix of τ up to and including

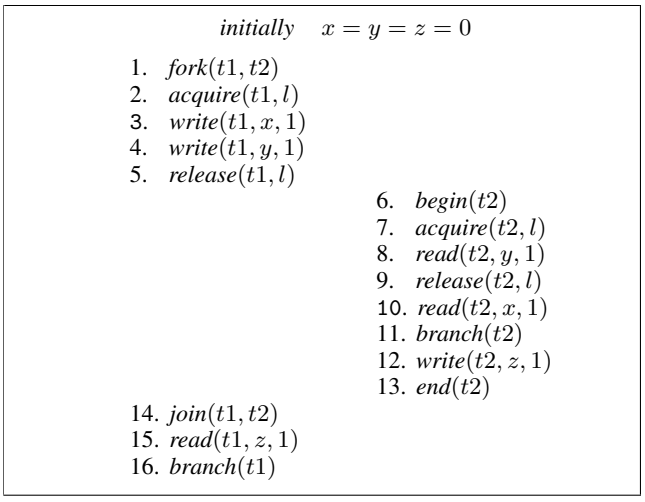


Figure 4. A trace corresponding to the example in Figure 1.

e: if $\tau = \tau_1 e \tau_2$ then τ_e is $\tau_1 e$. Let $last_{op}(\tau)$ be the last event of τ corresponding to operation *op*; e.g., $last_{write}(\tau)$ is the last write event of τ.

An *interleaving* of τ is a trace τ' such that $\tau'\restriction_t = \tau\restriction_t$ for each thread t. Trace τ is *(sequentially) consistent* iff $\tau\restriction_o$ satisfies o's serial specification for any object o [20]. Despite its simplicity, this notion of consistency based on concurrent object serial specifications is quite general. If all the events considered are those in Figure 3, then the consistency of τ means precisely the following:

- **Read Consistency** A read event contains the value written by the most recent write event on the same memory location. Formally, if e is a read event of τ then $data(e) = data(last_{write}(\tau_e\restriction_{target(e)}))$.
- **Lock Mutual Exclusion** Each *release* event is preceded by an *acquire* event on the same lock by the same thread, and each pair is not interleaved by any other *acquire* or *release* event on the same lock. Formally, for any lock l, if $\tau\restriction_l = e_1 e_2 \dots e_n$ then $op(e_k) = acquire$ for all odd indexes $k \leq n$, $op(e_k) = release$ for all even indexes $k \leq n$, and $thread(e_k) = thread(e_{k+1})$ for all odd indexes k with $k < n$.
- **Must Happen-Before** A *begin* event can happen only as a first event in a thread and only after the thread is forked by another thread: for any event $e = begin(t')$ in τ, the trace $\tau\restriction_{t'}$ starts with e and there exists precisely one $fork(t, t')$ event in τ_e. An *end* event can happen only as the last event in a thread, and a *join* event can happen only after the *end* event of the joined thread: for any event $e = end(t')$ in τ, the trace $\tau\restriction_{t'}$ terminates with e; also, for any event $e = join(t, t')$, the event $end(t')$ is in τ_e.

Since the branch events do not have serial specifications, they are allowed to appear anywhere in a trace without affecting the consistency of the trace. Figure 4 shows the (consistent) trace corresponding to our example in Figure 1. Note that read and write to local data (i.e., $r1$, $r2$, $r3$) are not included, as they are not needed for race detection and are also expensive to track in practice.

2.3 Feasibility Axioms

Consistency is a property of a trace alone, stating that all the serial specifications describing the legal behaviors of the involved concurrent objects are met. Any (complete or incomplete) trace produced by a running program is expected to be consistent. However, the various consistent traces that can be generated by a multithreaded program are not unrelated. Let $feasible(\mathcal{P})$ be the set of all

traces that can be produced by a hypothetical program $\mathcal{P}$, which we call $\mathcal{P}$-*feasible* traces. The most common characterizing axiom of *feasible*($\mathcal{P}$), rooted in Lamport's happens-before causality [22] and Mazurkiewicz' trace theory [25], is to require *feasible*($\mathcal{P}$) be closed under consistent interleavings. For the trace in Figure 4, e.g., this tells that consistent interleavings such as 1-6-2-3-4-5-7-8-9... and 1-2-6-3-4-5-7-8-9..., where we refer to events by their line numbers, are also $\mathcal{P}$-feasible, regardless of the program $\mathcal{P}$ that generated the original trace. This axiom is, however, too strong. What we want is the weakest axioms of *feasible*($\mathcal{P}$), which will give the resulting concurrency error detection technique the largest coverage.

Two weaker axioms governing *feasible*($\mathcal{P}$) are proposed in [33]: *prefix closedness* and *local determinism*. The former says that the prefixes of a $\mathcal{P}$-feasible trace are also $\mathcal{P}$-feasible. The latter says that each thread has a deterministic behavior, that is, only the previous events of a thread (and not other events of other threads) determine the next event of the thread, although if that event is a read then it is allowed to get its value from the latest write. Formally, if $\tau_1 e_1, \tau_2 \in$ *feasible*($\mathcal{P}$) and $\tau_1{\restriction}_{thread(e_1)} = \tau_2{\restriction}_{thread(e_1)}$ then (1) if $\tau_2 e_1$ is consistent then $\tau_2 e_1 \in$ *feasible*($\mathcal{P}$), and (2) if $op(e_1) =$ *read* and there exists some event e_2 with $e_2[data(e_1)/data] = e_1$ and $\tau_2 e_2$ consistent, then $\tau_2 e_2 \in$ *feasible*($\mathcal{P}$). Unlike the consistent interleavings axiom underlying the usual happens-before, these weaker axioms allow us to infer from the trace in Figure 4 that the consistent trace 1-6-7-8' is also $\mathcal{P}$-feasible regardless of the program $\mathcal{P}$ that generated the original trace, where 8' is the event *read*$(t2, y, 0)$ which reads 0, the value written initially to y, instead of 1 as the original event 8. Note, however, that the trace 1-6-7-8'-9 can *not* be inferred as $\mathcal{P}$-feasible with the axioms and model in [33] without control flow, because the projections of 1-6-7-8 and 1-6-7-8' to thread $t2$ are not equal. It would in fact be unsound to allow the trace 1-6-7-8'-9 to be inferred as $\mathcal{P}$-feasible, because the read of 1 for y in event 8 may be part of the condition in a conditional statement in $\mathcal{P}$, and the release event 9 generated by a branch that would not be taken if y were 0.

Our objective in the remainder of this section is to give the weakest possible axioms of feasibility that take into account our new branch events. Consider, for example, an assignment $x = y + 1$ inside a branch that has just been taken. The read of y and the write of x will happen unconditionally; other threads may at most change the data values of these events (in a sequentially consistent manner), but they cannot affect the existence of the events themselves. Moreover, the write of x event contains the same value, provided that the read of y event reads the same value. Similarly, the existence of concurrent operations inside the taken branch is conditioned only by global sequential consistency constraints, but not by the particular data values read or written by the current thread. Hence, the fact that the branch events alone determine the control flow in the original program allows us to significantly relax the requirement of the local determinism axiom of [33] that traces τ_1 and τ_2 must have *identical* projections on e's thread t in order for $\tau_1 e$ $\mathcal{P}$-feasible to determine the next event by t in the $\mathcal{P}$-feasible extensions of τ_2. Instead, we can now only require the two projections to be equal except for the data values in write and read events. However, since the branch and write events depend upon the evaluation of thread-local expressions which are not available in the execution trace, we conservatively assume that these events depend upon all the reads by the same thread.

Let $\mathcal{F}$ be a set of traces. Below we identify five axioms for $\mathcal{F}$ to be *feasible*. The intuition is that the set of traces that can be generated by a multithreaded program is such a feasible set.

Prefix Closedness *$\mathcal{F}$ is prefix closed*: if $\tau_1\tau_2 \in \mathcal{F}$ then $\tau_1 \in \mathcal{F}$. Prefix closedness ensures that events are generated in execution order, with the possibility of interleaving in-between any of them.

For the remaining axioms, suppose that $\tau_1 e_1, \tau_2 \in \mathcal{F}$, that *thread*$(e_1) = t$, and that $\tau_1{\restriction}_t \approx \tau_2{\restriction}_t$, where two traces are in the $\approx$ relation, called *data-abstract equivalence*, iff they are equal except for the data values in *read* and *write* events. The next axioms state the thread-local determinism requirements. Specifically, they state that under the above thread-local data-abstract equivalence condition between τ_1 and τ_2, the trace τ_2 can only continue with a well-determined event of t, which is data-abstract equivalent to e_1.

Local Determinism *Each event is determined by the previous events in the same thread and can occur at any consistent moment after them.* There are four cases to distinguish:

- ***Branch.*** If $op(e_1) =$ *branch* and $\tau_1{\restriction}_{t,read} = \tau_2{\restriction}_{t,read}$, then $\tau_2 e_1 \in \mathcal{F}$.
- ***Read.*** If $op(e_1) =$ *read* and e_2 is a (read) event such that $e_2[data(e_1)/data] = e_1$ and $\tau_2 e_2$ is consistent, then $\tau_2 e_2 \in \mathcal{F}$.
- ***Write.*** If $op(e_1) =$ *write* then there is a v such that $\tau_2 e_1[v/data] \in \mathcal{F}$; moreover, if $\tau_1{\restriction}_{t,read} = \tau_2{\restriction}_{t,read}$ then $v = data(e_1)$.
- ***Other.*** If $op(e_1) \notin \{branch, read, write\}$ and $\tau_2 e_1$ is consistent, then $\tau_2 e_1 \in \mathcal{F}$.

DEFINITION 1. *A set of traces $\mathcal{F}$ is **feasible** if it satisfies the prefix closedness and the local determinism axioms above.*

In addition to all the consistent interleavings and feasible sets of traces according to [33] derived from the trace τ in Figure 4 discussed above, we can now show that any feasible $\mathcal{F}$ including τ also includes many more other traces. For example, the trace 1-6-7-8'-9 with 8' the event *read*$(t2, y, 0)$ which was not allowed before is allowed now as feasible, and so is the trace 1-6-7-8'-9-2-3-10, which shows a race on x by events 3 and 10. Note that there is no way to infer a trace in $\mathcal{F}$ that brings events 4 and 8 (race on y) or 12 and 15 (race on z) next to each other, because that would violate the lock-mutual exclusion or must happen-before consistency.

Discussion We have made two assumptions and adopted a deliberate limitation in our feasibility axioms above. First, we assumed that the branch events and the data values in write events depend only on the previous read events by the same thread. If there are other factors that determine these events in a particular language, such as random jumps or expressions, then one either needs to generate additional read events corresponding to those external factors or alternatively to explicitly consider them as special events and modify the axioms accordingly. Second, we assumed that *all* possible places where the control dependence may be changed are logged as explicit branch events, e.g., mutable pointer dereferences and array indexing. If there are implicit control flow points in a particular program that are activated by the data flow, such as an exception thrown when a division by zero is performed, then one needs to generate additional branch events after each such implicit control choice. Third, we conservatively assumed that each branch or write depends on *all* the previous read events by the same thread. In most cases branch and write events only depend on the values read within the evaluation of a particular expression. We could get even weaker axioms if we assumed a preceding window of events for each write and branch in which the read values matter for these events, but that would involve more complex events and axioms.

2.4 Sound and Maximal Causal Model

Our objective here is to associate to any given consistent trace τ a *sound* and *maximal causal model*, *feasible*(τ), comprising precisely all the traces that can be generated by any program that can generate τ. It is irrelevant at this stage how we represent such a model; in Section 3 we show a way to represent it by means of logical constraints. *Soundness* means that any program $\mathcal{P}$ that can produce τ can also produce any of the traces of *feasible*(τ).

Maximality means that for any trace τ' which is not in *feasible*(τ) there is some program $\mathcal{P}$ which can produce τ but not τ'.

Following [33] (there without control flow), a natural choice for *feasible*(τ) would be the smallest set of traces that includes τ and is closed under the feasibility axioms. However, that simplistic approach does not work here, mainly because of the local write determinism axiom (Section 2.3): it would be unsound to pick any particular value v in the write event, because we have no further information about the program that generated the original trace τ and thus it is impossible to know how it computes the written value. To avoid picking any particular value v, we instead modify the second case of the local data-abstract determinism axiom to introduce (fresh) symbolic values. We keep all the other axioms in Section 2.3 unchanged, but note that traces appearing in *feasible*(τ) can contain symbolic values in their *read* and *write* events. For that reason, we call the new local determinism axioms *local symbolic determinism*. We use the symbolic axiom variant only to define our maximal causal model; for abstractions of programs $\mathcal{P}$ as their sets of traces we continue to use the non-symbolic axiom variant in Section 2.3. For clarity, below we give the formal definition of *feasible*(τ).

Let *Sym* be an infinite set of symbolic values. For technical reasons we assume that given any trace τ, we can always pick an arbitrary but fixed symbolic value sym_τ which is distinct from any other similar symbolic value: if $\tau_1 \neq \tau_2$ then $sym_{\tau_1} \neq sym_{\tau_2}$.

DEFINITION 2. *Given a consistent trace τ, let feasible(τ) be the* ***feasibility closure*** *of τ defined as the smallest set of (symbolic) traces that includes τ and is closed under the following operations:*

1. ***Prefixes.*** *if $\tau_1\tau_2 \in$ feasible(τ) then $\tau_1 \in$ feasible(τ).*
2. ***Local symbolic determinism.*** *Assume that $\tau_1 e_1, \tau_2 \in$ feasible(τ), that thread$(e_1) = t$, and that $\tau_1{\restriction_t} \approx \tau_2{\restriction_t}$. Then*
   - **Branch.** *If op(e_1) = branch and $\tau_1{\restriction_{t,read}} = \tau_2{\restriction_{t,read}}$ then $\tau_2 e_1 \in$ feasible(τ).*
   - **Read.** *If op(e_1) = read and e_2 is such that $e_2[data(e_1)/data] = e_1$ and $\tau_2 e_2$ is consistent, then $\tau_2 e_2 \in$ feasible(τ).*
   - **Write.** *Suppose that op(e_1) = write. There are two cases to distinguish: if $\tau_1{\restriction_{t,read}} \neq \tau_2{\restriction_{t,read}}$ then $\tau_2 e_1[sym_{\tau_2}/data] \in$ feasible(τ); if $\tau_1{\restriction_{t,read}} = \tau_2{\restriction_{t,read}}$ then $\tau_2 e_1 \in$ feasible(τ).*
   - **Other.** *If op$(e_1) \notin \{branch, read, write\}$ and $\tau_2 e_1$ is consistent, then $\tau_2 e_1 \in$ feasible(τ).*

*A trace in feasible(τ) is called τ-**feasible**.*

It can be easily seen that for any mapping θ of symbolic values to concrete values, $\theta($*feasible*$(\tau))$ is a feasible set of traces, in the sense of Definition 1. Recall that we abstract multithreaded programs as feasible sets of traces, namely all complete or incomplete traces that they can produce when executed. We can think of *feasible*(τ) as an abstract representation of all causal dependencies revealed by τ in all programs that can produce τ when executed, each θ corresponding to such a program. This intuition will be formally captured below, by our soundness and maximality results.

The next result states the soundness of our causal model:

THEOREM 1 (Soundness). *Suppose that $\mathcal{F}$ is a feasible set of (concrete) traces, like in Definition 1, and that $\tau \in \mathcal{F}$ is a consistent trace. Then there exists a mapping θ of symbolic values into concrete values such that $\theta($*feasible*$(\tau)) \subseteq \mathcal{F}$.*

PROOF: Since *feasible*(τ) is the smallest set of traces closed under prefixes and the local symbolic determinism axioms in Definition 2, we can order the traces in *feasible*(τ), say $\tau^0 = \tau$, τ^1, ..., τ^n, ... for $n \in \mathbb{N}$, so that each trace τ^{n+1} can be derived from one (if a prefix) or from two (if a locally deterministic continuation) of the traces τ^0, τ^1, ..., τ^n. We construct by induction on n a sequence of partial mappings $\theta_0 = \bot \sqsubseteq \theta_1 \sqsubseteq \theta_2 \sqsubseteq \cdots \sqsubseteq \theta_n \sqsubseteq \cdots$ taking symbolic to concrete values, where $f \sqsubseteq g$ iff *Dom*$(f) \subseteq$ *Dom*(g) and $f(s) = g(s)$ for each $s \in$ *Dom*(f), such that $\theta_n(\tau^n) \in \mathcal{F}$ for all $n \in \mathbb{N}$. Note that $\theta_i(\tau^i) = \theta_j(\tau^i)$ for any $i \leq j$. Then the result immediately holds, because we can take θ to be the least upper bound (lub) of the chain of these partial functions, $\bigsqcup_{n\in\mathbb{N}} \theta_n$.

If $n = 0$ then we pick $\theta_0 = \bot$; since $\tau^0 = \tau \in \mathcal{F}$ is a concrete trace, $\theta_0(\tau^0) = \tau^0 \in \mathcal{F}$. Now suppose that the desired property holds for all indexes less than or equal to n, and let us prove it for $n + 1$. If τ^{n+1} is derived as a prefix of some $\tau' \in \{\tau^0, \tau^1, \ldots, \tau^n\}$, then let θ_{n+1} be θ_n. By the induction hypothesis, $\theta_n(\tau') \in \mathcal{F}$, so $\theta_{n+1}(\tau^{n+1}) = \theta_n(\tau^{n+1}) \in \mathcal{F}$ because $\mathcal{F}$ is prefix closed (Definition 1). If τ^{n+1} is derived using a local symbolic determinism axiom, there there must exist two traces $\tau_1 e_1, \tau_2 \in \{\tau^0, \tau^1, \ldots, \tau^n\}$ such that $\tau_1{\restriction_t} \approx \tau_2{\restriction_t}$, where $t =$ *thread*(e_1). By the induction hypothesis, $\theta_n(\tau_1 e_1), \theta_n(\tau_2) \in \mathcal{F}$. Note also that $\theta_n(\tau_1){\restriction_t} \approx \theta_n(\tau_2){\restriction_t}$. If *op*$(e_1) =$ *branch* then it must be the case that $\tau_1{\restriction_{t,read}} = \tau_2{\restriction_{t,read}}$, so $\theta_n(\tau_1){\restriction_{t,read}} = \theta_n(\tau_2){\restriction_{t,read}}$, and that $\tau^{n+1} = \tau_2 e_1$. Let θ_{n+1} be θ_n. Then $\theta_{n+1}(\tau^{n+1}) = \theta_n(\tau_2)e_1 \in \mathcal{F}$ because of the local branch determinism of $\mathcal{F}$ (Definition 1). If *op*$(e_1) =$ *read* then it must be the case that there is some event e_2 such that $e_2[data(e_1)/data] = e_1$, so $\theta_n(e_2)[data(\theta_n(e_1))/data] = \theta_n(e_1)$, $\tau_2 e_2$ is consistent, so $\theta_n(\tau_2)\theta_n(e_2)$ is consistent, and $\tau^{n+1} = \tau_2 e_2$. Let θ_{n+1} be θ_n. Then $\theta_{n+1}(\tau^{n+1}) = \theta_n(\tau_2)\theta_n(e_2) \in \mathcal{F}$ because of the local read determinism of $\mathcal{F}$. Now suppose that *op*$(e_1) =$ *write*. There are two cases to distinguish. If $\tau_1{\restriction_{t,read}} = \tau_2{\restriction_{t,read}}$ then it must be the case that $\tau^{n+1} = \tau_2 e_1$. In this case we let θ_{n+1} be θ_n and $\theta_{n+1} \in \mathcal{F}$ follows similarly to the previous cases. If $\tau_1{\restriction_{t,read}} \neq \tau_2{\restriction_{t,read}}$ then it must be the case that $\tau^{n+1} = \tau_2 e_1[sym_{\tau_2}/data]$. By the local write determinism of $\mathcal{F}$ (Definition 1), there is some value v such that $\theta_n(\tau_2)\theta_n(e_1)[v/data] \in \mathcal{F}$. In this case we pick θ_{n+1} to be equal to θ_n in all symbolic values in which θ_n is defined, and $\theta_{n+1}(sym_{\tau_2}) = v$. Note that θ_{n+1} is well-defined because of our assumption that sym_{τ_2} is uniquely determined by τ_2. Finally, if *op*$(e_1) \notin \{branch, read, write\}$ then it must be the case that $\tau_2 e_1$ is consistent, so $\theta_n(\tau_2)\theta_n(e_1)$ is consistent, and that $\tau^{n+1} = \tau_2 e_1$. In this case we can again let θ_{n+1} be θ_n. Then $\theta_{n+1}(\tau^{n+1}) = \theta_n(\tau_2)\theta_n(e_1) \in \mathcal{F}$ also by the feasibility of $\mathcal{F}$. □

In words, the soundness theorem says that if a hypothetical program $\mathcal{P}$ (abstracted above by the complete or incomplete traces in $\mathcal{F}$ that it can produce) generates a trace τ, then any τ-feasible trace, which may contain symbolic data values, corresponds to some concrete trace τ' that $\mathcal{P}$ can also generate, obtained by instantiating the symbolic values with some concrete ones. Therefore, if a dynamic error detection technique is based on our maximal causal model, say a data race detector, then any error reported by the technique is a real error, which can happen under a different thread schedule.

The next result states the maximality of our sound causal model:

THEOREM 2 (Maximality). *Suppose that τ, τ' are concrete traces such that τ is consistent and $\tau' \notin \theta($feasible$(\tau))$ for any θ mapping symbolic values to concrete values. Then there is a multithreaded program $\mathcal{P}$ with $\tau \in$ feasible$(\mathcal{P})$ and $\tau' \notin$ feasible$(\mathcal{P})$.*

PROOF: To refer to programs and their execution traces, we formally define a simple programming language that can produce all the events that we consider in our traces, noting that other events can similarly be supported and that this language is so basic that its instructions can easily be reproduced in any other language. The formal definition of the language is then used to show that the set of traces that any program $\mathcal{P}$ can produce when executed, *feasible*$(\mathcal{P})$, is indeed a feasible set in the sense of Definition 1. Finally, given a consistent trace τ we construct a program $\mathcal{P}_\tau$ such that *feasible*$(\mathcal{P}_\tau) \subseteq \bigcup_\theta \theta($*feasible*$(\tau))$. We next detail the above.

For brevity, we here describe the language and its semantics informally. The language has threads which can be forked and joined, (non-reentrant) locks which can be acquired and released, and both shared and thread-local variables. Shared variables can only be read and written with simple assignments $r := x$ and $x := r$, respectively, where r is a local and x is shared. The trace semantics of the language is that all statements of the language produce corresponding events when executed, except for reads and writes of local variables. Complex assignments of the form $r := (r_1 == v_1)\&\& \cdots \&\&(r_n == v_n)?v : v'$ are also allowed, where $r, r_1, ..., r_n$ are locals and $v_1, ..., v_n, v, v'$ are values, with the meaning that if r_i equals v_i for all $1 \leq i \leq n$ then r takes the value v, otherwise v'. However, these generate no events. Finally, we also introduce a simple conditional statement to account for branch events: $\texttt{if}(r)$. Its semantics is that it produces a branch event and the execution continues only if local variable r is 1; otherwise the execution gets stuck without a branch event. Using the language semantics, it is relatively easy (albeit tedious) to define the set *feasible*$(\mathcal{P})$ of all complete or incomplete traces of a program $\mathcal{P}$, and to show that it satisfies the feasibility axioms in Definition 1.

The only thing left is to construct a particular program $\mathcal{P}_\tau$ from a consistent trace τ such that *feasible*$(\mathcal{P}_\tau) \subseteq \bigcup_\theta \theta($*feasible*$(\tau))$, where the union goes over all mappings θ from symbolic to concrete values. The idea is to traverse the trace τ and generate the program $\mathcal{P}_\tau$ by replacing each event in τ with one or more corresponding instructions in $\mathcal{P}_\tau$. We discuss the read, write and branch events last. Each *fork*$(t1, t2)$ event generates a corresponding fork statement in thread $t1$, making sure that all subsequent events of thread $t2$ are used to generate instructions in the forked thread. Similarly, each *join*$(t1, t2)$ event generates a corresponding join statement in thread $t1$. Events *acquire*(t, l) and *release*(t, l) generate corresponding acquire and release instructions of lock l in thread t, and similarly for wait/notify events. The interesting events are the reads, writes and branches. For each event *read*(t, x, v), we generate an assignment statement $r := x$, where r is a fresh local variable that we keep track of in the generation algorithm that it is paired with value v. For each event *write*(t, x, v) we generate two instructions in thread t,

$$r := (r_1 == v_1)\&\& \cdots \&\&(r_n == v_n)?v : v'$$
$$x := r$$

where (r_1, v_1), ..., (r_n, v_n) are *all* the pairs between a local variable and a value corresponding to read events as above that the generation algorithm stored for thread t so far, and where v' is a value distinct from v. Finally, for events *branch*(t) we generate the following instructions in thread t:

$$r := (r_1 == v_1)\&\& \cdots \&\&(r_n == v_n)?1 : 0$$
$$\texttt{if}(r)$$

In both cases above, the complex assignment ensures that r gets the expected value only if the thread's read history is the same as that in the original trace τ. In the case of *write*, if that is the case then the precise value v that appeared in τ is written, which accounts for the first case of the local write determinism axiom before Definition 1; otherwise a different value v' is written, which accounts for the second case of the local write determinism axiom. We need not worry about which particular value v' should be written to avoid having τ' as a possible trace, because $\tau' \notin \theta($*feasible*$(\tau))$ for any θ guarantees that no v' has this property. In the case of *branch*, the thread is allowed to continue only if its read history is identical to that of τ; otherwise the thread gets stuck without issuing a branch event. This accounts for the local branch determinism axiom.

The generated program $\mathcal{P}_\tau$ is therefore quite simple, its instructions corresponding almost identically to the trace τ and having no loops; in fact, our simple language is not even Turing-complete. Using the language semantics we can show that $\tau \in$ *feasible*$(\mathcal{P}_\tau)$, and also that any trace τ'' in *feasible*$(\mathcal{P}_\tau)$ is included in $\theta($*feasible*$(\tau))$ for some mapping θ of symbolic to concrete values; specifically, it is a θ that maps symbolic values sym_{τ_2} introduced by the second case of the local write determinism of the feasibility closure (see Definition 2) to (arbitrarily chosen) concrete values v' as in the corresponding instructions associated to the write event. Therefore, *feasible*$(\mathcal{P}_\tau) \subseteq \bigcup_\theta \theta($*feasible*$(\tau))$, so $\mathcal{P}_\tau$ is such a multithreaded program $\mathcal{P}$ with $\tau \in$ *feasible*$(\mathcal{P})$ and $\tau' \notin$ *feasible*$(\mathcal{P})$. □

In words, the maximality theorem says that for any concrete trace τ' which is not an instance of a (possibly symbolic) τ-feasible trace, there exists a "witness" program $\mathcal{P}$ that can produce τ but not τ'. Therefore, any dynamic error detection technique that produces counterexample traces which are not instances of τ-feasible traces, must be unsound: there are programs for which they report false alarms. Of course, our soundness and maximality results are intrinsically based on the assumption that the traces generated by multithreaded programs obey our feasibility axioms for sequential consistency[1] (see also the discussion at the end of Section 2.3), and the statements of the target multithreaded programming language have the granularity of our events.

2.5 Maximal Causal Properties

The existence of a maximal causal model allows us to define maximal variants of concurrency properties, such as races, atomicity, etc. In this paper we only focus on races, but the same maximal causal model approach can be used to define other notions.

DEFINITION 3 (COP). *Events a and b form a **conflicting operation pair**, written COP(a, b), iff op(a) = write, op$(b) \in \{$write, read$\}$, target(a) = target(b), and thread$(a) \neq$ thread(b).*

DEFINITION 4 (Data race). *Consistent trace τ has a **race** iff there is a consistent trace $\tau_1 ab \in$ feasible(τ) such that COP(a, b)* [2].

Consider again the trace τ in Figure 4. There are three conflicting pairs: COP$(3, 10)$, COP$(4, 8)$, and COP$(12, 15)$. However, only COP$(3, 10)$ is a race because, as previously discussed, there is a consistent trace $\texttt{1-6-7-8'-9-2-3-10} \in$ *feasible*(τ), with $\texttt{8'} =$ *read*$(t2, y, 0)$, in which event `3` is immediately before `10`, and there is no way to bring events `4` and `8` (race on y) or `12` and `15` (race on z) next to each other without breaking consistency.

Theorem 1 implies that our notion of a race above is sound, so any dynamic race detection technique reporting only races among those in Definition 4 is sound (no false alarms). However, the key feature of our definition of a race is its maximality: the witness of the race in τ is a (possibly symbolic) trace τ' that belongs to the maximal causal model of τ; note that the symbolicity of τ' is irrelevant for races. Therefore, any sound (not necessarily maximal) dynamic race detection technique can detect no races which are not captured by Definition 4. A technique that can detect precisely all the races in Definition 4, like our technique presented shortly in Section 3, is therefore both sound and maximal.

3. Maximal Dynamic Race Detection

This section presents our technique for maximal dynamic race detection. We first give an illustrative technical overview, followed by the formal modeling of our technique based on the maximal causal model foundation presented in Section 2.

3.1 Technical Overview

Given an input trace τ, the goal of dynamic race detection is to find a *τ-feasible* trace τ' and a COP(a, b) such that a and b are

[1] Other memory models would require a different feasibility axiomatization.

[2] Note that the dual case $\tau_1 ba$ is equivalent and we can consider either one for defining a race and for race detection [30].

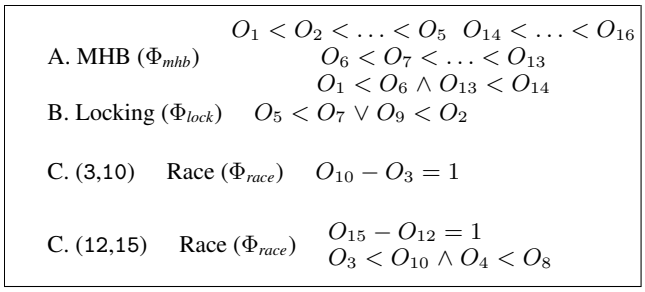


Figure 5. Constraint modeling of the example trace in Figure 4.

next to each other in τ' (Definition 4). Since here we only discuss race detection, where the particular values written or read by events are irrelevant, to simplify the presentation we make no distinction between an event that appears in τ and its data-abstractly equivalent variants appearing in τ-feasible traces. We formulate the maximal race detection problem as a constraint solving problem. Specifically, we introduce an order variable O_e for each event e in τ, which represents the order of e in τ'. Then we generate a formula Φ over these variables corresponding to the race problem for τ and COP(a, b), that is, one which is satisfiable iff $O_b - O_a = 1$ for some $\tau' \in$ *feasible*(τ). By solving Φ using any constraint solver, we are able to determine whether (a, b) is a race or not.

For concreteness, we only consider the common concurrent objects that yield the event types in Figure 3, whose serial specifications generate the consistency requirements spelled out at the end of Section 2.2. Figure 5 shows our constraint modeling of the example trace in Figure 4. Let O_i refer to the order variable of the event at line i. The constraints consist of three parts: (A) the must happen-before (MHB) constraints, (B) the locking constraints, and (C) the race constraints. A and B are common for all races, whereas C is race specific. For instance, the MHB constraints for the *fork* event at line 1 and the *join* event at line 14 are written as $O_1 < O_6 \wedge O_{14} > O_{13}$, meaning that the *fork* event should happen before the *begin* event of $t2$ at line 6, and the *join* event should happen after the *end* event of $t2$ at line 13, which are determined by the must happen-before consistency requirement in Section 2.2.

The locking constraints encode lock mutual exclusion consistency over *acquire* and *release* events. For example, $O_5 < O_7 \vee O_9 < O_2$ means that either $t1$ acquires the lock l first and $t2$ second, or $t2$ acquires l first and $t1$ second. If $t1$ first, then the *acquire* at line 7 must happen after the *release* at line 5; otherwise if $t2$ first, the *acquire* at line 2 should happen after the *release* at line 9.

The race constraints encode the race and control flow conditions specific to each COP. For example, for the COP (3,10), the race constraint is written as $O_{10} - O_3 = 1$, and its control-flow condition is empty, because there is no *branch* event before the two events at lines 3 and 10. For (12,15), however, because there is a *branch* event (at line 11) before line 12, in addition to the race constraint $O_{15} - O_{12} = 1$, we need to ensure that the control-flow condition at the *branch* event is satisfied. To respect the local branch determinism axiom in Section 2.3, we require that all *read* events by $t2$ before this *branch* event read the same value as that in the original trace. Hence, we add the control-flow constraints $O_3 < O_{10} \wedge O_4 < O_8$ to ensure that the *read* event at line 10 reads value 1 on x, and that the *read* event at line 8 reads value 1 on y. This guarantees that the event at line 12 is feasible.

Putting all these constraints together, we invoke an SMT solver, such as Z3 [11] or Yices [12] in our current implementation, to compute a solution for these unknown order variables. For (3,10), the solver returns a solution which corresponds to the schedule 1-6-7-8-9-2-3-10, so (3,10) is a race. For (12,15), the solver reports no solution exists, so it is not a race.

The above example illustrates how our technique works in a nutshell. We present our implementation and optimization details in Section 4. We next formalize our constraint modeling in detail.

3.2 Constraint Modeling

As mentioned, given an observed trace τ, we encode the maximal race detection problem as a formula Φ specifying all the τ-feasible traces with respect to each race. Φ contains only variables of the form O_e corresponding to events e, which denote the order of the events in the to-be-computed τ-feasible trace (if there exists one) that can manifest the race. Although we define a race of τ as a property over the maximal causal model *feasible*(τ) (Definition 4), for performance reasons we purposely do *not* follow the same approach here when generating the constraints. That is because the characterizing formula of *feasible*(τ) would be unnecessarily complex for the bare task of detecting races, e.g., it would need to generate constraints for all branches, not only for those immediately guarding the events in a COP, and to account for the fact that the constraints corresponding to events following an invalidated branch do not influence the overall formula satisfiability. Thus, Φ is constructed by a conjunction of three sub-formulae:

$$\Phi = \Phi_{mhb} \wedge \Phi_{lock} \wedge \Phi_{race}$$

MHB Constraints (Φ_{mhb}) The must happen-before (MHB) consistency requirements discussed at the end of Section 2.2 cover all the trace consistency requirements except for those of the *read*/*write* and *acquire*/*release* events (which we treat differently). MHB, together with the total orders of the events in each thread, yield an obvious partial order $\prec$ on the events of τ which must be respected by any τ-feasible trace. Fortunately, $\prec$ can be specified quite easily as constraints over the O variables: we start with $\Phi_{mhb} \equiv$ *true* and conjunct it with a constraint $O_{e_1} < O_{e_2}$ whenever e_1 and e_2 are events by the same thread and e_1 occurs before e_2, or when e_1 is an event of the form *fork*(t, t') and e_2 of the form *begin*(t'), etc. We assume a background first-order theory where $<$ is transitive, like in Z3 and Yices, so we do not need to encode the transitivity of $\prec$. The size of Φ_{mhb} is linear in the length of τ.

Note that our MHB differs from Lamport's *happens-before* [22] in two aspects: (1) *read* and *write* events by different threads are not included, because their order may be consistently commutable; (2) *acquire* and *release* events are not included, because the order between different lock regions may also be consistently commutable.

Locking Constraints (Φ_{lock}) Lock mutual exclusion consistency (Section 2.2) requires that two sequences of events protected by the same lock do not interleave. Φ_{lock} captures the ordering constraints over the lock *acquire* and *release* events. For each lock l, we extract the set S_l of all the corresponding pairs (a, r) of *acquire*/*release* events on l, following the program order locking semantics: the *release* is paired with the most recent *acquire* on the same lock by the same thread. Then we conjunct Φ_{lock} with the formula

$$\bigwedge_{(a,r),(a',r') \in S_l} (O_r < O_{a'} \vee O_{r'} < O_a)$$

which is quadratic in the number *acquire*/*release* events on l in τ.

Race Constraints (Φ_{race}) For a COP(a, b), Φ_{race} contains two categories of constraints: the arithmetic constraint $O_b - O_a = 1$ that specifies the race condition, and a conjunction of two control-flow constraints $\Phi^{\approx}_{cf}(a) \wedge \Phi^{\approx}_{cf}(b)$ specifying the data-abstract feasibility of a and b. We next define $\Phi^{\approx}_{cf}(e)$ for any read/write event e.

Recall the local determinism axioms that the events appearing in τ-feasible traces follow a deterministic order when projected on their thread, but the data values in read and write events are allowed to be different from those in τ, in order to satisfy the read consistency requirement of the τ-feasible trace. Making abstraction

of the particular data values in read/write events, the only factor that can affect the feasibility of such an event e, in addition to the MHB and lock consistency requirements which are already encoded as detailed above, is that some event e' that must happen before e, i.e., $e' \prec e$, is infeasible because of a previous branch by the same thread that is infeasible. Because of the local branch determinism axiom, the feasibility of a branch is determined by the complete read history of its thread, so we only need to consider the feasibility of the last branch event of each thread that must happen before e. Let $\mathcal{B}_e$ be the set of last branch events e' of each thread with $e' \prec e$. Then let

$$\Phi^{\approx}_{cf}(e) = \bigwedge_{e' \in \mathcal{B}_e} \Phi_{cf}(e'), \qquad op(e) \in \{read, write\}$$

be the formula stating that the data-abstract feasibility of a read or write event e reduces to the feasibility of all the branch events in $\mathcal{B}_e$. We next model the *concrete* feasibility of any read, write, or branch event e as a formula $\Phi_{cf}(e)$, where "concrete" means that e appears unchanged, including its data value, in the τ-feasible trace.

According to the local branch and write determinism axioms, the concrete feasibility of *branch* and *write* events is determined by the complete read history of their thread, that is,

$$\Phi_{cf}(e) = \bigwedge_{r \in \tau_e \restriction_{t,read}} \Phi_{cf}(r), \quad op(e) \in \{branch, write\},\ t = thread(e)$$

So far, we have straightforwardly encoded the axioms of the maximal causal model in Section 2 using constraints. The part which does not follow explicitly from the axioms is how to encode the concrete feasibility of *read* events (needed in formula above). We need to ensure that a $read(t, x, v)$ event reads the same value v written by a concretely feasible $write(\_, x, v)$ event ("_" means any thread). Specifically, if for a read event r, say $read(t, x, v)$, we let W^r be the set of $write(\_, x, \_)$ events in τ, and W^r_v the set of $write(\_, x, v)$ events in τ, then we define the following:

$$\Phi_{cf}(r) = \bigvee_{w \in W^r_v} (\Phi_{cf}(w) \wedge O_w < O_r \bigwedge_{w \neq w' \in W^r} (O_{w'} < O_w \vee O_r < O_{w'}))$$

The above states that the read event $r = read(t, x, v)$ may read the value v on x written by any *write* event $w = write(\_, x, v)$ in W^r_v (the top disjunction), subject to the condition that the order of w is smaller than that of r and there is no interfering $write(\_, x, \_)$ in between. Moreover, w itself must be concretely feasible, which is ensured by $\Phi_{cf}(w)$.

The size of Φ_{cf}, in the worst case, is cubic in the number of *read* and *write* events in τ. Nevertheless, in practice, the size of Φ_{cf} can be significantly reduced by taking $\prec$ into consideration. Consider two *write* events w_1 and w_2 in W^r_v. If $w_1 \prec w_2 \prec r$, we can exclude w_1 from W^r_v because it is impossible for r to read the value written by w_1 due to the read consistency axiom. Similarly, for any $w' \in W^r$, if $r \prec w'$, then w' can be excluded from W^r. Also, when constructing the constraints for matching an event $w \in W^r_v$ to r, if $w' \prec w$, then w' can be skipped.

3.3 Soundness and Maximality

Our race detection technique above is sound and maximal. Soundness means *every detected race is real*. Maximality means that *our technique does not miss any race that can be detected by any sound dynamic race detector based on the same trace*.

It suffices to prove the following:

THEOREM 3 (Soundness and maximality). *If Φ is the first-order constraint associated to a given trace τ as above, then Φ is satisfiable iff (a, b) is a race in τ in the maximal sense of Definition 4.*

PROOF: Suppose that $\tau = e_1 e_2 \dots e_n$. Note that $\rho \models \Phi$ for some $\rho : \{O_{e_1}, O_{e_2}, \dots, O_{e_n}\} \to \mathbb{N}$ iff $\rho' \models \Phi$ for a bijective $\rho' : \{O_{e_1}, O_{e_2}, \dots, O_{e_n}\} \to \{1, 2, \dots, n\}$. That is because the particular values assigned to the O variables are irrelevant, except for the race constraint $O_b - O_a = 1$, so we can find an ordering of $\rho(e_1)$, $\rho(e_2)$, $\dots \rho(e_n)$ such that $\rho(a)$ is followed by $\rho(b)$. Therefore, from here on we can only consider valuations of the form $\rho : \{O_{e_1}, O_{e_2}, \dots, O_{e_n}\} \to \{1, 2, \dots, n\}$. Any ρ yields the permutation $e_{\rho(O_1)} e_{\rho(O_2)} \dots e_{\rho(O_n)}$ of τ, which we write $[\rho]$.

It is easy to see that $\rho \models \Phi_{mhb}$ iff $[\rho]$ satisfies the must happen-before consistency requirements, and that $\rho \models \Phi_{lock}$ iff $[\rho]$ satisfies the lock mutual exclusion requirements. We can also show by induction on i that for any event e_i of τ with $op(e_i) \in \{branch, read, write\}$ and $t = thread(e_i)$, it is the case that $\rho \models \Phi_{cf}(e_i)$ iff $[\rho]_{e_i} \restriction_{t,read} = \tau_{e_i} \restriction_{t,read}$ and any read event in these trace projections satisfies the read consistency requirement in $[\rho]_{e_i}$: for *branch* and *write* events, the definition of Φ_{cf} reduces the property to previous *read* events, and for *read* events the definition of Φ_{cf} reduces the property to previous *write* events.

Let us first prove the soundness, that is, that if Φ is satisfiable then (a, b) is a race in τ. Let $\rho \models \Phi$. Then by the properties above and the definitions of $\Phi^{\approx}_{cf}$ and of Φ_{cf}, the following hold: $[\rho]$ satisfies the must happen-before and lock mutual exclusion consistency requirements; $[\rho]_b = [\rho]_a b$; and for all $e' \in \mathcal{B}_a$, if $t = thread(e')$ then $[\rho]_{e'} \restriction_{t,read} = \tau_{e'} \restriction_{t,read}$. We can then inductively build a trace τ_1 over data-abstract variants of the events in the set $\{e \mid e \prec a\}$, traversing them in the order they occur in $[\rho]$, as follows, where e is the next such event: if e is not a *read* or a *write* then append it to τ_1; if e is a *read* then to ensure read consistency we need to possibly change its value to the value written by the last event in τ_1 so far, and then append e to τ_1; if e is a *write* event then (1) if $\rho \models \Phi_{cf}(e)$ then append e to τ_1, otherwise (2) change the value of e to the symbolic value sym_{τ_1} and then append it to τ_1. All the steps above preserve the consistency of τ_1 and accord with the local determinism axioms characterizing $feasible(\tau)$, so we can deduce that $\tau_1 \in feasible(\tau)$. We can now extend τ_1 with (possibly data-abstract variants of) a and b similarly to the above, and thus obtain that $\tau_1 ab \in feasible(\tau)$, so (a, b) is a race in τ.

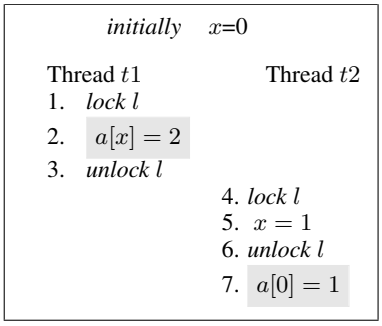
Let us now show the maximality, that is, that if (a, b) is a race in τ then Φ is satisfiable. Let $\tau_1 ab \in feasible(\tau)$ and let τ_2 be the trace formed with the remaining elements of τ, in the order in which they appeared in τ. Although the trace $\tau' = \tau_1 ab \tau_2$ may not be τ-feasible, it still respects the must-happen before and lock mutual exclusion consistency requirements. Let ρ be the valuation with $[\rho] = \tau'$. Then clearly $\rho \models \Phi_{mhb} \wedge \Phi_{lock} \wedge O_b - O_a = 1$. Since $\tau_1 ab$ is τ-feasible, prefix closedness ensures that $[\rho]_{e'}$ is also τ-feasible for each branch event $e' \in \mathcal{B}_a$; the local branch determinism axiom then implies that $[\rho]_{e'} \restriction_{t,read} = \tau_{e'} \restriction_{t,read}$, so by the property above and the definition of $\Phi^{\approx}_{cf}$ we conclude that $\rho \models \Phi^{\approx}_{cf}(a)$. We can similarly show $\rho \models \Phi^{\approx}_{cf}(b)$, so $\rho \models \Phi$. □

4. Implementation

We have implemented our technique in RVPredict, a runtime predictive analysis system for Java. Although the Java memory model (JMM [23]) is not sequentially consistent, it does not affect the soundness of our implementation as any race in a sequential consistency model should also be a race in JMM. To properly model the Java language constructs (i.e., to ensure that the Java execution conforms to our abstract model), we make the following treatments in our implementation:

- *branch* - the branch events include not only explicit control flow statements, but also implicit data flow points that can affect the control flow. For example, both shared pointer dereferences (e.g., calling a method of a shared object) and array indexing statements (e.g., read/write to an array with a non-constant index) are considered as additional branch events.

Consider this program executed following the order of line numbers. Lines 2 and 7 are unordered and they both access $a[0]$. However, (2,7) is not a race, because if line 2 is scheduled next to line 7, line 2 will access $a[1]$ instead of $a[0]$. Hence, we must ensure the same implicit data-flow for array accesses.



- *wait-notify* - Java's *wait*() and *notify*()/*notifyAll*() are usually not discussed in previous studies [18, 35]. In our implementation, we treat *wait*() as two consecutive *release-acquire* events, *notifyAll*() as multiple *notify*() where the number is equal to the number of currently waiting threads on the same signal, and keep a mapping from *wait*() to its corresponding *notify*() in the original execution. In the constraint, we ensure the order of the *notify*() is between that of the two consecutive *release-acquire* events of the corresponding *wait*(), but not between that of any other *wait*() on the same signal (to ensure that the *notify*() is matched with the same *wait*() as that in the original execution). Currently, we do not model spurious wakeups and lost notifications in our implementation. However, since they happen rarely in practice, this does not limit the usability of RVPredict.
- *re-entrant locking* - To simplify the constraint, re-entrant lock *acquire*/*release* events are filtered out dynamically in the execution, i.e., discarding all but the outermost pair of *acquire*/*release* events on the same lock.
- *volatile variables* - as concurrent conflicting accesses to volatile variables are not data races in Java, we do not report them.

RVPredict consists of two main phases: trace collection and predictive race analysis. In trace collection, we log a sequentially consistent trace of shared data accesses, thread synchronizations, and branch events. To support long running programs, traces are first stored event by event into a database. Note that trace collection can be performed at various levels, e.g., via static or dynamic code instrumentation, inside the VM, or at the hardware level. As trace collection is not our main concern here, our implementation is based on static instrumentation and is not optimized. Nevertheless, ideally, we can use hardware tracing techniques to minimize the runtime perturbation. In predictive race analysis, we first use a hybrid lockset and weaker HB algorithm (similar to PECAN [21]) to perform a quick check on each conflicting operation pair (COP). Only if a COP passes the quick check, do we proceed to build constraints for it.

To optimize the constraint solving, instead of adding a conjunction $O_b - O_a = 1$ for each COP(a, b), we simply replace O_a by O_b in the constraints. In this way, all constraints become simple ordering comparisons over integer variables, which can be solved efficiently using the Integer Difference Logic (IDL) (provided in both Z3 [11] and Yices [12]). We set the default constraint solving time to one minute for each COP. If the solver returns a solution within one minute, we report a race. In addition, to avoid redundant computation on races that have the same signature (from the same program locations), once a COP is reported as a race, we prune away all the other COPs with the same signature with no further analysis.

Handling long traces From an engineering perspective, handling long traces is challenging for any race detection technique. For real world applications, the trace is often too large to fit into the main memory. Moreover, for our approach, the generated constraints for long traces can be difficult to solve. Even with a high performance solver like Z3 or Yices, the constraints may still be too heavy to solve in a reasonable time budget. For practicality, we employ in RVPredict a windowing strategy similar to CP [35]. We divide the trace into a sequence of fixed-size windows (typically 10K events in a window) and perform race analysis on each window separately. This simple strategy has two advantages for performance optimization: First, each time only a window size of events are processed, which can be easily loaded in memory. Second, the generated constraints for a window instead of the whole trace become much smaller, so that Z3 and Yices can solve them much easier. The downside of this strategy is that a race between operations in different windows will not be detected. Fortunately, because the likelihood for two operations to race dramatically decreases when the distance between them gets larger, we did not find many such cases in practice. Moreover, this windowing strategy does not affect the soundness of our implementation. All detected races by RVPredict are real, i.e., it does not report any false positive.

5. Evaluation

Our evaluation aims to answer the following research questions:

1. *Race detection capability* - How many races can our technique detect in popular benchmarks and real world systems? As our technique is maximal, how many more races can it detect than the other state-of-the-art sound but non-maximal techniques?
2. *Scalability* - How efficient is our technique? Can it scale to real world executions?

To properly compare our technique with the state-of-the-art, we have also implemented HB [22], CP [35], and Said *et al.* [30] in RVPredict. We attempted to conduct an unbiased comparison and faithfully implemented the techniques according to their representative publications [22, 30, 35]. All our implementations are available at `http://fsl.cs.illinois.edu/rvpredict/`.

We evaluated these techniques on an extensive collection of widely used multithreaded benchmarks as well as several real world large concurrent systems, mostly from previous studies [5, 10, 17, 18, 21, 30, 35, 36]. To perform a fair comparison, for each benchmark, we collected one trace and ran different techniques on the same trace. To evaluate with long traces, because all techniques (including HB and CP) need the windowing strategy to scale, for all techniques and all benchmarks, we set the window size to 10K. This is sufficient to cover the traces of small benchmarks and at the same time to ensure that for large traces all techniques can finish within a reasonable time.

All experiments were conducted on a 8-core 3.50GHz Intel i7 machine with 32G memory and Linux version 3.2.0. The JVM is OpenJDK 1.7.0 64-Bit Server with 32G heap space. We next discuss our experimental results in detail as reported in Table 1.

Benchmarks and Traces Columns 1-2 list our benchmarks. The total source lines of code of these programs is more than 1.7M. The first row shows our example program in Figure 1; the second set of small benchmarks are from IBM Contest benchmark suite [17]; the third set contains three popular multithreaded Java Grande benchmarks; the last set contains real world large applications. The most substantial real systems include:

- FTPServer - Apache's high-performance FTP server;
- Jigsaw - W3C's web server;
- Derby - Apache's widely used open source Java RDBMS;
- Sunflow, Xalan, Lusearch, Eclipse - popular multithreaded applications from Dacapo benchmark suite 9.12 [5].

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Program	**LOC**	**Trace**					**Data race**					**Time**			
		#Thrd	**#Event**	**#RW**	**#Sync**	**#Br**	QC	**RV**	Said	CP	HB	**RV**	Said	CP	HB
example	64	2	20	12	6	2	1	**1**	0	0	0	0.4s	0.8s	0.8s	0.8s
critical	63	3	38	23	4	11	8	**8**	7	4	4	0.9s	0.8s	0.8s	0.8s
airline	83	11	214	127	10	77	9	**9**	6	8	8	0.9s	1.0s	0.8s	0.8s
account	87	3	126	82	2	42	9	**5**	5	3	3	0.9s	0.9s	0.9s	0.9s
pingpong	124	5	64	40	5	19	7	**4**	4	3	3	0.8s	0.8s	0.8s	0.8s
bbuffer	334	4	1.7K	975	209	497	15	**13**	10	5	5	4.1s	6.8s	1s	1s
bubblesort	274	26	4.5K	2.3K	188	2K	17	**8**	8	7	7	7.4s	44s	1.3s	1.2s
bufwriter	199	5	411	242	36	133	18	**2**	0	2	2	1s	1.2s	0.8s	0.8s
mergesort	298	5	5.5K	2.9K	122	2.5K	16	**9**	7	3	3	2.5s	7.9s	1.2s	1.2s
raytracer		2	23.4K	17.7K	674	5.3K	5	**5**	5	3	3	1.2s	3.4s	1.2s	1.2s
montecarlo	2.9K	2	11.7M	7.2M	285K	4.3M	0	**0**	0	0	0	13.5s	13.7s	14.2s	12s
moldyn		2	273K	193K	64	79.5K	509	**11**	5	2	2	53.5s	170s	1.2s	1.6s
Total: bench	4.3K	-					614	**75**	57	40	40	-			
ftpserver	32K	12	59K	40.3K	7.7K	10.7K	233	**37**	3	31	27	34s	96s	1.6s	1.3s
jigsaw	101K	12	4.6M	3M	15.5K	1.6M	54	**15**	14	7	7	81s	92s	6.2s	5.9s
derby	302K	3	856K	613K	38K	205K	469	**118**	15	14	12	1820s	1h	24s	23s
sunflow	109K	9	14.8M	7.7M	482	7.05M	66	**2**	2	2	2	14.6s	14.5s	14.4s	14.9s
xalan	180K	9	12.5M	6.9M	130K	5.5M	108	**108**	107	5	3	14.6s	14.6s	13.5s	12.5s
lusearch	410K	10	11.3M	6.7M	30.6K	4.6M	87	**16**	15	15	15	12.1s	12.8s	12.3s	11.9s
eclipse	560K	10	14.3M	7.6M	644K	6.05M	8	**3**	2	2	2	17.3s	23.6s	37.2s	14.5s
Total: real	1.7M	-					1025	**299**	158	76	68	-			

Table 1. Overall results - Columns 3-7 report metrics of the traces: the number of threads (*#Thrd*), total number of events (*#Event*), reads/writes (*#RW*), synchronizations (*#Sync*), and branch events (*#Br*). Column 8 (*QC*) reports the number of potential races that passes the quick check algorithm (a hybrid of lockset and weak HB). Columns 9-12 reports the number of real races detected by our technique (*RV*), Said *et al.* [30] (*Said*), Causally-Precedes [35] (*CP*), and Happens-Before [22] (*HB*), respectively, and Columns 13-16 report the total race detection time taken by the corresponding techniques. For all the evaluated programs, our technique detected more or at least the same number of races as the other techniques. For the efficiency, HB and CP are faster than the other two, and our technique is faster than Said.

Columns 3-7 report metrics of the collected traces. The traces cover a wide range of complexity. The number of events in the traces ranges from hundreds in small benchmarks to as large as 14.8M in real systems. For most real systems, the traces contain more than 10 threads. The number of read/write, synchronization, and branch events is significant in the real systems, ranging between 40K-7.7M, 0.5K-650K, and 200K-6M, respectively. We are not aware of previous sound predictive race detector implementations that have been evaluated on executions with such a large scale.

Bug Detection Capability Column 8 reports the number of potential races that pass the quick check of a hybrid lockset and weaker HB algorithm. These races comprise a superset of all the real races that can be detected from the trace. Because the hybrid algorithm is unsound, some races in this set may be false positives. For instance, there are 18 potential races detected in *bufwriter*, but only 2 of them are real races. Columns 9-12 report the number of real races detected by different sound techniques.

The results show that, for every benchmark, our technique is able to detect more or at least the same number of races (i.e., a super set) as the other sound techniques. For instance, for *derby*, our technique (*RV*) detected 118 races, while Said *et al.* detected 15, CP detected 14, and CP detected 12. This demonstrates that our technique achieves a higher race detection capability not only theoretically, but also in practice. For Said *et al.*, it detected more races than HB and CP in most benchmarks, with a few exceptions, though. For instance, for *ftpserver*, CP and HB detected 31 and 27 races, respectively, whereas Said *et al.* only detected 3. The reason for this, as explained in Section 1, is that the all read-write consistency prevents Said *et al.* from detecting races in feasible incomplete traces, though its SMT-based solution is able to explore more valid whole trace re-orderings than CP and HB. Between CP and HB, they detected the same number of races in the small benchmarks. This was because the lock regions in these small benchmarks typically have conflicting accesses. However, this does not hold for the real systems. In *ftpserver*, *derby*, and *xalan*, CP detected a few more races than HB.

For the real systems, our technique detected a total number of 299 real races. Notably, among these races, a number of them are previously unknown. For instance, we found three real races in *eclipse*, one is on the field variable *activeSL* of class *org.eclipse.osgi.framework.internal.core.StartLevelManager*, and the other two happen on the field *elementCount* of class *org.eclipse.osgi.framework.util.KeyedHashSet*. Interestingly, *KeyedHashSet* is documented as thread unsafe. The Eclipse developers misused this class and created a shared instance by multiple threads without external synchronization. Shortly after we reported these races, the developers fixed them and also contacted us for adopting our tool. Now the team is using RVPredict to detect races in the codebase of Virgo. We also found eight previously unknown races in *lusearch*, all of which happen in the class *org.apache.lucene.queryParser.QueryParserTokenManager*. We first reported these races in the *lucene* bug database. However, the developer pointed out that *QueryParserTokenManager* is documented as thread unsafe. It turned out that this class was misused by the Dacapo developers in writing the *lusearch* benchmark.

Note that our technique is sound and fully automatic. Unlike many unsound techniques that report false warnings or even sound techniques that require manual post-processing for most races (e.g., CP [35]), every race detected by our technique is real. This has been supported by our manual inspection: every reported race has been checked and confirmed to be real. On the other hand, because the maximality of our technique is concerned with sound race detection only, it is possible that our technique may miss some real races that can be reported by an unsound race detector. For example, not all the potential races reported in Column 8 are necessarily false alarms if they are not reported in Column 9 as well. However, if such a race exists, our technique guarantees that it cannot be reported by any sound technique using the same input trace. Note that any dynamic race detection technique (including ours) is sensitive to the observed execution trace. The results reported for different traces are incomparable. Therefore, it is possible for our technique

to miss certain races reported in other studies, because the traces in our experiments may be different from those used in other's work.

Scalability The performance of our technique largely depends on the complexity of the constraints and the speed of the constraint solver, as the core computation of our technique takes place in the constraint solving phase. With the high performance solvers and our windowing strategy, our technique shows good scalability when dealing with large traces. Column 13 reports the total time for our technique to detect races in each program using Yices. The performance of Z3 was comparable with only slight variances. For most small benchmarks, our technique was able to finish in a few seconds. For most real systems, our technique finished within around a minute. The most time consuming case is *derby*, which our technique took around 30 minutes to process. The reason is that the trace of *derby* has a lot more potential races (469 COPs) and also it contains many fine-grained critical sections (38K synchronizations), making the generated constraints much more complex.

Columns 14-16 report the race detection time for the other three techniques. Among the four techniques (including ours), HB and CP are comparable and are typically faster than Said *et al.* and our technique. This is expected because HB and CP do not rely on SMT solving and explore a much smaller set of trace re-orderings. Between our technique and Said *et al.*, our technique typically has better performance. For instance, for the *derby* trace, Said *et al.* took more than one hour (timeout) without finishing, while our technique finished within around 30 minutes. The reason is that our technique generates less constraints to solve than Said *et al.* for capturing the read-write consistency. While Said *et al.* generate constraints for all read events in the trace to ensure the whole trace read-write consistency, our technique concerns only the read events that have control flow to the race events.

6. Related Work

There is a rich body of race detection work in the literature. Our work is distinguished from other approaches in two main aspects: 1. we include control flow information - branch events - into the execution trace, enabling us to detect more races than other sound dynamic race detectors. 2. we encode the maximal causal model with control flow as a minimal set of feasibility constraints to achieve the maximal race detection capability. We next review important race detection techniques and discuss some representative recent work.

Predictive Trace Analysis Our technique belongs to the school of predictive trace analysis (PTA) approaches [10, 18, 21, 30, 36], which generate valid trace reorderings under certain scheduling constraints to find bugs unseen in the observed execution. The work by Said *et al.* [30] is representative in this direction and inspired our technique. Both techniques rely on efficient encoding of the trace constraints and modern SMT solvers to explore feasible reorderings. The key difference, as we noted earlier in Section 1, is that from the perspective of control flow our technique is able to encode the minimal set of feasibility constraints to achieve maximality. Without considering the control flow, [30] has to conservatively enforce the whole trace read-write consistency, which cannot detect races beyond fake control dependencies such as (3,10) in Figure 1 and also misses races in incomplete feasible traces such as (1,4) in Figure 2. ExceptioNULL [18] presents a PTA technique that predicts null-pointer dereferences using constraint solving. Similar to [30], it encodes the whole trace data-validity constraints and does not achieve maximality. jPredictor [10] is another representative PTA technique that predict races and atomicity violations based on sliced causality [9], which is a sound causal model concerning precise data or control dependencies. Differently, jPredictor requires expensive static dependence analysis (hard to implement soundly in practice) and it is non-maximal.

Runtime analysis Runtime detection techniques are typically designed for high runtime efficiency and do not perform comprehensive exploration of feasible trace permutations. Modern dynamic race detectors are often based on one or both of the lockset algorithm (first used in in Eraser [31]) and Lamport's happens-before (HB) principle [22]. Lockset-based approaches would detect all races detected by our technique but may also report many additional false alarms. It is unsound because many conflicting operations can be actually ordered by control flow even though they have different locksets. To address this problem, active testing techniques such as RaceFuzzer [32] create concrete executions to expose real races by actively controlling a race-directed thread scheduler. PECAN [21] statically generates racy schedules and uses a deterministic thread scheduler to create races. On the other hand, HB is precise but may miss races and is also more expensive. Numerous tools have been proposed to combine lockset with HB. For example, Choi and O'Callahan [28] investigate a two-phase scheme that first uses lockset analysis to find out problematic fields and then performs HB analysis to produce precise detection. Goldilock [14] uses lockset and HB to support continuous monitoring of race conditions in the JVM. To improve performance, FastTrack [19] proposes an adaptive lightweight representation for HB, and IFRit [13] uses static analysis to identify interference-free regions that reduce redundant instrumentations at compile time. Pacer [7], LiteRace [24], and DataCollider [16] use sampling-based approaches to detect races with negligible runtime overheads.

Static analysis Many whole program static analysis techniques have been developed for identifying races in various languages, including C [15, 29, 37, 39], Java [27], and SPMD programs [4]. The primary advantage of static detection is that they can potentially explore all paths to find possible bugs. However, applying static analysis to large and complicated programs without producing many false alarms is challenging. Language and type systems [6, 8] have also been proposed to statically prevent races. These approaches typically require programmer annotations to specify property regions or have a limited language expressiveness.

Model checking An alternative way to achieve maximality in detecting races is to exhaustively explore the thread scheduling space, employed by model checking techniques [26, 34, 38]. For instance, CHESS dynamically explores different thread schedules of the target program in a context-bounded way. Shacham et al. [34] use a model checker to construct the witness for races reported by the lockset algorithm. Unfortunately, facing the exponentiality of both the program path and the scheduling space, it is still hard for model checking techniques to scale to large multithreaded programs. As our technique focuses on exploring races with respect to a single dynamic trace, it is much more scalable than model checking.

7. Conclusion

We have presented a sound predictive race detection technique based on a new foundation of maximal causal model incorporating the control flow that achieves the maximal detection capability for any sound race detector given the same execution trace under sequential consistency. We formulate race detection as a constraint solving problem over a minimal set of valid trace reordering constraints, and use an SMT solver to find all real races captured by the new maximal model. We have conducted extensive experiments with our technique and demonstrate that our technique is not only theoretically sound but also practically feasible. It is effective and scalable in detecting races in real world large concurrent systems.

Acknowledgments

We would like to thank the FSL members and the anonymous reviewers for their valuable feedback on an early version of this paper. The work presented in this paper was supported in part by the NSF grant CCF-1218605, the NSA grant H98230-10-C-0294, the DARPA HACMS program as SRI subcontract 19-000222, the Rockwell Collins contract 4504813093, and the (Romanian) SMIS-CSNR 602-12516 contract no. 161/15.06.2010.

References

[1] `https://bugs.eclipse.org/bugs/show_bug.cgi?id=419383`.

[2] `https://bugs.eclipse.org/bugs/show_bug.cgi?id=419543`.

[3] `http://www.eclipse.org/virgo/`.

[4] A. Aiken and D. Gay. Barrier inference. In *POPL*, 1998.

[5] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, 2006.

[6] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel Java. In *OOPSLA*, 2009.

[7] M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: Proportional detection of data races. In *PLDI*, 2010.

[8] C. Boyapati and M. Rinard. A parameterized type system for race-free java programs. In *OOPSLA*, 2001.

[9] F. Chen and G. Roşu. Parametric and sliced causality. In *CAV*, 2007.

[10] F. Chen, T. F. Serbanuta, and G. Rosu. jPredictor: a predictive runtime analysis tool for Java. In *ICSE*, 2008.

[11] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, 2008.

[12] B. Dutertre and L. D. Moura. The Yices SMT solver. Technical report, 2006.

[13] L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H.-J. Boehm. IFRit: Interference-free regions for dynamic data-race detection. In *OOPSLA*, 2012.

[14] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware Java runtime. In *PLDI*, 2007.

[15] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP*, 2003.

[16] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *OSDI*, 2010.

[17] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. *IPDPS*, 2003.

[18] A. Farzan, P. Madhusudan, N. Razavi, and F. Sorrentino. Predicting null-pointer dereferences in concurrent programs. In *FSE*, 2012.

[19] C. Flanagan and S. N. Freund. FastTrack: Efficient and precise dynamic race detection. In *PLDI*, 2009.

[20] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 1990.

[21] J. Huang and C. Zhang. PECAN: Persuasive prediction of concurrency access anomalies. In *ISSTA*, 2011.

[22] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 1978.

[23] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL*, 2005.

[24] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: Effective sampling for lightweight data-race detection. In *PLDI*, 2009.

[25] A. Mazurkiewicz. Trace theory. In *Advances in Petri nets*, 1987.

[26] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.

[27] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI*, 2006.

[28] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *PPoPP*, 2003.

[29] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: Context-sensitive correlation analysis for race detection. In *PLDI*, 2006.

[30] M. Said, C. Wang, Z. Yang, and K. Sakallah. Generating data race witnesses by an smt-based analysis. In *NFM*, 2011.

[31] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multi-threaded programs. *TOCS*, 1997.

[32] K. Sen. Race directed random testing of concurrent programs. In *PLDI*, 2008.

[33] T. F. Serbanuta, F. Chen, and G. Rosu. Maximal causal models for sequentially consistent systems. In *RV*, 2012.

[34] O. Shacham, M. Sagiv, and A. Schuster. Scaling model checking of dataraces using dynamic information. In *PPoPP*, 2005.

[35] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound predictive race detection in polynomial time. In *POPL*, 2012.

[36] F. Sorrentino, A. Farzan, and P. Madhusudan. PENELOPE: Weaving threads to expose atomicity violations. In *FSE*, 2010.

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

[38] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java pathfinder. In *ISSTA*, 2004.

[39] J. W. Voung, R. Jhala, and S. Lerner. RELAY: Static race detection on millions of lines of code. ESEC-FSE, 2007.