RECIPE: Relaxed Sound Predictive Analysis

Peng Liu
Purdue University
peng74@cs.purdue.edu

Xiangyu Zhang
Purdue University
xyzhang@cs.purdue.edu

Omer Tripp
IBM Research, NY
otripp@us.ibm.com

Yunhui Zheng
Purdue University
zheng16@cs.purdue.edu

***Abstract*—Predictive analysis has recently emerged as a promising technique for sound detection of multithreading bugs (i.e., without false alarms). The idea is to start from an instrumented execution trace, and transform it based on the concrete information it provides, such as concrete values and memory accesses, to expose bugs, while preserving feasibility. For example, it may reoder trace events to simulate different thread schedules, as long as value flow dependencies are preserved.**

Predictive analysis has been successfully applied to data-race detection, although the state-of-the-art techniques are overly restrictive in the transformations they allow such that their analysis coverage is unnecessarily limited.

In this paper, we present RECIPE, a predictive race detector that has two important relaxations: value and path relaxations without affecting the soundness guarantee. The former allows variables to have different values and the latter allows exploring unexecuted paths that are neighbors to the traced path. Our results show that RECIPE is able to detect $2.5X$ as many data races on a representative set of large-scale benchmarks as compared to the state-of-the-art.

I. INTRODUCTION

Program analyses can be largely classified to static and dynamic analyses. Static analyses abstract programs statically. They are usually complete and do not require concrete program inputs to drive the analyses. However, they have difficulty in scaling to large and complex programs when the analyses are path-sensitive, due to the sheer number of paths and the complexity of the paths. Path-insensitive analyses often do not have sufficient precision on the other hand. Static analyses also have difficulty in modeling complex aliasing behavior. In contrast, dynamic analyses focus on analyzing a few executions, usually just one. They are sound and easily scale to long and complex executions. But they can only reason about properties that manifest themselves in the execution(s).

Recently, a novel kind of analysis called predictive analysis [18], [7], [14] was proposed. It has the capabilities of achieving soundness as dynamic analyses and reasoning about properties that did not occur during the execution just like in static analysis. The basic idea is to leverage dynamic analysis to provide the information that is difficult to acquire through static analysis, such as a complex execution path that covers important functionalities and aliasing information. Note that during execution, we know precisely which write access reaches any given read access. While such information is encoded in a trace, predictive analysis reasons about mutation of the trace to expose problems or study properties that may not manifest during execution. It was used in sound data race detection. In particular, it collects a trace of threaded execution that contains rich runtime information such as concrete values, execution path, and memory accesses. It then leverages constraint solving to reason about if races can occur with a different thread schedule, *while enforcing the critical runtime information (e.g. values, thread local paths, and memory accesses) remains the same during schedule perturbation*. The rationale is that it becomes as difficult as static analysis if we allow these critical runtime information change. Predictive analysis has been shown to be very effective. It can detect real data races from complex programs. And more importantly, it guarantees the results are sound (i.e. no false positives).

```
        x = 0; y = 0;
      T1               T2
 1: y = 3;
 2: x = 1;
 3: y = 5;
                   4: if (y > 2)
                   5:   z=1+x;
                   6: else
                   7:   w=2+x;
```

Fig. 1. Example illustrating ordering constraints beyond synchronization primitives

Despite its effectiveness, we observe that existing predictive analyses are too restrictive. They require too much runtime information to remain unchanged during analysis such that coverage is unnecessarily limited.

Consider the example in Figure 1, which shows an execution trace of two threads. The statements in red denote an unexplored branch outside the trace. Existing predictive analyses are unable to detect the race between lines 2 and 5 because they require that during trace permutations (by different schedules), a variable must not have a different value in a permuted trace[1]. As such, they cannot execute line 4 before line 3, which is needed to expose the race between lines 2 and 5. A second race between lines 2 and 7 is also missed as the mutated trace must have the same thread local path as the original trace.

However, we observe that such restrictions are unnecessary. For example, we can allow line 4 to receive its value from line 1 as it has the same effect on thread local execution (i.e.

[1] They allow the variable involved in a race to have different values.

the same branch is taken). This relaxation would allow us to execute lines 4 and 5 before line 2, thereby exposing the first missing race. We can also allow the trace to take the else branch such that the second race is also exposed. Note that even though that branch was not executed originally, its effect can be precisely modeled as the address of x is known from the original trace. As such, we can still perform sound analysis.

Therefore in this paper, we propose RECIPE, a relaxed predictive analysis. We identify the critical runtime informantion that needs to be preserved during trace perturbation and relax the remaining information. In particular, we preserve all the addresses de-referenced (e.g. heap and array accesses) in the original execution, and some of the branch outcomes such that part of the thread local paths stay intact. The criterion is that we forbid RECIPE to explore an unexecuted branch if the effect of the branch cannot be precisely modeled (e.g. an array element is read while it was not defined in the original trace). The essence of our technique is to *explore the neighborhood of the original execution as much as possible*. Due to the substantially enlarged search space, our results show that a race detector based on our relaxed predictive analysis can detect $2.5X$ as many races as the existing predictive analysis, while guaranteeing soundness.

Our contributions are summarized as follows.

- We propose to relax existing predictive analysis and identify a bound of relaxation (i.e. what can be relaxed).
- We develop a constraint encoding scheme that encodes the necessary information (i.e. the part that must be preserved) and the relaxed information (i.e., the part that may be mutated) uniformly as part of a single logical representation.
- We develop a propotype RECIPE. Our results on 16 benchmarks show RECIPEis significantly more effective than existing predictive analyses. We also conducted case studies to confirm the effectiveness.

II. TECHNICAL OVERVIEW

In this section, we walk the reader through a detailed technical description of our approach based on the example in Figure 1. As input, we assume (i) a program P as well as (ii) a trace of P recorded during a dynamic execution.

A. Preliminaries

We first introduce some notation used throughout this paper. Intuitively, a trace is a sequence of events recorded during the observed run. An event, $e = < t, id, inst >$, is a concrete representation of the runtime execution of a static instruction $inst$, such that (i) t refers to the thread; (ii) id is a *unique* identifier of the event (unless specified otherwise, we use the index of an event in the trace as its id); and (iii) $inst$ is a static three-address instruction (involving at most three operands).

For this paper, we utilize the instructions listed in Table I. When the variable does not appear on the left hand side of an equation, such as y in $x.f = y$, it may refer to a variable, a constant or an object creation expression $new(...)$.

s ::=		
`y = x.f`	|	**(heapr)**
`x.f = y`	|	**(heapw)**
`z = x` *bop* `y`	|	**(assign)**
`if (x` *bop* `y) goto ...`	|	**(branch)**
`lock(l)` | `unlock(l)`	|	**(sync)**
`fork(t)` | `join(t)`| `begin(t)` | `end(t)`		**(thread)**

TABLE I
INSTRUCTION TYPES

Symbol *bop* stands for a binary operator, which may refer to $+, -, *, /, \%, \wedge, \vee$ in an assignment, or $<, >, =, \wedge, \vee$ in a branch instruction. The target of the branch instruction is not important in our scope, therefore, we may abbreviate a branch instruction as its boolean expression afterwards. The listed instructions suffice to represent all trace events of interest. Note that a concrete finite execution trace can be reduced to a straight-line loop-free call-free path program (argument passing of a method call is modeled as assignments and virtual call resolutions are modeled as branches).

Trace Representation A trace τ is represented as $\tau = < \Gamma, \{\tau_{t_1}, \tau_{t_2}, \dots \tau_{t_n}\}, O, \theta >$, where

- Γ refers to the set of threads involved in the trace, which include t_1, t_2, …, t_n;
- $\{\tau_{t_1}, \tau_{t_2}, \dots \tau_{t_n}\}$ denotes the event sequences produced by each thread;
- O is a function that assigns to every event an integer value that denotes its scheduling order: e_i is scheduled earlier than e_j iff $O(e_i) < O(e_j)$; and
- θ is a function that maps every live variable to a value.

To ensure the integrity of θ as well simplify downstream encoding steps, we rewrite the raw input trace into SSA form, such that each variable is defined exactly once.

B. Constraint System

In this section, we are going to informally explain the two relaxations we make which allow us to catch the two races missed by existing predictive analyses. Our analysis takes the program and a trace as the input. Suppose it starts with the following trace of the program in Figure 1: $e_1 e_2 e_3 e_4 e_5$, where the line number is used as the event id (subscript).

I. Value Relaxation Our first relaxation is to allow variables to have different values as long as the same thread local path is preserved. It does not require data dependences or same variable values as in [18], [7], [14]. This allows us to catch the racy pair (e_2, e_5). Given a trace, $\tau = < \Gamma, \{\tau_{t_1}, \tau_{t_2}, \dots \tau_{t_n}\}, O, \theta >$, our analysis computes a new trace, $\tau' = < \Gamma, \{\tau_{t_1}, \tau_{t_2}, \dots \tau_{t_n}\}, O', \theta' >$. τ' preserves the same event sequence for each thread, i.e., T_1 still executes $e_1 e_2 e_3$ and T_2 still executes $e_4 e_5$, but it reschedules the events from different threads so that the potential race pair runs concurrently, i.e., e_2 and e_5 run concurrently. When the schedule changes, the variables may get different values. Therefore, we need to compute both the new schedule O' and the new value mapping θ', in order to witness a race. To guarantee that τ' is a feasible trace of the program P, the computation needs to respect a set of constraints, which include the value constraints, control flow constraints and synchronization constraints [2]. We explain the constraints briefly in the following. Details are in Section III and the soundness guarantee is explained in Section V.

We first preprocess the trace as follows. We introduce symbols to replace the shared accesses in the instructions, e.g., We use W_x^{id}/R_x^{id} to replace the write/read of the shared variable x by the event e_{id}.

Race Condition A potential race pair involves two accesses of the same shared location from different threads, where at least one is write. Given any potential race pair, e.g., (e_2, e_5), it is a real race if and only if the two accesses can occur at the same time (race condition), which is captured by the race condition constraint: $O'(e_2) = O'(e_5)$. The race condition constraint— together with the other constraints — guarantees the feasibility of the predicted race if a solution is found for the overall constraint system.

Intra-thread Constraints Our analysis needs to preserve the same event sequence for each thread, which requires

- **Control Flow Constraints** Each thread takes the same control flow (or branch decisions) to reproduce the events. For the running example (ignoring the statements in red), this yields: $\theta'(R_{\mathtt{y}}^4) > 2 \equiv \mathbf{true}$. That is, in the predicted trace τ', the value of variable $\mathtt{y}$ read at e_4 should be greater than 2.
- **Intra-thread Order Constraints** The events should follow the same thread-local order as in the original trace. This constraint is imposed by the fact that the two runs share the same instruction sequence and take the same control flow. We obtain the formula: $O'(e_1) < O'(e_2) < O'(e_3) \wedge O'(e_4) < O'(e_5)$.
- **Intra-thread Value Constraints** The events should respect the constraints imposed by each instruction. For our running example, we obtain:

$$\begin{array}{l}\theta'(W_{\mathtt{x}}^0) = 0 \wedge \theta'(W_{\mathtt{y}}^0) = 0 \wedge \theta'(W_{\mathtt{y}}^1) = 3 \\ \wedge \theta'(W_{\mathtt{x}}^2) = 1 \wedge \theta'(W_{\mathtt{y}}^3) = 5\end{array}$$

  As an example, $\theta'(W_{\mathtt{y}}^3) = 5$ denotes that the predicted run still assigns the value 5 to the variable $\mathtt{y}$ at event e_3.

Inter-thread Value Constraints These constraints capture what writes the read events read from and under what scheduling condition the value flow occurs. The resulting formula for our example is

$$\begin{array}{ll} & (\theta'(R_{\mathtt{y}}^4) = \theta'(W_{\mathtt{y}}^0) \wedge O'(e_4) < O'(e_1)) \\ \bigvee & (\theta'(R_{\mathtt{y}}^4) = \theta'(W_{\mathtt{y}}^1) \wedge O'(e_1) < O'(e_4) < O'(e_3)) \\ \bigvee & (\theta'(R_{\mathtt{y}}^4) = \theta'(W_{\mathtt{y}}^3) \wedge O'(e_3) < O'(e_4))\end{array}$$

Notice, importantly, that the formula associates the value constraints with the scheduling order constraints. As an example, $\theta'(R_{\mathtt{y}}^4) = \theta'(W_{\mathtt{y}}^1) \wedge O'(e_1) < O'(e_4) < O'(e_3)$ means that, in the predicted run, the read e_4 reads from the write e_1, under the condition that e_4 happens after e_1 and no other writes (such as e_3) interleave them.

The formulas from the different encoding steps are conjoined and sent to an off-the-shelf solver such as z3. The race under analysis is real if the solver returns a solution, which includes the scheduling order and the values for the variables.

II. Path Relaxation Besides the first relaxation, which focuses on the given trace, the second relaxation is to even reason about unexecuted branches as long as they can be precisely modeled. In our running example, this is essential to discovering the race between lines 2 and 7. We explore the unexecuted branch statically and collect constraints during the exploration. We then form a disjunction of the original constraints and the collected constraints to include more possible paths. The solver may pick up the unexplored branch to detect a race. Back to our example, event e_7 is in the unexplored branch, RECIPE is able to identify the race between e_7 and e_2 after encoding the unexplored branch.

For the racy events, e_2 and e_7, we first specify the race condition constraint $\theta'(R_y^4) \leq 2 \rightarrow O'_{e_2} = O'(e_7)$. Note that the constraint associates the race with the branch condition that enables each event. Also we need to encode (1) the intra-thread constraints, such as the order constraints $O'(e_4) < O'(e_5) \wedge O'(e_4) < O'(e_7)$, and (2) the inter-thread constraints, which remain the same as before. With these constraints, the solver confirms the race pair, (e_2, e_7). More details are in Section IV.

Highlights RECIPE finds two more pairs of races beyond existing analyses [18], [7]: (e_2, e_5) and (e_2, e_7). For the first race, we do not require e_4 to read from e_3 to retain the original value as in the existing approaches. Rather, we allow e_4 to read from e_1 while still preserving the thread local path. This relaxation allows the events after e_4 to run concurrently with the events before e_3, leading to the detection of the first race. For the second race, we cover unexecuted branches and query the solver to compute a feasible schedule that exercises them.

III. VALUE RELAXATION

Our first relaxation is to allow variables to have different values as long as thread local paths are preserved. Given a trace, $\tau =< \Gamma, \{\tau_{t_1}, \tau_{t_2}, \ldots \tau_{t_n}\}, O, \theta >$, the main goal of our analysis is to derive a new trace, $\tau' =< \Gamma, \{\tau_{t_1}, \tau_{t_2}, \ldots \tau_{t_n}\}, O', \theta' >$, which reschedules the events from different threads (while preserving intra-thread event order) to prove the race. The key insight underlying this relaxation is that a thread is allowed to read a different value, potentially from a different write, as long as branching decisions remain unchanged. Note that we also need to compute the new value mapping θ', because the rescheduling may alter the value every variable is assigned to. The computation of the new schedule and new value mapping is explained in the following.

A. SSA Form of the Trace

First of all, we rewrite the trace into SSA form, such that every variable is defined exactly once in the trace. This requirement is a prerequisite for the computation of the mapping, in which each variable is mapped to exactly one value.

[2] We omit the synchronization constraints in this paper, which are already well explained in existing techniques [18], [7].

RECIPE handles local assignments and heap accesses differently.

- **Local Assignment** The SSA form for local assignments resembles the static SSA form in compiler optimizations, except that loops and recursions are fully resolved in the trace, obviating the need for Phi nodes[3] . More concretely, we replace the variable v defined in an event, as well as the following uses of the definition, to a new variable v^{id}, where id is the unique id of the event. The uniqueness of the id guarantees that no two events define the same variable, i.e., each variable is defined exactly once. In Figure 2, for example, the local variable x defined at line 4 is renamed as x^4.
- **Heap accesses** The def/use relation between local heap accesses can be determined locally. Therefore, we treat them in the same way as local assignments. The def/use relations between shared heap accesses are more complex. We may link a read with different writes under different schedules, e.g., the read of y at line 4 in Figure 1 can be from before line 1, line 1, or line 3. Therefore, we introduce two symbols to represent the shared read and shared write respectively and use them to reason about the possible options. The details are as follows.
  - **Local Heap Accesses** Given the write $x.f = y$ to a local heap location at event e_{id}, we introduce a fresh local variable $l^{id}_{o.f}$ (o is the runtime object referenced by x) to replace $x.f$. We also replace the uses of the location $o.f$ accordingly. Here the subscript $o.f$ is interpreted as a heap location, i.e., the field f of the object o. Consider the example in Figure 2. At lines 6-7, we replace the local heap accesses as the local variable $l^6_{o3.f}$.
  - **Shared Heap Accesses** Given the write $x.f = y$ or the read $y = x.f$ of a shared location at event e_{id}, we introduce two symbols, $W^{id}_{\texttt{o.f}}$ and $R^{id}_{\texttt{o.f}}$ respectively, where o is the runtime object referenced by x. Similarly, the subscript $o.f$ is interpreted as a heap location, i.e., the field f of the object o. Note that in local heap encoding, both read and write are bounded to the same variable, whereas in shared heap encoding, we use different variables to denote read and write, allowing the solver to reason about the different dataflow.

  We use the heap location such as $o.f$ in the symbolic form of heap accesses. An underlying assumption is that the heap location remains unchanged, which we ensure through additional constraints (Section III).

B. Constraint System

Based on the SSA form of trace τ, we build the constraints to compute a new trace τ' with a new schedule O' and a new value mapping θ'. Consider the example in Figure 2. In the original trace τ, the event e_8 reads the value of 2 from e_3. Existing approaches enforce the same value dependence between e_3 and e_8. Therefore, e_{11} must happen after e_2 and the race between them is missed. Our analysis allows event e_8 to read from e_1, producing the new value mapping $\theta'(y^8) = 3$. Accordingly, at e_9 the value mapping for x^9 is also updated as 13 ($\theta'(y^8)$=3, $\theta'(z^7) = 10$), enabling the same true branch at line 10. Finally, the events e_2 and e_{11} are scheduled to run concurrently. The relaxation needs to respect a set of constraints, which we explain in the following. We omit the synchronization constraints as they are well explained in [18], [7].

```
                    s1 = new();//o1
                    s2 = new();//o2
        T_1                 T_2                 Trace
1: s1.f=3;                               | W^1_{o1.f}=3;
2: s2.f=1;                               | W^2_{o2.f}=1;
3: s1.f=2;                               | W^3_{o1.f}=2;
                    4: x=0;              | x^4=0;
                    5: w=new();//o3      | w^5=o3;
                    6: w.f=10;           | l^6_{o3.f}=10;
                    7: z=w.f;            | z^7=l^6_{o3.f};
                    8: y=s1.f;           | y^8=R^8_{o1.f};
                    9: x=y+z;            | x^9=y^8+z^7;
                    10: if(x>11)         | if(x^9>11)
                    11:     s2.f=x;      | W^11_{o2.f}=x^9;
```

Fig. 2. Running Example (shared variables are in bold font, the line number is used as the event id).

Race Condition An even pair (e_i, e_j) forms a race iff (1) e_i and e_j access a shared location ℓ, (2) at least one of them writes ℓ, and (3) e_i and e_j run concurrently (being associated with different threads). We refer to the candidate pair throughout this section as (candidate) *racy events*.

We first identify all candidate pairs. We then check each pair separately. The checking is achieved by encoding all necessary constraints. The first constraint asserts the concurrency: $O'(e_i) = O'(e_j)$.

Intra-thread Constraints Given racy events e_i and e_j, suppose e_i is after e_j in the original trace τ. Our analysis reschedules the events in the prefix up to e_i, so that e_i and e_j can run concurrently. We refer to the prefix as $prefix(\tau, e_i)$. Throughout this section, we only consider events in the prefix.

Preservation of the event sequence (up to e) for each thread requires the following intra-thread constraints:

- **Control Flow Constraints** The branches in the predicted run should take the same decisions as in the original trace so that each thread reproduces the same sequence of events[4].

  Without loss of generality, we assume the branch event e_k is in the form of $if(x < y)$. Then we require that

$$\bigwedge_{e_k \in prefix(\tau, e_i) \wedge inst^{e_k} = if(x<y).} \theta'(x) < \theta'(y) \equiv \theta(x) < \theta(y)$$

  The constraint specifies that the branch condition should be evaluated as the same boolean value in the predicted

[3] While RECIPE in fact uses Phi nodes in the second relaxation, for simplicity during discussion, we do not assume Phi nodes in this stage.

[4] We allow path differences in the second relaxation. This is to avoid presenting the encoding of both relaxations together, which is overly complex.

run τ' as in the original trace τ. Importantly, Unlike existing analyses [18], [7], we do not pose the requirement that the values flowing into branching statements remain the same, but adopt the relaxed requirement that the branch outcome remains the same. Consider the example in Figure 2, given the branch if$(x^9{>}11)at e_{10}$, the original mapping is $\theta(x^9) = 12$. We allow a different value mapping, e.g., $\theta'(x^9) = 13$, which retains the same truth value for the branch.

- **Intra-thread Order Constraints** The events in the sequence should follow the same order as in the original trace. More formally, we require that

  $$\begin{array}{ll} \forall e_m, e_n \in prefix(\tau, e_i), s.t., & t^{e_m} = t^{e_n}. \\ O(e_m) < O(e_n) & \Rightarrow O'(e_m) < O'(e_n) \end{array}$$

  This constraint specifies that two events from the same thread (t^{e_m} denotes the thread of e_m) should follow the same order as in the original trace.
- **Intra-thread Value Constraints** The value mapping of the variables should not contradict the constraints imposed by individual instruction. Without loss of generality, suppose the instruction is a local assignment in the form of $x = y + z$, then we require that

  $$\bigwedge_{e_k \in prefix(\tau, e_i) \wedge inst^{e_k} = x=y+z} \theta'(x) = \theta'(y) + \theta'(z)$$

  Recall that every variable may obtain a different value in the new trace τ' because it may read from some shared variable affected by the rescheduling. The constraint specifies that, the variables should be re-assigned to some values that are consistent with the instruction. Consider the example in Figure 2, given the event $x^9 = y^8 + z^7$ at line 9, if the mappings $\theta'(y^8)$ and $\theta'(z^7)$ change to 3 and 10 respectively, the mapping $\theta'(x^9)$ should change to 13 to be consistent.

Inter-thread Value Constraints for Relaxation

To ensure feasibility under value relaxation we need to ensure the flow between read and write with some execution schedule. For example, in Figure 2, for the read at e_8 to read from the write e_1, we need a schedule where e_8 happens after e_1 and other writes such as e_3 do not interleave them.

In general, the constraint formula, given read R^m_ℓ of location ℓ at event e_m with set $\mathcal{W}$ the matching write events (i.e., events including write access to ℓ), takes the following form:

$$\begin{array}{ll} \bigvee_{e_n \in \mathcal{W}} & (\theta'(R^m_\ell) = \theta'(W^n_\ell)) \\ \quad\bigwedge & O'(e_n) < O'(e_m) \\ \quad\bigwedge_{e_p \in \mathcal{W} \setminus \{e_n\}} & (O'(e_p) < O'(e_n) \vee O'(e_m) < O'(e_p)) \end{array}$$

This disjunctive formula iterates over all matching write events, and requires for each write that (i) it occurs prior to the read event $O'(e_n) < O'(e_m)$ and (ii) all other write events either occur before $O'(e_p) < O'(e_n)$ it or after the read event $O'(e_m) < O'(e_p)$.

An important concern that arises due to value relaxation is that heap accesses may change their meaning, i.e., they involve different base objects and no longer match with each other. As an illustration, we refer to Figure 3. While the read at event e_7 appears to match the write at e_5, this is conditioned on the read at e_6 being linked to the assignment at e_4 (object o_2 is the object denoted by z according to the source code). Suppose event e_6 reads from e_3 due to the relaxation. Then e_7 and e_5 should not match with each other as they do not share the same base object.

T_1	T_2
1: x^1 = new();// creates o1	
2: x^2= new();// creates o2	
3: W^3_y = x^1;	
4: W^4_y = x^2;	
5: $W^5_{o2.f}$ = 5;	
	6: z^6 = R^6_y;
	7: w^7 = $R^7_{o2.f}$;

Fig. 3. A trace. The source code of line 7 is "w=z.f".

Invariant Base Object Constraints To address this challenge, we enhance the constraint system with the requirement that heap objects that are dereferenced in field accesses before the racy events retain their original address in the predicted trace. This provides two guarantees: First, matching heap access events in the original trace are guaranteed to also match in the predicted trace. Second, candidate races in the original trace remain viable in the predicted trace as they still refer to same location.

Formally, we require that

$$\bigwedge_{e_k\ has\ x.f \wedge e_i \in prefix(\tau, e_i)} \theta'(x) \equiv \theta(x)$$

This constraint fixes that all heap dereferences prior to the racy event retain their original base object as in the original trace τ. The array accesses are handled similarly to the field accesses, but we need to additionally require the index variable to be the same as in the original trace. In addition, for local heap accesses, in case that the base variable reads a local object from a shared reference variable, the base variable may change due to the rescheduling. We also fix the base variable as a constant, in order to ensure the validity of the SSA encoding of the local heap accesses and preserve the def/use chains among them. The constraint is specified during the SSA encoding.

By sending the above constraints to a solver, we compute the necessary schedule orders among the events as well as the mapping of the variables. The necessary schedule orders define a partial order among the events, which permit a set of schedules that define the complete order that complies with it.

IV. Path Relaxation

We now switch to the second feature of Recipe, which is to reason about neighboring executions involving unexecuted branches. At the high level, we enhance the constraints to model the unexecuted branches. However to ensure soundness, we cannot encode the unexecuted branches as normal static analyses do because they usually perform approximations such as bounded loop unrolling and heap abstraction. The basic idea is to *only encode the unexecuted branches when they can be precisely modeled based on our trace τ.*

In particular, we have the following criteria in determining if an unexecuted branch should be included in the constraint system. These are essentially the bound of our path relaxation. (1) If there is a loop inside the unexecuted branch, the loop bound needs to be resolved to a constant from the trace (otherwise we do not know how many times we need to unroll). (2) The unexecuted branch is ignored if it contains recursive function(s). (3) Every base reference variable in the unexecuted branch needs to be fully resolved to a concrete object, as required by our earlier analysis (Section III). This can be done by backward traversal of the trace: if the base reference depends on some shared read in the unexplored branch, of which the value is uncertain, it cannot be resolved to an object; if the base reference depends on multiple definitions from different paths, it cannot be resolved to an object. Otherwise, it can be resolved. (4) The indices of array read accesses in the unexecuted branch need to be resolved to constant values from the trace. This is to prevent reading an array element that was not defined. (5) The unexecuted branch is ignored if it uses collections. (6) For method calls in the unexecuted branch, we require the caller object can be fully resolved from the trace.

Next, we will explain how to encode an unexecuted branch in details, using the example in Figure 1 for demonstration. Given a branch event e_b, we assume one of its branches is unexecuted and needs to be (statically) extended in the following discussion.

Events We traverse the control flow graph of the unexplored branch and generate an event, $< t, id, inst >$, to represent each instruction $inst$. The id can be any unique value. We simply use the line number as the event id in our example. As loops are unrolled, we establish the one-to-one mapping between each instruction $inst$ and the event e_{inst}.

SSA Form Similar to in Section III, we need the SSA form of the instruction in each event. To achieve this, we apply SSA transformation to the code of unexecuted branch and combine its SSA form with the existing SSA form of the corresponding executed branch to produce the final SSA form. The SSA transformation is standard, but special considerations are needed.

- **Local primitive variables** For local reads inside the extended branch whose definitions are before the branch event e_b, we reuse their SSA forms which are already in the trace τ. In addition, we encode conditional statements inside the extended branch using Phi node, e.g., $y = Phi(y_1, C_1, y_2, C_2, \dots y_i, C_i)$, meaning that y is equal to y_i in the condition C_i. Phi nodes are also used to integrate the definitions in the extended branch with the executed branch.
- **Local Heap Accesses** Given a heap access $x.f$, recall that the base variable x is fully resolved to a concrete object o, we replace it as a local variable $l_{o.f}$ before the SSA transformation and treat it as a local primitive variable during the SSA transformation.
- **Shared Heap Accesses** The standard SSA transformation does not encode the shared heap accesses. Similar to Section III, we introduce special symbols to denote shared read/write.

```
4: if($R_y^4$>2)
5:     $z^5$=1+$R_x^5$;
6: else
7:     $w^7$=$2+R_x^7$;
```

Fig. 4. SSA form of both true and false branches.

The SSA form for extending line 4 in Figure 1 is shown in Figure 4. We omit the Phi nodes for z and w at the end as they are never used later.

Constraints During traversal, we also record the shared access events. For the set of accesses S_1 of the location ℓ, we also find its accesses S_2 from other threads. We assert the feasibility of each race pair $\in S_1 \times S_2$ by specifying constraints and checking them.

Race Condition Constraint The first type of constraint is the race condition constraint, e.g., $O'(e_2) = O'(e_7)$ specifies that the event at line 2 may form a race with the event at line 7, which is actually not executed in the original trace. Importantly, we need to specify the race condition under the specific branch condition, e.g., $\theta'(R_y^4) \leq 2 \rightarrow O'(e_2) = O'(e_7)$. The underlying reason is, the race is valid, or e_7 can be executed, only under the branch condition $\theta'(R_y^4) \leq 2$.

Intra-thread Control Flow Constraint Different from the strategy in Section III, we do not assert the branch the predicted run will take. Instead, we assert that either branch is possible. The solver automatically reasons about which branch should be taken and encodes it in the solution.

Intra-thread Value Constraint Intra-thread value constraint is the core constraint that realizes the exploration of different branches. It asserts the value constraints imposed by the assign/heapr/heapw events take effect only when the guarding condition is satisfied. Suppose $IF(e_b)$ denotes the event sequence that corresponds to the true branch, $ELSE(e_b)$ denotes the event sequence that corresponds to the false branch, the branch event e_b is $if(x_b < y_b)$, the event e_m in $IF(e_b)$ is in the form of $x_m = y_m + z_m$ and the event e_n in $ELSE(e_b)$ is in the form of $x_n = y_n + z_n$. Then, we have,

$$\bigwedge_{e_m \in IF(e_b)} (\theta'(x_b) < \theta'(y_b) \rightarrow \theta'(x_m) = \theta'(y_m) + \theta'(z_m))$$
$$\bigwedge$$
$$\bigwedge_{e_n \in ELSE(e_b)} (\neg(\theta'(x_b) < \theta'(y_b)) \rightarrow \theta'(x_n) = \theta'(y_n) + \theta'(z_n))$$

Consider our example, the corresponding value constraint is,

$$\theta'(R_y^4) > 2 \rightarrow \theta'(z^5) = 1 + \theta'(R_x^5)$$
$$\wedge$$
$$\theta'(R_y^4) \leq 2 \rightarrow \theta'(w^7) = 2 + \theta'(R_x^7)$$

This is also how we encode Phi operations.

Inter-thread value constraints The inter-thread value constraints are similarly extended. Details are elided.

Intra-thread Order Constraints The intra-thread order constraint captures the program order imposed by the branch structure and the instruction sequence. Suppose $IF(e_b)$=$e_{i_1}e_{i_2}\ldots e_{i_m}$, $ELSE(e_b) = e_{j_1}e_{j_2}\ldots e_{j_n}$, the first event after the branch structure is e_{end}. Then we have,

$$
\begin{aligned}
&O'(e_b) < O'(e_{i_1}) < O'(e_{i_2}) < \cdots < O'(e_{i_m}) < O'(e_{end}) \\
&\wedge \\
&O'(e_b) < O'(e_{j_1}) < O'(e_{j_2}) < \cdots < O'(e_{j_n}) < O'(e_{end})
\end{aligned}
$$

The order constraints capture the program order, which are unconditionally valid. That is also why we do not specify the order constraints under specific branch conditions.

V. Theoretical Guarantees

In this section, we first prove the soundness of Recipe and then discuss the detection capability as compared to existing approaches. Both the proof and discussion are in the context of sequential consistency model [10].

Theorem 1 (Soundness). *The trace τ' computed by the solver is a feasible trace for the concurrent program.*

Proof. The proof consists of two parts: (1) the trace is feasible, and (2) the trace can be generated by the program. We sketch the proof as follows.

- τ' is feasible. Researchers [21], [17] point out a trace is feasible iff it satisfies the sequential consistency, i.e., $\forall object\ o, \tau' \downarrow_o$ satisfies the serial specification of the object o. If all the events considered are of the types in Table I, then the sequential consistency precisely means the following: (1) read-write consistency, i.e., each read event of a variable should contain the value written by the most recent write event, and (2) the synchronization consistency, e.g., the acquisition and release of a lock should not interleave with other lock operations of the same lock, the start event of a thread should follow the fork event of the parent thread. Our inter-thread data flow constraints guarantee the read-write consistency among shared locations. The SSA form encodes the read-write consistency among local variables. The synchronization consistency is also captured as constraints in our solver.
- τ' can be generated by the program, i.e., for each thread t, $\tau' \downarrow_t$ can be generated by the corresponding thread code. This claim requires that, for any two adjacent events e and e' in $\tau' \downarrow_t$, (1) if e is not a branch, $inst^e$ and $inst^{e'}$ should be adjacent in the code; (2) if e is a branch, $inst^e$ and $inst^{e'}$ should be adjacent in terms of control flow and the $inst^e$ should be evaluated as the boolean value implied by $inst^{e'}$. We derive the trace through either concrete execution or precise modeling of unexplored branches, which satisfies the requirements and guarantees the correctness of the claim.

Discussion 1 (Detection Capability). *Our technique has stronger detection capability than existing techniques [18], [7], [14].*

First, to the best of our knowledge, none of the existing techniques automatically explore the un-executed branches by relaxing the schedules, whereas Recipe enables this. Second, assume we disable the exploration of un-executed branches and all techniques start with the same trace. Our technique still explores more traces. Figure 5 illustrates the difference, where two vertical lines represent the progress of two threads and the circles with numbers denote events with ids. Existing techniques require the reads to read the same values, and hence enforce the schedule order $e_5 \rightarrow e_6$ and $e_2 \rightarrow e_3$. We allow the reads to read different values from different writes as long as the feasibility is preserved. Comparatively, we enforce a weaker schedule order $e_4 \rightarrow e_6$ and $e_2 \rightarrow e_3$, which allows more scheduling, e.g., the concurrent execution of e_7 (or some event after e_7) and e_5.

However, Recipe has not reached the maximum bound of sound exploration. The underlying reason is that, we still enforce the schedule for preserving the base object, e.g., $e_2 \rightarrow e_3$. It is possible to further relax the schedule so that e_3 reads from e_1, but this requires challenging reasoning of the field accesses as their base objects become uncertain. We plan to explore this idea in future work.

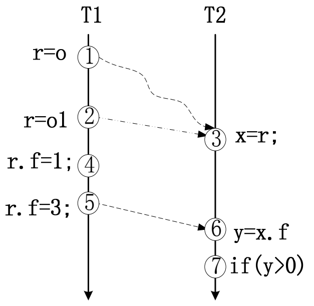


Fig. 5. Illustration of Detection Capability.

VI. Evaluation

Our evaluation focuses on the effectiveness and scalability. To measure the effectiveness, we compare with the predictive analysis, RV [7]. We chose RV for two main reasons: (1) RV is the only open source predictive analysis, (2) RV represents the state-of-the-art, which is theoretically proven to have higher detection capability than other approaches.

Prototype Implementation We use the Z3 SMT solver for constraint resolution [4]. For static analysis, to discover un-executed branches, we utilize the SOOT compiler framework [19].

For scalability, as well as to constrain memory footprint, we have applied several engineering optimizations. In specific, (i) events are stored in a database rather than in an in-memory buffer; and (ii) the analysis is split into several segments, between which we restart the VM and resume the last snapshot, such that the different logs are fused together at the end.

To improve the scalability in constraint solving, we again apply a splitting optimization. The full trace is split into N equal-length subtraces or "time windows" (in our experiments, $N = 1,000$). The values of local and shared variables are then passed at the end of a given window as the inputs to the next window. For lack of space, we omit the details, and instead refer the reader to Huang et al. [7], who apply a similar optimization.

Evaluation Method We conducted our experiments on a set of 16 applications, which were also used to evaluate RV. Specifically, the set includes large applications such as Jigsaw, Xalan and Lusearch. To handle large applications that make use of reflection, we applied the Tamiflex tool [1], which replaces reflective calls with the concrete method invocations recorded in the observed run. We omit the benchmark eclipse because the current version of Tamiflex leads to abnormal execution after the instrumentation, which throws an exception when the main service is started. By applying RECIPE to the abnormal execution, we identify only 3 races, similarly to the report of RV [7]. In addition, the montecarlo benchmark requires an external input that we downloaded from the internet and simplified. For the large applications, we utilized the most lightweight available configuration. To further control the cost of our experiments, we monitored each execution for at most 10 minutes.

As the detection capability depends on the observed run, for fair comparison we monitor the execution once by recording all necessary information required by both approaches, and then apply both techniques to the monitored run. Besides, our reported data for RV may be different from the original report for two reasons: (1) the different observed runs lead to different sets of races, (2) the original implementation of RV contains a bug in branch identification, which leads to misses of many branches and incorrect reduction of dependences during the analysis. We confirmed the bug with the author and fixed the bug in our experiments.[5]

Our experiments and measurements were all conducted on an x86 64 Thinkpad W530 workstation with eight 2.30GHz Intel Core i7-3610QM, 16GB of RAM and 6M cache. The workstation runs Ubuntu 12.04, and has the Sun 64-Bit 1.6.0_26 JVM installed.

A. Effectiveness

Table II shows the main results of our analysis, which includes four sections: Trace (details about the trace), Races (detected races), Difference (comparison with RV) and Running time (the time taken by the analysis).

The Trace section includes the number of threads (Th), the number of shared reads ($Reads$) and shared writes ($Writes$), the subset of shared reads that read the base object references ($Base$), where the base object references are references used as the base/target in the following field reference or method invocation, the number of branches (Br), the synchronization events ($Sync$) and the total number of events ($Total$), which includes local accesses in addition to the aforementioned events. Specifically, branch events are reported in the form A/B, where A refers to the number of branches used by our analysis and B refers to the number of branches used by RV. To ensure that the predicted run sees the same base object at each shared read/write, RV inserts an artificial branch immediately in front of each shared access (and array accesses). We do not use such artificial branches.

We make a few interesting observations about the Trace section. The non-local events, i.e., all the events listed in Table II, occupy around 30% of the total trace in the first 7 benchmarks, but occupy less than 1% of the trace in the rest of the benchmarks, which have relatively more complex logic. Reads of the shared base objects occupy a small portion ($\frac{1}{10}$-$\frac{1}{3}$) of the total shared reads. The rest of the shared reads read only primitive values or references not involved in field accesses (e.g., references involved in nullness checking). Besides, our analysis involves significantly less branch events as compared to the RV approach. This difference plays an important role in the detection, which we will explain shortly.

The $Races$ section shows the number of races detected by Recipe-s, i.e., Recipe without exploring un-executed paths; Recipe, i.e., the full-fledged version; and RV. Comparison between Recipe-s and Recipe reveals that Recipe finds >100 more races, which demonstrates the strength of exploring un-executed paths. Intuitively, Recipe-s predicts based on a single trace, while Recipe predicts based on multiple traces containing different execution paths. We also compare between Recipe and RV version, as shown in the $Difference$ section, where $Diff$ shows the races found by Recipe but missed by RV while $Diff'$ shows the races found by RV but not Recipe.

First of all, Recipe finds >250 more races than RV. There are several reasons for that: (1) Recipe can reason about the accesses in the un-executed paths, while RV and Recipe-s can only reason about the accesses in the executed paths. (2) Recipe or Recipe-s relax the scheduling even if it changes variable values in the observed run. Recipe allows the majority of the shared reads, i.e., the reads of primitive values, to freely read a different value from a different write as long as the value leads to the same branch decisions. RV, however, requires them to read the same values as in the observed run. (3) A critical optimization proposed by RV is to preserve dependences only for the reads before the preceding branches of the racy events, rather than all the reads. However, this optimization is underplayed by the fact that RV introduces huge amount of artificial branches, i.e., one before each shared field access, which ensures the use of the same base objects. We get rid of such artificial branches and instead, rely on the small amount of base read events to ensure the use of the same base objects (Section III and Section IV). Our strategy reduces the number of branches greatly and amplifies the effectiveness of the optimization. We also carried out case studies (Section VI-B) to better illustrate the scenarios.

Another interesting observation is that although Recipe should produce all races found by RV in theory, Recipe may miss races found by RV in practice (Column $Diff'$), i.e., Recipe is not strictly more effective than RV in practice. The reason is the limitations of the underlying constraint solver: (1) the solver cannot compute constraints with very complex

[5] We have created an in-depth report on the bug, along with a witness test case. (See: https://sites.google.com/site/recipe3141/.)

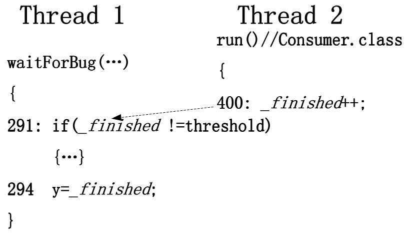
arithmetic operations, and (2) the solver does not support some program constants such as the scientific notation $3E-10$. In our empirical evaluation, we encountered only one benchmark, account, where these issues affect our detection capability.

The last section, Running time, compares the analysis time for both approaches. We find our approach is significantly slower than RV, e.g., RV often finishes within 200 seconds, while our approach may take more than 1 hour. This is because Recipe has a larger search space and needs to reason about the computation among variables inside the local access events, while RV needs to only reason about the order relations among the events.

B. Case studies

We manually inspected the reported races over small applications to gain better understanding about our approach.

Relaxing Values Figure 6 demonstrates the relaxation of inter-thread dependence enabled by our approach. The code is from the benchmark bbuffer, where the line numbers are marked. Huang et al. [7] detect the race between lines 291 and 400, but fail to detect the race between lines 294 and 400. The reason is that in the observed run, the execution follows the order 400, 291 and 294. For the event at line 294, its preceding branch at line 291 reads from line 400. Therefore, Huang et al. [7] require the predicted run to preserve the dependence between line 291 and line 400, so that line 291 reads exactly the same value and the branching expression evaluates to the same truth value. The dependence enforces the order constraint, 400 → 291, which further enforces the order 400 → 291 → 294. Our approach does not require the existence of such dependence. Specifically, we allow line 291 to happen before line 400 in the predicted run as long as the value read by it leads to the same branch decision, which is true in this case. As a result, there is no order constraint between line 294 and line 400, and the two form a race. We were able to replay this race easily using the eclipse IDE breakpoints.

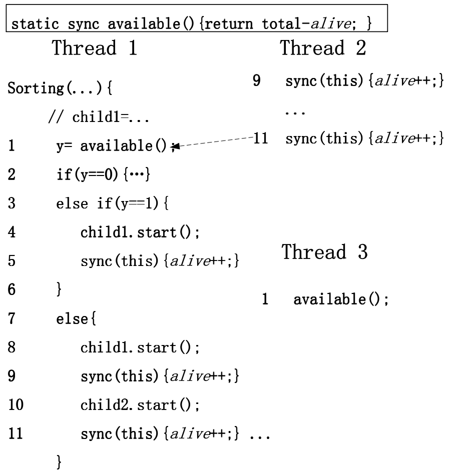


```
  Thread 1               Thread 2
                         run()//Consumer.class
waitForBug(…)
                         {
{
                         400: _finished++;
291: if(_finished !=threshold)
     {…}
294  y=_finished;
}
```

Fig. 6. Relaxation of Inter-thread dependence

Relaxing the Paths Figure 7 demonstrates how our approach relaxes the scheduling to account for unexecuted paths in mergesort. All threads invoke the method Sorting, which recursively starts two children threads if there are two more available entries in the thread pool (lines 8-11), or starts one child thread if there is only one available entry (lines 4-5). The availability is computed through the static method available with the constant total, which is equal to 5. The shared variable alive denotes the used entries.

Initially there is one sorting thread. After it starts Thread 1 and Thread 2, there are three threads alive and only two more entries are available. Suppose that in the observed run Thread 2 consumes both thread entries and starts the child Thread 3 and Thread 4 (not shown), updating alive to 5. Then Thread 1 cannot execute the branch at lines 4-5. Huang et al. [7] require the predicted run to preserve the dependence denoted by the dotted line since this dependence affects the branch condition at line 2. As a result, the predicted run follows the same branch decision and cannot reason about the code at lines 4-5. Our analysis does not suffer from this limitation. Instead, it allows the predicted run to reason about the unexplored code. Specifically, it does not need to preserve the dependence from line 11 to line 1, and it allows line 1 (Thread 1) to read from line 9 (Thread 2). As a result, the branch condition guarding the unexplored branch is evaluated to be true, enabling the unexplored path in the constraint solver. Finally, the solver identifies the race between line 5 (Thread 1) and line 1 (Thread 3). Note that the two lines are synchronized on different locks [6].



```
static sync available(){return total-alive; }
     Thread 1                     Thread 2
                              9   sync(this){alive++;}
Sorting(...){
     // child1=...                ...
1      y= available();       11   sync(this){alive++;}
2      if(y==0){…}
3      else if(y==1){
4          child1.start();
5          sync(this){alive++;}  Thread 3
6      }
                              1    available();
7      else{
8          child1.start();
9          sync(this){alive++;}
10         child2.start();
11         sync(this){alive++;} ...
       }
```

Fig. 7. Relaxation of Paths

VII. Threats to Validity

A Java method can consist of up to 65535 bytecode instructions. Therefore, we count the number of bytecode instructions inside each method. If the resulting number exceeds 65535, we avoid instrumenting the method. The consequence is that we will potentially miss races inside a method for which we skipped instrumentation. In this case, we specify the variables read from the method to be equal to their concrete values in the constraints. The constraint solver can proceed safely without being affected by such methods.

Recipe lacks support for certain operators. In our current implementation, we do not support bit operations such as & and <<, which accounts for most of our misses.

[6] We abbreviate the synchronized keyword as sync.

TABLE II
RELAXED ANALYSIS

	Trace							Races			Difference		Running time (sec)	
Benchmarks	Th	Reads	Writes	Base	Br	sync	Total	RV	Recipe-s	Recipe	Diff	Diff'	Recipe	RV
critical	3	13	7	5	2/13	6	78	8	8	8	0	0	8	2
airline	11	45	15	4	32/82	30	317	9	9	9	0	0	490	4
account	3	46	21	10	3/47	6	227	5	2	5	3	3	41	4
pingpong	4	7	7	3	0/15	6	111	1	1	1	0	0	19	1
bbuffer	4	640	118	10	634/1.1K	217	3.3K	9	13	25	21	5	62	5
bubblesort	26	1.3k	966	121	155/2.8K	322	8.4K	7	7	7	0	0	3295	3
bufwriter	5	165	52	75	16/130	44	525	2	4	10	8	0	63	9
mergesort	5	38	33	5	15/472	28	1.7K	3	3	10	7	0	37	5
raytracer	2	31	5	12	314/8,2K	676	94.5K	4	4	6	2	0	47	2
montecarlo	2	5	86	2	1.9K/38.2K	21.1K	1.9M	1	1	4	3	0	1	17
moldyn	2	605	61	104	19.6K/52.6K	62	203.4K	2	6	14	12	0	2842	1
ftpserver	28	684	299	71	4.4K/233.3K	78.2K	3.9M	57	99	152	108	13	811	153
jigsaw	12	525	702	211	63.2K/467.9K	86.7K	5.5M	8	17	23	15	0	33	7
sunflow	9	2.1K	1.3K	473	201.3K/827.0K	50K	7.1M	20	38	78	69	11	4520	22
xalan	9	1.4K	0.9K	209	15.7K/103.2K	190.1K	6.6M	2	2	6	4	0	5317	10
lusearch	10	2.3K	0.5K	715	22.2K/164K	93.2K	9.1M	14	27	49	38	3	5430	8

VIII. RELATED WORK

A. Race Detection Techniques: Broad Survey

Initial attempts to address the challenge of race detection focused on built-in synchronization primitives [16], [15], [20], [2]. These include locks as well as the wait/notify and start/join primitives. Notable among these efforts is the *lockset* analysis, which considers only locks [16]. Because the derived constraints are partial, permitting certain infeasible event reorderings, lockset analysis cannot guarantee soundness [12].

A different tradeoff is struck by the *happens-before (HB)* approach [3], [5], [11], which is founded on Lamport's HB relation [9]. In HB analysis, all synchronization primitives are accounted for, though reordering constraints are overly conservative.

Recently there have been successful attempts to relax HB constraints, including the hybrid analysis [13], which permits both orderings of lock-synchronized blocks, and the *Universal Causal Graph (UCG)* representation [8], which also enables both orderings but only if these are consistent with wait/notify- and start/join-induced constraints.

B. Predictive Trace Analysis (PTA)

Given a concurrent execution trace t, PTA derives new traces to witness the data races. PTA is founded on the notion of sound causality, as it considers feasible reorderings of the input trace that prove a candidate data race as such.

Wang et al. [21] pioneer the symbolic predictive analysis, which our approach also belongs to. However, the approach lacks support of heap encoding, therefore, cannot be applied to real-world applications. The first PTA that applies to real-world applications is proposed by Smagardakis et al. [18]. In their original work, both synchronization constraints and inter-thread dependencies are preserved, where inter-thread dependencies are respected by only allowing reorderings that leave the dependence structure exhibited by the original trace intact. This ensures that the values of shared memory locations remain the same, which secures the soundness argument.

Said et al. [14] and Huang et al. [7] proposed PTAs that are also sound. Said et al. perform symbolic analysis of the input trace, and then utilize an SMT solver to search for schedules that establish the presence of data races. Soundness is guaranteed by their ability to precisely encode the semantics of sequential consistency. Inspired by Said et al., Huang et al. present an improvement, whereby control-flow information is encoded into the constraint system so as to relax value flow dependencies as long as the values do not flow into the control-flow event. ExceptionNULL [6] is another example of sound PTA. Its goal is to detect null dereferences rather than data races.

RECIPE is similar to Said et al. and Huang et al. in that it also makes use of an SMT solver based on symbolic encoding of the input trace. Still, the two sound relaxations that RECIPE features, which enable (i) value- rather than dependence-based reasoning about data flow and (ii) consideration of unexplored branches, are strictly beyond the scope of both techniques. In Section VI, we demonstrate via direct comparison with Huang et al. the substantial improvement in coverage thanks to these relaxation methods.

C. Maximality Claim

Serbanuta et al. [17] and Huang et al. [7] both claim maximality of the detection capability, but under different assumptions. Assuming branch events are not modeled in the trace, [17] guarantees maximality. Given few information in the trace, it conservatively requires the shared read to obtain the same value in the predicted run to ensure soundness. However, if branch events are modeled, the technique does not guarantee maximality. Assuming the branch events are modeled but local assignments are not recorded, [7] guarantees maximality. Compared to [17], it allows some events to read different values if the values do not flow into the following branch conditions, as they do not affect the following branch decisions. However, it requires the events that affect the branch decisions to read exactly the same values. In practice, these assumptions are too restrictive because local assignments can be easily collected in practice.

Assuming local assignments are available in the trace, both techniques cannot guarantee maximality. As illustrated in our example and discussed in Section V, our analysis allows higher detection capability.

IX. Conclusion

We develop RECIPE, a relaxed predictive analysis technique. It has two critical relaxations on existing predictive analysis that substantially enlarge analysis coverage. The first relaxation allows variables to take different values as long as doing so does not affect trace feasiblity; the second relaxation allows exploring neighboring unexecuted paths as long as doing so does not affect the soundness guarantee. Our results show that RECIPEis able to detect 2.5X as many races as compared to the state-of-the-art, without any false positives.

References

[1] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE*, 2011.

[2] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, 2002.

[3] 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.

[4] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, 2008.

[5] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *PPoPP*, 1990.

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

[7] J. Huang, P. O. Meredith, and G. Rosu. Maximal sound predictive race detection with control flow abstraction. In *PLDI*, 2014.

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

[9] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM '1987*.

[10] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 1979.

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

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

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

[14] M. Said, C. Wang, Z. Yang, and K. Sakallah. Generating data race witnesses by an smt-based analysis. In *Proceedings of the Third International Conference on NASA Formal Methods*, 2011.

[15] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated type-based analysis of data races and atomicity. In *PPoPP*, 2005.

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

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

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

[19] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research*, 1999.

[20] C. von Praun and T. R. Gross. Object race detection. In *OOPSLA*, 2001.

[21] C. Wang, S. Kundu, M. Ganai, and A. Gupta. Symbolic predictive analysis for concurrent programs. In *FM*, 2009.