IPA: Improving Predictive Analysis with Pointer Analysis

Peng Liu¹²
liup@us.ibm.com

Omer Tripp^{3*}
trippo@google.com

Xiangyu Zhang¹
xyzhang@cs.purdue.edu

¹Purdue University
United States

²IBM Research
United States

³Google
United States

ABSTRACT

Predictive analysis, recently proposed for race detection, guarantees to report no false positives and achieves good coverage. Predictive analysis starts with the trace of an execution and mutates the schedule order of the trace to “predict” the executions that expose the hidden races.

Ideally, the predictive analysis should allow the schedule mutation to change the memory location accessed by the field access, which helps meet the “same memory location” requirement of the data race. However, existing predictive approaches, including causality-preserving approaches and symbolic approaches, lack this capability.

We propose the first predictive analysis that allows changing the accessed locations. The key challenge is that modeling of the field accesses relies on the location, which may however become unknown due to schedule mutation. We solve this challenge through a novel combination of predictive analysis and pointer analysis. Furthermore, unlike previous work, our analysis applies a hybrid encoding scheme to increase practical applicability.

We have implemented our approach as a prototype IPA, and compared it against the most recent predictive analysis over a set of popular Java applications. Our experimental evaluation confirms the effectiveness of our approach: IPA is able to find close to 2X as many races as previous approaches.

CCS Concepts

•Software and its engineering → Software testing and debugging;

Keywords

Predictive analysis, concurrency, data race detection, causality model, symbolic analysis, constraint solver, thread scheduling

*Work performed while the author was employed by IBM.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ISSTA’16, July 18–20, 2016, Saarbrücken, Germany
© 2016 ACM. 978-1-4503-4390-9/16/07...\$15.00
<http://dx.doi.org/10.1145/2931037.2931046>

1. INTRODUCTION

Data races underlie many concurrency bugs, which cause exceptional program behaviors. A data race (or for short, a race) occurs when different threads concurrently access the *same memory location*, and at least one of the threads writes to it.

Predictive analysis [26], a novel form of race detection analysis, has been recently proposed. Predictive analysis starts with an execution trace (*seed execution*), and mutates the schedule order of the trace to “predict” the executions (*predicted executions*) that expose the hidden races. Contrary to lockset-based analyses [22, 15], which often report false positives, predictive analysis guarantees to report only true positives. Researchers [26] refer to this guarantee as *soundness*. Compared to the happens-before-based analyses [16, 5, 1], predictive analysis offers better coverage of data races by allowing more schedule mutations. Our work belongs to the category of predictive analysis.

The schedule mutation may cause the shared read $x=S$ to read a different value. If the read value is an object reference, it will cause the field access $x.f$ to access a different memory location. In such a case, we would like the predictive analysis to allow the schedule mutation to change the memory location accessed by the field access. Given a pair of field accesses of different locations in the seed execution, the analysis may change them so that they access the *same location* and form the data race in the predicted execution. We refer to such a race, whose manifestation requires the change of the accessed locations, as a *latent race*. Surprisingly, existing predictive analyses [26, 7, 19, 27] *cannot* change the location accessed by the field access, thereby missing the latent races.

We illustrate the importance of this feature via a real-world example. Figure 1 shows the trace of an execution of the application *ftpserver*. In the seed execution, event 9 accesses the location $o_2.filter$ because event 7 reads o_2 from event 4 into the reference variable y . This implies that event 9 accesses a location other than that accessed by event 2 ($o_1.filter$). According to the definition of the race, events 2 and 9 cannot form a race. Nonetheless, they can form a race after a schedule mutation. If we mutate the schedule so that the events 6-9 are immediately executed after event 1, event 7 would read a new value o_1 into the reference variable y , and hence, event 9 would access a new location $o_1.filter$. As a result, events 2 and 9 access the same location and form a race. This is a latent race.

Existing Approaches Many designs of predictive analysis [26, 7, 19, 27] have been proposed, of which the goal is

session.nextEntry = ...; // o ₁	
T ₁	T ₂
1: x=session.nextEntry; // o ₁	
2: x.filter=...; // o ₁ .filter	
3: lock (session);	
4: session.nextEntry=o ₂ ;	
5: unlock (session);	
	6: lock (session);
	7: y=session.nextEntry;
	8: unlock (session);
	9: ...=y.filter; // o ₂ .filter

Figure 1. A trace fragment of ftpserver

to allow as many schedule mutations as possible while ensuring soundness. They can be classified into two categories: *causality-preserving* approaches and *symbolic* approaches.

Causality-preserving approaches [26, 7] need to preserve the *causal order*, which is the happens-before (HB) order that results in the data dependence.¹ Consider Figure 2. The HB order between events 3 and 4 in (a) is a causal order because it guarantees the data dependence between events 2 and 5, while the HB order in (b) is not a causal order. Compared to the HB-based approaches [16, 5, 1], which preserve all the HB orders, the causality-preserving approaches preserve only a subset of the HB orders (i.e., the causal orders), thereby allowing more schedule mutations. By preserving the causal orders, the causality-preserving approaches guarantee that the shared accesses access the same values as in the seed execution and further guarantee soundness.

Clearly, these approaches *cannot* change the location accessed by the field access `x.f`, as they cannot change any read value including the value of the reference `x`. The state-of-the-art analysis tool RVPredict [7] reduces the causal orders that need to be preserved during the schedule mutation. However, RVPredict also *cannot* change the accessed location. Specifically, RVPredict treats the field access as the branch event,² and disallows any change to the branch event [7]. Consider Figure 1. The causality-preserving approaches [26, 7] preserve the casual order between events 5 and 6 as well as the dependence between events 4 and 7. Therefore, they force event 9 to access `o2.filter` in the predicted execution, and so the race between events 2 and 9 is missed.

Different from the causality-preserving approaches, the symbolic approaches [19, 27] allow changing the read values if the values are of the *primitive* type. Figure 3 illustrates the difference. The causality-preserving approaches preserve the causal order between events 4 and 5, which prevents events 1 and 8 from being concurrently executed, thereby missing the race between them. The symbolic approaches do not need to preserve the causal order. By re-scheduling the lock region 5-9 before the lock region 2-4, they allow event 6 to read a different value (i.e., the initial value 1) and apply the solver to account for the feasibility of the downstream execution (e.g., the feasibility of the true branch at event 7). In this example, the solver confirms that the predicted execution, which exposes the race between events 1 and 8, is

¹Data dependence includes flow dependence, anti-dependence and output dependence.

²See the implementation section of RVPredict.

Figure 2. (a) HB order is a causal order. (b) HB order is not a causal order. The arrow denotes the HB order.

Figure 3. The symbolic approaches can find the race between events 1 and 8, while the causality-preserving approaches cannot.

feasible. Note that only the value of the primitive variables is changed.

The existing symbolic approaches [19, 27] do not support the change of the location accessed by any field access. They assume each field access accesses the same location as in the seed execution. Following the *assumption*, they hardcode the location information in the symbolic encoding (i.e., the concurrent SSA representation) of the field access. As a consequence, the existing symbolic approaches *cannot* find the latent race, which requires the change of the location at some field accesses, such as the race in Figure 1.

Our Approach Given that symbolic approaches are less constrained than the causality-preserving approaches, we propose a symbolic approach. In contrast with existing symbolic approaches, our approach supports change of the locations accessed by the field accesses by lifting the aforementioned assumption. By changing the heap references `x` and `y`, the analysis may let the field accesses `x.f` and `y.f` access the same location and form the necessary condition of the race. Through this support, our analysis enables more schedule mutations. According to recent studies [21, 20], as well as our experiments (Section 6), the trace usually includes a large number of field accesses, implying that the support may have a dramatic effect on race detection, as confirmed by our evaluation.

It is unclear what location each field access refers to after the schedule mutation. However, without the location information, the field accesses, as well as the data race and the data flow between them, cannot be encoded. We solve this

challenge through a *novel* combination of predictive analysis and pointer analysis. We first apply the pointer analysis to the trace program [8], a simple program generated from the trace, to compute the locations that each field access *may* point to; we then condition the encoding of the field access on every location in the point-to set.

Another problem with the existing symbolic approaches is that they assume the solvers support all types of operations. The assumption is impractical, e.g. today’s solvers cannot support the `lengthof` operation of the String. To make our approach applicable to real-world applications, we propose a hybrid symbolic encoding scheme similar to the *concolic execution* [2]: For operations supported by the solver, we encode such operations symbolically. For operation not supported by the solver, we require all the operands to take the concrete values observed in the seed execution.

We implemented a tool called IPA, and conducted our experiments over 11 benchmarks including large applications such as `tomcat`. The results show that our approach detects 2X as many races as the most recent work *RVPredict* [7] does, while incurring a tolerable slowdown of the analysis time (e.g., IPA finishes within 16 min). We believe IPA represents a good tradeoff between the effectiveness and the performance.

In summary, this paper makes the following contributions:

- We propose IPA, the first predictive analysis that permits schedule mutations that change the location accessed by a shared access. The feature is achieved through a novel combination of the predictive analysis and the pointer analysis.
- Unlike existing symbolic predictive analyses, our analysis takes the solver’s limited expressive power into account and applies a hybrid symbolic encoding scheme to increase its applicability to real-world applications.
- We conducted experiments over 11 benchmarks, including large-scale applications such as `tomcat`. The results show that IPA is significantly more effective than existing predictive analyses.

2. TECHNICAL OVERVIEW

This section presents an overview of our technique. We first lay out the formal model underlying our analysis (Section 2.1), then illustrate our analysis by applying it to a real-world example (Section 2.2).

2.1 Formal Model

The formal model states the conditions for feasible schedule mutations, under the general assumption — generally made by all existing predictive analyses — that individual threads proceed along the same path as in the seed run. Note that the model (1) allows changing the location accessed by the field access and (2) takes the solver’s limited expressive power into consideration. In the following, we sometimes use the terms *schedule mutation* and *trace transformation* interchangeably.

- (**Strict Consistency**) The value read by an event may change after the schedule mutation, but it should be consistent with the value written to the variable or the location by the latest write. Besides, events originating from the same thread cannot be reordered.

- (**Event Semantics**) The trace transformation should respect the semantics of the statement executed by the event. For example, in $x = y + z$, if the read value of y or z changes, then the value of x should change accordingly. *If the solver cannot reason about the operation, any value change in the event should be prohibited.*
- (**Synchronization Semantics**) The trace transformation needs to respect the semantics of the synchronization primitives, e.g., the `start` primitive always happens before the events of the child thread.

Strict consistency [25] is stronger than sequential consistency [10], linearizability [6] and many other consistency models. Strict consistency has multiple implications. First, the schedule mutation may change the latest write prior to a shared read, forcing the read to obtain a different value. Second, the difference is further propagated by the thread-local assignments. Third, *if the heap reference of a field read changes, the field read should read from a different location.* Consider Figure 1, the schedule mutation changes the reference `y` in event 9 from o_2 to o_1 , therefore, after the schedule mutation, event 9 should read from the location o_1 .`filter`, instead of o_2 .`filter`.

2.2 Constraint System

We translate the conditions in the formal model about the feasibility to a set of constraints and reduce the feasibility check to the constraint solving. As explained before, the schedule mutation may induce the change of the value of each variable (primitive variable or reference variable). Let e_i denote the i th event in the seed run. We introduce the symbol P_i to denote the schedule order of event e_i after the schedule mutation. The solver will assign the integer values to the symbol P_i or P_j , and $P_i < P_j$ means e_i will be scheduled earlier than e_j after the mutation. Besides, given variable x or field expression $x.f$ involved in event e_i , we use x^i or $x.f^i$ to denote its value after the change is induced.

Race in ftpserver Consider Figure 1. It requires changing the location accessed by event 9 to expose the data race. Through a pointer analysis (we shall explain it shortly), we first find that `y` in event 9 and `x` in event 2 may point to the same object. Therefore, we treat events 2 and 9 as a candidate race pair. We check if they can form a real race by constructing and solving the constraints.

Following the formal model, we specify a set of constraints.

- (**Strict Consistency**) The reference values read by events 2 and 9 should be consistent with the values updated by their latest writes.

$$\text{assert} : x^2 = x^1 \wedge y^9 = y^7$$

For a field access $x.f$ in event e_i , two symbols x^i and $x.f^i$ are maintained to represent the values of the reference and the field expression, respectively. *The value of $x.f^i$ is conditioned on the value of x^i .* This implies that, once the value of x^i is known, the value of $x.f^i$ can be expressed. However, the value of x^i may become unknown due to the schedule mutation. To address the problem, we apply the pointer analysis to compute the finite domain of the value of x^i and condition the value of $x.f^i$ on every possible value in the finite domain.

Consider the shared read at event 7, it may read from the initial definition (Let us denote it as event e_0) or read from event e_4 . In the following, we encode the former case only. The latter case can be similarly encoded. The encoding of the two cases should be disjointed, meaning that either case is possible.

```

assert :      session7 = session0
             session.nextEntry7 = session.nextEntry0
             ∧ P0 < P7
             ∧ ((P4 < P0 ∨ P7 < P4) ∨ session7 ≠ session4)
             ...

```

Event 7 reads the value of `session.nextEntry` from event 0 iff (1) the events involve the aliased heap reference, i.e., the events access the same location, and (2) the def-use relation between them is not interfered with by another definition of the same location (e.g., event 4).

- **(Event Semantics)** The trace transformation needs to respect the semantics imposed by each event. Take event 7 for example:

```
assert : y7 = session.nextEntry7
```

- **(Synchronization Semantics)** The trace transformation needs to respect the synchronization semantics. Consider the lock regions 3-5 and 6-8:

```
assert : session3 = session6 ⇒ (P5 < P6 ∨ P8 < P3)
```

The mutual exclusion is conditioned on that the two lock regions are guarded by the same lock object.

Finally, we specify the race condition between events 2 and 9, which requires the aliasing of the heap references:

```
assert : x2 = y9 ∧ P9 = P2 + 1
```

By combining all the constraints and solving them with a solver, we calculate the new schedule that exposes the race: 1, 6, 7, 8, 2, 9.

In this example, all reference variables named `session` refer to the same object. However, in general, they may refer to different objects, making the constraint encoding even more complex. Therefore, a principled encoding method is needed, which we present in Section 4.

Now we come back to the pointer analysis mentioned above. A simple idea is to apply traditional pointer analysis [18] to the trace program, i.e. the statements involved by the trace events. However, it has been shown that the complexity of flow-sensitive schedule-aware pointer analysis is $O(n^4)$, which is prohibitively expensive. We propose to impose a practical restriction, dubbed **bounded heap writes**, to lower the complexity of pointer analysis. Roughly speaking, if we fix the location accessed by each field write as the location accessed in the seed run, a linear scan of the trace, which only reasons about the assignments and field reads, suffices to propagate the point-to set. We explain the details in Section 4.1.

3. FORMAL MODEL

The formal model states the conditions for feasible schedule mutations. The formal model is parameterized with the solver and takes the solver's expressive power into account. Another important property of the model is that it allows changing the location accessed by the field access.

$e ::= x = v$	(Initialization)
$x = y$	(Assignment)
$z = x \oplus y$	(Arithmetic)
$x = y.f$	(Field Read)
$x.f = y$	(Field Write)
$\text{if}(b) \text{ then } e \text{ else } e$	(Branch)
$\text{lock}(l) \mid \text{unlock}(l)$	(Sync)
$\text{start}(t) \mid \text{join}(t)$	
$\text{wait}(l) \mid \text{notify}(l)$	

Figure 4. Language syntax

3.1 Language Syntax and Notation

Language. For the formal discussion, we utilize the core language in Figure 4. All the statements have their standard meaning, where $\oplus$ denotes the standard binary arithmetic operations (+, ×, etc). For simplicity, we have omitted certain syntax, e.g. unary operations and loops.³

We assume a standard operational semantics, which consists of the following domains:

$T \subseteq \mathcal{T}$	Threads
$O \subseteq \mathcal{O}$	Objects
$C \subseteq \mathcal{C}$	Constants
$V \subseteq \mathcal{V} = \mathcal{O} \cup \mathcal{C}$	Values
$L \in \mathcal{L} = \mathcal{O} \times \text{FldId}$	Heap Location

At any given event, a finite set $O (\subseteq \text{domain } \mathcal{O})$ of objects is live. We treat the privileged reference value `null` as a constant. `FldId` denotes the set of field identifiers in the program.

Events and Traces. We refer to the runtime execution instance of a statement as an *event*. Each event e has a unique id i , which does not change during the transformation⁴. We refer to the event as e_i . We use t_e and l_e to denote the thread that executes e and the statement executed by e , respectively. An event involves the right-hand-side (RHS) arguments, and if applicable, the left-hand-side (LHS) variable. The event can be abstracted as a map, which maps every variable involved by the event to a value. For a variable x , we use $e(x)$ to get the value of x , and use $e[x \mapsto v']$ to update the value of x in the map.

Given program P , a *trace* of P is a sequence of events produced by the execution of P . The trace τ supports the following operations: $\tau \cdot e$ concatenates τ and e , τ_t projects τ onto a thread t . Besides, $\tau(x)$ or $\tau(o.f)$ returns the value of the variable x or the location $o.f$ (o is an object) at the end of the trace τ .

3.2 Trace Transformation

Given a seed trace τ of the program, our predictive analysis attempts to transform τ into a new trace τ' that is feasible and exhibits a data race. In general, the trace transformation induces changes of the values of variables. Theorem 1 presents the theory underlying our predictive analysis. It guarantees that the transformed trace is feasible if the induced changes satisfy certain conditions. We first present the formal definitions that are used by Theorem 1.

³Loops are unrolled in the trace.

⁴In our implementation, we use the offset of e in the seed trace as the id.

DEFINITION 1 (SOLVABILITY). *Given solver S and statement st that declares an LHS variable x , st is solvable by S , denoted as $S \downarrow st$, if S can compute the value of x given the concrete values of the RHS arguments.*

An event e in τ is transformed into e' in τ' by the schedule mutation. The events e and e' involve the same identifier, the same statement and the same thread. We define this relationship below.

DEFINITION 2 (ISOMORPHISM). *Events e and e' are isomorphic if they are the same (i.e., involving the same statement, the same identifier and the same thread) except for the concrete LHS/RHS values.*

By extension, we say the traces τ and τ' are isomorphic if the events at each entry are isomorphic.

THEOREM 1 (FEASIBILITY). *Given feasible traces $\tau \cdot e$ and τ' and solver S , let t denote t_e . Assume that τ_t and τ'_t are isomorphic. Then $\tau' \cdot e'$ is feasible if e' satisfies the following criteria:*

- e is an initialization event, and $e' = e$ (i.e., the events are identical).
- $l_e : x = y$, and $e' = e[x \mapsto \tau'(x), y \mapsto \tau'(y)]$.
- $l_e : z = x \oplus y$, $S \downarrow z = x \oplus y$, and $e' = e[x \mapsto \tau'(x), y \mapsto \tau'(y), z \mapsto \llbracket \tau'(x) \oplus \tau'(y) \rrbracket]$.
- $l_e : x.f = y$, and $e' = e[y \mapsto \tau'(y), x \mapsto \tau'(x), \tau'(x).f \mapsto \tau'(y)]$.
- $l_e : x = y.f$, and $e' = e[y \mapsto \tau'(y), y.f \mapsto \tau'(\tau'(y).f), x \mapsto \tau'(\tau'(y).f)]$.
- $l_e : \text{if}(z) \dots$, and $e' = e$.
- e is a sync event with the argument x , $e' = e[x \mapsto \tau'(x)]$ and $\tau'e'$ should respect synchronization semantics.

PROOF SKETCH 1. *Since the conditions in Section 2.1 guarantee the correctness [6], here we simply prove that Theorem 1 is equivalent to the conditions in Section 2.1. We sketch the proof below. First, the general assumption stated in Section 2.1 is satisfied by Theorem 1, which requires the branch event to take the same branch decision as in the seed run.*

For the variable x involved in each event (e.g. the variable in the assignment or the reference in the field access), the strict consistency is respected by always reading the latest written value $\tau'(x)$. For the field read event, $x=y.f$, it preserves the strict consistency by reading the latest value $\tau'(o.f)$ of the location $o.f$, where o is the latest object that y references. The thread-local execution order is preserved because the transformation always appends the thread-locally next event from the seed run to the thread. The semantics of every event is preserved, with the solver's expressive power taken into account. Consider the arithmetic statement. If $S \downarrow z = x \oplus y$, then we let the solver compute z , which preserves the semantics of $\oplus$; Otherwise, we conservatively force the variables to take their original values (omitted above), which also preserve the semantics of $\oplus$. The synchronization semantics needs to be preserved as well.

We refer to the above transformation as **appending**. Since the prefix of a feasible trace is also feasible, we can apply the appending transformation to any prefix of τ . Besides, we can *iteratively* apply the appending transformation if it preserves the isomorphism (the premise of Theorem 1). We prove that this condition holds universally.

LEMMA 1. *Given feasible trace τ , let τ' be a trace predicted from τ following Theorem 1. Then for each thread t executing in τ , τ_t and τ'_t are isomorphic throughout their common prefix.*

PROOF SKETCH 2. *Appending guarantees that the pair of events e and e' appended to a given thread t are isomorphic. By induction, appending guarantees isomorphism at every entry. Besides, the prefix trace and the original trace are isomorphic thread-wise at each entry.*

4. CONSTRAINT SYSTEM

In this section, we instantiate a practical constraint system on top of the formal model in Section 3. Without loss of generality, we assume the solver implementation supports only the basic arithmetic operation and the equality check operation. Most off-the-shelf solvers support them. The constraint system checks the criteria of the formal model by solving a set of constraints.

The trace transformation changes the schedule order of the events and induces the changes of the values. We use symbolic variable P_i to denote the predicted schedule order of e'_i in the transformed trace τ' , and use the symbolic variable x^i or $x.f^i$ to denote the value of x or $x.f$ in the event e'_i . The constraint system constructs the constraints over the symbolic variables and solves the constraints. If a solution is found, then that means that the transformed trace is feasible and the found race is sound.

We construct the constraints per the criteria in Theorem 1, and defer the encoding of the field accesses to Section 4.1. Note that Section 4.1 presents one of our novel design features, while the rest of Section 4 is simply a reiteration of previous work (We show it for comprehensiveness).

Initialization For initialization event e'_i with statement $x = v$. We have the constraint **assert** : $x^i = v$.

Simple Assignment Consider event e'_j involving statement $x = y$. Suppose event e'_j is the latest write of y . The two events should agree on the value of y . Besides, e'_i should ensure the equality between x and y . This yields the constraint, **assert** : $y^j = y^i \wedge x^i = y^i$. The latest write e'_j can be easily derived because it is isomorphic to the latest write in the seed run (Lemma 1). Intuitively, each thread takes the same path and executes the same sequence of statements in both traces.

Arithmetic Consider event e'_i involving statement $z := x \oplus y$ and thread t . If $\oplus$ is solvable by the solver, we specify the constraints to encode its behavior, as well as the consistency with the latest definitions. Denote the most recent definitions of x and y by thread t as e'_j and e'_k . Then we have the constraint **assert** : $z^i = x^i \oplus y^i \wedge x^i = x^j \wedge y^i = y^k$.

If $\oplus$ is not supported by the solver, we require x , y and z to retain their original values in the seed trace, $e_i(x)$, $e_i(y)$ and $e_i(z)$ respectively, which should further be consistent with the latest definitions. We thus have the constraint **assert** : $z^i = e_i(z) \wedge x^i = e_i(x) \wedge y^i = e_i(y) \wedge x^i = x^j \wedge y^i = y^k$.

Branch Consider a branch event e'_i with respective statement **if**(z) ... and thread t . The assumption of the formal

model is that the thread in the predicted run takes the same branch decision as in the seed run. Therefore, we require z to evaluate to the same value as $e_i(z)$, which should also be consistent with the latest definition e'_j by t . We have $\text{assert} : z^i = e_i(z) \wedge z^i = z^j$.

Thread-local Order Theorem 1 requires each transformation step preserves the thread-local order since it always append the thread-locally next event from the seed run. Consider the projection of trace τ' onto each thread t , denoted τ'_t . Let $[e'_1, e'_2, \dots, e'_n]$ be the sequence of events in τ'_t . Then we impose the respective constraint $\text{assert} : P_1 < P_2 < \dots < P_n$.

Synchronization Synchronization events include lock/unlock events, which specify the mutual exclusion constraints, and the start/join/wait/notify events, which specify the **must happens-before order**, e.g. the start event should happen before the first event of the child thread. A lock region starts with a lock event and ends with a matching unlock event. Two lock regions from two threads guarded by the same lock cannot overlap. One should happen completely before the other. Thus, given a lock region with l_{ij} as the lock variable and delimiting events (e'_i, e'_j) , and another lock region with l_{mn} as the lock variable and delimiting events (e'_m, e'_n) , the values of the lock variables are denoted l^i_{ij} and l^m_{mn} , respectively. We formulate the following constraint: $\text{assert} : l^i_{ij} = l^m_{mn} \Rightarrow P_j < P_m \vee P_n < P_i$. Note that the mutual exclusion constraints are imposed only when the lock regions share the same lock.

4.1 Field Accesses

We now turn to the encoding of field reads and field writes, which needs to resolve the reference variable x and the field expression $x.f$. We address the challenge in the following. At a high level, we first apply the pointer analysis to compute the point-to set of the reference variable, then encode the field accesses based on every possible object in the point-to set, and at last, combine the encodings for all the possibilities.

Pointer analysis The schedule mutation may force the variable x in $x.f$ to change the object it references. Pointer analysis can be applied to the trace program [8] to compute the point-to set of every variable x . The trace program is a sequence of statements executed by the events in the trace.

However, it has been shown [18] that the complexity of flow-sensitive schedule-aware pointer analysis is $O(n^4)$, where n is the number of events in the trace. Given that n is typically large in our setting, we propose a restriction, which we dub *bounded heap writes*, to lower the complexity of pointer analysis.

The bounded heap writes restriction requires that, given any shared write e , $x.f = y$, both x and y retain their original values in the seed trace, respectively. These values should also be consistent with the latest writes, which we omit hereafter for simplicity. The same restriction has been adopted for improving the scalability of the symbolic execution [3].

Given this restriction, we then scan the trace, in a linear fashion, to build up the point-to set at two types of events: simple assignment and field read.

- If the event e involves a simple assignment $x = y$, then we maintain $pts(x, e) = pts(y, e)$, where $pts(x, e)$ refers to the point-to set of x in e . The point-to set of the LHS variable x is directly copied from the latest definition

of y , where the latest definition can be easily identified during the trace preprocessing (Section 5).

- For a field-read event e involving statement $x = y.f$, we maintain

$$pts(x) = \bigcup_{o \in pts(y)} lookup(o.f)$$

, where $lookup(o.f)$ looks up the bounded heap writes for all the values that heap location $o.f$ may point to. We assume the heap writes are already indexed by heap locations, and so the lookup for each heap location takes $O(1)$ time.

Reading from the latest writes Although the field writes are fixed, the field read may choose to read different values from different writes. As per Theorem 1, the field read should read from the heap, which is written to by the latest write. After the schedule is mutated, the read may read from a different update. The details are explained in the following.

DEFINITION 3 (READS-FROM RELATION). *Read event e'_i with statement $x = y.f$ reads from the write event e'_j , denoted $e'_j \rightsquigarrow e'_i$ if the following conditions are met:*

1. The memory location $o.f$ read by e'_i (where o is the object referenced by y) is also written to by e'_j .
2. e'_i is scheduled after e'_j .
3. Any other write event e'_k to the same heap location occurs either before e'_j or after e'_i , so that e'_k does not interfere with the data flow between e'_j and e'_i .

The corresponding constraints are specified below. Given the read e'_i and the respective statement $x = y.f$, we enumerate every object o that y may reference in e'_i . For each object o , we establish the reads-from relation between the read event and the writes to the location $o.f$ below. The reads-from relations based on the objects are disjointed to union all the possibilities.

Suppose the write is denoted as e'_j in the form $w.f = z$, we require the conditions in Definition 3: (1) the object o referenced by the read is also referenced by the write, i.e., the write event updates $o.f$; (2) e'_j occurs before the read event (with index i): $P_j < P_i$; and (3) any other write event e'_k of the same location $o.f$ either occurs before the write event ($P_k < P_j$) or after the read event ($P_i < P_k$). Thus we have the following clause, where the symbol *writes* stands for the writes of the location $o.f$:

$$\bigvee_{e'_j \in \text{writes}} \quad y^i = o$$

$$\bigwedge \quad y.f^i = w.f^j$$

$$\bigwedge_{e'_k \in \text{writes} \setminus \{e'_j\}} \quad (P_j < P_i \wedge (P_k < P_j \vee P_i < P_k))$$

The final step is to identify the candidate race pairs and specify the race condition. A pair of events, e'_i and e'_j , form a race iff (1) e'_i and e'_j access the same location ℓ , (2) at least one of them writes to ℓ , and (3) e'_i and e'_j run concurrently. Following the standard single-observer model [26], (3) can also be stated as, e'_i and e'_j are adjacent in the schedule.

Identification of candidates is based on intersecting the point-to sets of the reference variables involved in the field accesses. If the point-to sets overlap, we treat the field accesses as the candidate. For each candidate pair, (e'_i, e'_j) , we check if the pair can form a real race. Suppose e'_i and e'_j access fields $x.f$ and $y.f$ respectively. The values of the reference variables are denoted as x^i and y^j . We have the constraint, $\text{assert} : x^i = y^j \wedge P_i = P_j + 1$

We combine all the constraints constructed above and discharge them to an SMT solver. The solver may report SAT, meaning that the race is feasible, and output the schedule needed to trigger the trace as well as the corresponding program state. Alternatively, the solver may report UNSAT, meaning that the race is not feasible (i.e., none of the predicted runs, as qualified by the constraints, exhibits the race).

5. IMPLEMENTATION

We have implemented our approach as the prototype system IPA.⁵ IPA accepts as input (i) a concurrent program P and (ii) an input to P . IPA outputs data races confirmed to occur the program as well as the schedules exposing the reported races.

Figure 5 presents the workflow of IPA. The **trace recorder** component collects the trace by inserting monitoring code to the program and executing the instrumented program. The instrumentation is implemented using the **Soot** compiler at the three-address intermediate-representation (IR) level. The trace is persisted to a database, where each table hosts a given type of events. An event identifier is also generated and associated with each event during this phase.

Next, the **preprocessor** loads the trace from database to memory, encodes the trace into SSA form, and computes a pointer-analysis solution over the trace. SSA form distinguishes multiple definitions of the same variable as multiple SSA variables, each of which corresponding to a particular definition. SSA form is required for constraint encoding, because the solver assigns exactly one value to each variable.

The pointer analysis computes the objects referenced by a given (SSA) variable (under the assumption that the schedule does not impose schedule order constraints on field reads/writes). The point-to set extends field accesses, such that they refer to additional possible locations. Specifically, we need to link the uses with their latest definitions such that the SSA variable and the point-to set are propagated along the link. We find the definition of a use via backward scanning of the thread-local trace, where the definition of a variable with the same name is searched.

The **candidate selector** forms the candidate race pairs by picking field accesses referencing the same location (including those introduced by the pointer analysis). The candidate race pairs are pruned by

- the **HB filter**, which filters a candidate pair out if there exists a happens-before order between the pair elements; and
- the **lockset filter**, which filters a candidate pair out if both accesses are guarded by the same lock.

Figure 5. Software Implementation

Specifically, in the HB filter, we collect the thread-local order as well as ordering imposed by **start/join/wait/notify**. We then compute their transitive closure.

Given a candidate race pair, we either confirm it or reject it by applying the **constraint encoder** and **constraint solver** to encode/solve the constraints. For this purpose, we use the popular **z3** solver. Essentially, we translate the constraints in Section 4 to the **z3** constraint format. In our implementation, we have defined APIs that abstract away the details of the **z3** format to help express the desired logic more naturally. Finally, IPA invokes the **z3** solver by executing an **ant** build file.

In the following, we explain subtle challenges that we addressed in the implementation:

Type Mismatch Some types of program variables cannot be expressed by the solver, and in particular (custom) user-defined types. To solve this problem, we map each object o (or constant c) to a unique integral value int_o for the solver. This one-to-one mapping preserves the equality/inequality relation between objects: $o_1 = o_2$ iff $int_{o_1} = int_{o_2}$.

Method Calls To handle methods defined by application classes, we instrument their code. The method call is modeled at the low level of assignment events that describe parameter/return passing. Consider the call statement $x = y.f(z)$ as an example. We model the parameter passing as **this** = y and $arg_0 = z$, and the return as $x = ret$, where **this**, arg_0 and ret are variables used inside the function $f()$.

The thread-start call **t.start()** introduces a complication, because the **this** variable in the invoked method **run()** is used by the child thread in spite of being defined by the parent thread. Recall that the **symbolizer** and **pointer analysis** scan the trace in a thread-local manner in search of a definition with a matching variable name. We fix this problem by associating the definition with the object **this** references and performing the search with the object.

We do not instrument the methods from JDK library classes. Instead, we treat them as the blackbox while handling the method invocations. We fix their parameters and return value to be the original values. Additionally, if the method is invoked on a shared object, then we also fix the happens-before order with respect to other calls on the same object. Fixing the order is important, as reordering may lead

⁵<https://sites.google.com/site/predictivedetector>

to different return values, which are beyond our reasoning power.

Another case of interest is the `Map get(k)` and `put(k,v)` methods, which are frequently invoked. We model the invocations of such methods as field access events: `x = map.k` and `map.k = v`, where `k` acts as the field. Modeling these calls as field access increases variability: The two types of invocations can be reordered, and the `get` invocation will return a different value after the reordering. Because we do not model other methods of the `Map` data type, we have to treat these method calls as a blackbox and fix their order (including the orders with respect to `get/put` calls).

Array Index We do not allow array indices to change during our analysis, and treat the index as if it were an object field (similarly to our treatment of `Map` keys).

Segmentation The trace is typically too long for the solver to be performant. To address this scalability issue, similarly to existing approaches [7], we split the trace into multiple segments (each of length 1,000), and apply the solver to each segment independently. The final local-store values, heap values, SSA naming scheme and point-to sets from each segment are passed to the next segment. Additionally, we may slightly adjust the segment size to avoid breaking lock regions into two segments.

6. EVALUATION

In our experiments, we measure the effectiveness and running time of our approach. For comparison purpose, we compare with `RVPredict` [7], the most recent causality-preserving predictive analysis of which the tool is available. We could not compare with the existing computation-based solutions [19, 27] of which the tools are not publicly available.

6.1 Benchmarks and Experimental Setup

We have evaluated both tools on a suite of 11 popular benchmarks, which contains large real-world applications, including `ftpserver`, `tomcat` and `lusearch`. Our suite covers concurrent programs of different size. The complete list of benchmarks appears in Table 1.

The effectiveness of predictive analysis critically hinges on the seed run. While the question of how to select the seed run is interesting and important, we do not make any contributions in this regard and leave it for future research. Instead, for fair comparison, we let both `IPA` and `RVPredict` operate on the same input trace, which is randomly generated with the input data corresponding to the most lightweight workload. For example, workloads consisting of two threads and relatively few loop iterations typically suffice to expose data races [11].

Our experiments were all conducted on an x86 64 Thinkpad W530 workstation with eight 2.30GHz Intel Core i7-3610QM cores, 16GB of RAM and 6M cache. The workstation runs Ubuntu 12.04, and has the Sun 64-Bit 1.6.0.26 JVM installed.

6.2 Performance Results

Table 1 provides information on the runtime behavior of our analysis. We describe the categories and columns comprising the table in turn.

The first category, **Candidates**, describes candidate race pairs to be checked by the constraint solver. Column **Raw**

presents the number of race pairs that access the same location in the seed run. Column **Ext** presents the number of pairs that do not access the same location in the seed run but may access the same location due to the change of the accessed locations (Section 4.1). Column **Hb** shows the number of the remaining pairs (in **Raw** and **Ext**) after the **HB** filter (Section 5) is applied to remove the false positives. Column **Lock** shows the number of the remaining pairs after the **Lockset** filter (Section 5), in addition to **HB** filter, is applied to remove the false positives. Each pair in the above is a pair of dynamic events. However, the race in the final report is represented as a pair of statements. Therefore, multiple dynamic event pairs may be mapped to one race pair in the final report.

From this category, we make two interesting observations.

First, according to Column **Ext**, our analysis significantly extends or changes the set of accessed locations in the large applications, but it hardly manages to extend the set in the small applications. This is because the small applications usually use the arrays to store the data and the reference of the array usually cannot be changed by the schedule mutation. Note that this does not imply that our approach has no effect on the small applications. Instead, similar to the existing symbolic analysis [27], our approach also allows changing the shared read of the primitive variable, thereby enabling more schedule mutations than the causality-preserving approaches [7, 26].

Second, the **HB** filter and **Lockset** filter perform the significant pruning in some cases. For example, in `ftpserver`, $476+28=504$ candidates are reduced to 131 by the **HB** filter. Close inspection shows that some shared variables (e.g., the `ready` flag variable) are written to by the main thread before the start of the children threads and the children threads only read the variables. The happens-before order imposed by the `start` primitive ensures that the writes and the reads cannot form actual races. This reduction lifts the burden on the ensuing confirmation.

The second category, **Events**, summarizes the events in the trace. They include initializations (**Init**), assignments (**Assign**), arithmetic operations (**Comp**), field accesses (**Acc**), branches (**Br**) and the synchronization operations (**Sync**). According to this breakdown, field accesses comprise most of the trace (around 40%), followed by assignments, computations and branches, where each of these other statement types accounts for roughly 20% of the trace. The fact that the majority of the trace are the field accesses implies that allowing the change of the heap locations may have a dramatic effect on the race detection.

The third category, **Constraints**, conveys the number of symbolic variables (**Var**) and the total number of `assert` constraints (**CON**). First, the number of symbolic variables is smaller than the number of assignments/computations, in spite of the fact that we use a single symbolic variable to denote each defined value. This is because we do not need to specify all assignments/computations in the constraint system. We simply need to specify the relevant ones, such as those in the trace segment preceding the candidate race pair.

The number of constraints is roughly proportional to the number of candidates to be confirmed. For each candidate, we need to specify independently (i.e., in a separate file) all the constraints related to the candidate.

Table 1. Runtime Data

Bench	Candidates				Events						Constraints		Time		
	Raw	Ext	Hb	Lock	Init	Assign	Comp	Acc	Br	Sync	Var	CON	Pre	Enc	Sol
bbuffer	57	0	51	24	227	1.3K	2.5K	3.1K	1.1K	570	0.8K	15K	0.01	1.5	34
mergesort	295	0	123	78	505	1.8K	1.5K	2.7K	1.6K	96	2.2K	97K	0.09	2.4	70
raytracer	164	0	164	127	7.8K	43K	74K	187K	107K	226	8.9K	664K	2.1	3.4	279
moldyn	1571	0	1543	1058	1968	21K	76K	163K	47K	84	32K	2546K	0.3	29.1	813
montecarlo	354	0	320	311	5.6K	18K	42K	98K	29K	130	12K	1302K	0.2	25.2	300
cache4j	294	0	258	150	316	5.9K	561	2K	722	396	8.9K	129K	0.07	3.6	216
hedc	261	34	201	125	1.9K	18.9K	7.4K	14K	14K	10	55K	2802K	0.1	35.4	410
weblech	9	0	9	6	2.4K	30K	1.3K	13K	8.3K	1.2K	8.8K	7.1K	0.01	0.03	5
ftpserver	476	28	131	87	1.3K	10K	256	12K	6.2K	201	21K	38K	1.7	4.2	770
tomcat	346	257	317	246	3.9K	32K	1.1K	12K	7.7K	632	67.3K	5412K	2.2	48.0	968
lusearch	11	9	15	9	2.7K	21K	7.8K	18K	5.8K	193	9.3K	9.9K	0.2	0.2	8

The fourth category, **Time**, specifies the time required for preprocessing (**Pre**), the time for constraint encoding (**Enc**), as well as the time spent on solving (**Sol**), where the time unit is second (s). Our measurements suggest that solving time dominates the overall analysis time. Note that while solving time is naturally correlated with the number of constraints, having more constraints does not immediately imply longer solving time. As counterexamples, we point to **moldyn** and **hedc**. the latter application has more constraints, but requires less solving time. This is because in general, solving time is also related to other factors, such as the complexity of each assert constraint.

While the scalability is a concern of IPA, we argue this is not a problem given that the solver research is progressing remarkably and the future solver would make the approach much more scalable. Even now, according to our experiments, the solving can finish within 16 min (a tolerable time for in house testing) for a large application.

6.3 Effectiveness

Table 2 lists the comparison between IPA and RVPredict, w.r.t. the solving time and the number of findings. Scrutiny of the results indicates significant differences. First, RVPredict finds 143 races in total, whereas IPA reports 259 races (i.e., close to 2X data races). We further validated, via manual inspection of all reported races, that the races output by RVPredict are a strict subset of ours.

This is compatible with our formal coverage guarantees. First, our analysis can find the **latent** races mentioned in Section 1, which requires the change of the locations accessed by the field accesses. Existing approaches cannot find such races due to the lack of the capability of changing the accessed locations. Second, by allowing more changes than existing approaches, our analysis enables many more trace transformations.

In terms of solving time, RVPredict mostly completes within 10%-60% of our running time. RVPredict does not need to specify constraints about program values. It only specifies the ordering constraints, which can be efficiently solved using the integer difference logic. Our approach, in contrast, needs to account for computations generating program values. However, note that our analysis' worst running time is 16 minutes (on **tomcat**, which is a large applications that generates the long traces). This is acceptable for testing, especially given that our analysis is able to detect significantly more real races than RVPredict.

In terms of the quality of the reported races, we have confirmed manually that all the data races output by IPA are indeed valid (i.e., none of the race is infeasible). Still, generally speaking, certain races are benign. In fact, some are introduced on purpose by the programmer to achieve better performance under certain circumstances (e.g., the customized spinlock implementation). We did not classify the races as benign or harmful, as this is an orthogonal concern to our goal of maximizing the coverage offered by predictive analysis. If needed, programmers can utilize available algorithms to classify reported races as either benign or harmful [14].

Table 2. Comparison

Bench	RVPredict		IPA	
	Solving	Races	Solving	Races
bbuffer	5	10	34	12
mergesort	71	15	70	15
raytracer	178	4	279	4
moldyn	757	0	813	2
montecarlo	142	9	300	12
cache4j	109	2	216	7
hedc	120	25	410	38
weblech	3	1	5	3
ftpserver	89	37	770	54
tomcat	122	38	968	107
lusearch	3	2	8	5

7. RELATED WORK

Initial attempts to address the challenge of race detection can be classified into two categories: (1) the lockset approaches and (2) happens-before approaches. The lockset approaches [13, 22] consider only lock mutual exclusion and are therefore unsound. The happens-before approaches [5, 4, 1] strictly follow the happens-before orders imposed in a concrete run, which are overly restrictive and lead to limited coverage. Hybrid approaches have been proposed [15, 9] to combine the two categories to improve coverage, but at the cost of sacrificing soundness.

Predictive analysis [26, 7, 19, 27] has recently been proposed to guarantee soundness while attempting to improve coverage. Variants of predictive analysis consider feasible reorderings of the seed trace. However, to achieve soundness, they often impose highly conservative constraints. For example, the shared reads should load the same objects into

the LHS reference variables. The requirement is proposed under the consideration that the value changes **may** lead to infeasibility in the downstream execution. Our analysis lifts the heavy barrier that has blocked predictive analysis from achieving high coverage.

Serbanuta et al. [24] and Huang et al. [7] both claim maximal detection capability, but under certain assumptions. Serbanuta et al. guarantee maximality under the assumption that branch events and local accesses are not included in the trace. Huang et al. guarantee maximality under the assumption that the local accesses are not included. The assumptions require their approaches analyze the trace conservatively. General speaking, the analysis that misses more information is more conservative. However, we think the assumptions are too restrictive in practice. The programmers can easily collect the local accesses and branches by instrumenting the binary code. Our approach, which relies on such collected information, enables much higher coverage.

CHESS [12] also focuses on exploring different schedules of a seed run. It schedules the execution at the synchronization points (e.g. lock/unlock) and exhaustively explores the schedules, which may cause scalability problems for real world applications. Heuristic-based search strategy may mitigate the scalability problems by guiding the exploration towards the defective location, but designing it requires great manual efforts. Comparatively, our approach is goal-directed. It starts with a goal, i.e., the candidate race, and then applies logic reasoning to infer the desired schedules. In addition, our approach allows the fine-grained scheduling at the level of individual shared accesses, which is often required to derive the program states necessary for exposing the bug.

Active testing [23] realizes the concurrent execution of a candidate race pair by blocking the threads at the racy accesses. It effectively finds many real races. However, there are still subtle races, for which simply blocking the threads cannot achieve the concurrent execution of the candidate race pair. For such races, the predictive analysis offers the fine-grained control of the schedule required to achieve the concurrent execution.

Predictive analysis follows the same paths taken in the seed run while mutating the schedules, which complements the model-checking approaches [12] that can explore different paths. Predictive analysis also complements the concurrent test generation [17]. Predictive analysis can be applied to the library code with the help of test generation, and the test generation can generate higher quality test cases by incorporating the predictive analysis.

8. CONCLUSION

We have developed IPA, a predictive analysis that mutates the schedule order of a trace to produce other feasible traces that expose bugs. IPA allows the schedule mutation to change the location referenced by a shared access. IPA applies a hybrid symbolic encoding scheme to achieve practical applicability. Our experiments show that IPA is able to detect 2X as many data races as existing predictive analysis, and do so without any false positives.

9. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments. This research was supported, in part, by DARPA under contract FA8650-15-C-7562, NSF under awards 1409668,

1320444, and 1320306, ONR under contract N000141410468, and Cisco Systems under an unrestricted gift. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

10. REFERENCES

- [1] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: Proportional detection of data races. In *PLDI*, 2010.
- [2] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *CACM*, 56(2), Feb. 2013.
- [3] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Security and Privacy*, pages 380–394, 2012.
- [4] M. Christiaens and K. De Bosschere. Trade, a topological approach to on-the-fly race detection in java programs. In *Symposium on JVM*, '01, 2001.
- [5] C. Flanagan and S. N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *PLDI*, 2009.
- [6] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *TOPLAS*, 1990.
- [7] J. Huang, P. O. Meredith, and G. Rosu. Maximal sound predictive race detection with control flow abstraction. In *PLDI*, 2014.
- [8] P. Joshi, M. Naik, K. Sen, and D. Gay. An effective dynamic analysis for detecting generalized deadlocks. In *FSE*, 2010.
- [9] V. Kahlon and C. Wang. Universal causality graphs: A precise happens-before model for detecting bugs in concurrent programs. In *CAV*, 2010.
- [10] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 1979.
- [11] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, pages 329–339, New York, NY, USA, 2008. ACM.
- [12] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.
- [13] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *PLDI*, 2006.
- [14] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *PLDI*, 2007.
- [15] R. O’Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *PPoPP*, 2003.
- [16] E. Pozniarsky and A. Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. In *PPoPP*, 2003.
- [17] M. Pradel and T. R. Gross. Fully automatic and precise detection of thread safety violations. In *PLDI*, 2012.
- [18] R. Rugina and M. C. Rinard. Pointer analysis for structured parallel programs. *TOPLAS*, 25(1), Jan. 2003.
- [19] M. Said, C. Wang, Z. Yang, and K. Sakallah. Generating data race witnesses by an smt-based analysis. In *FM*, 2011.

- [20] M. Samak and M. K. Ramanathan. Multithreaded test synthesis for deadlock detection. In *OOPSLA*, 2014.
- [21] M. Samak, M. K. Ramanathan, and S. Jagannathan. Synthesizing racy tests. In *PLDI*, 2015.
- [22] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *SOSP*, 1997.
- [23] K. Sen. Race directed random testing of concurrent programs. In *PLDI*, 2008.
- [24] T. F. Serbanuta, F. Chen, and G. Rosu. Maximal causal models for sequentially consistent systems. In *RV*, 2012.
- [25] P. K. Sinha. *Distributed operating systems: concepts and design*. IEEE Press, 1997.
- [26] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound predictive race detection in polynomial time. In *POPL*, 2012.
- [27] C. Wang, S. Kundu, M. Ganai, and A. Gupta. Symbolic predictive analysis for concurrent programs.