RECIPE: Relaxed Sound Predictive Analysis

***Abstract*—The abstract goes here.**

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 analysese 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 [**?**] was proposed. It has the capabilities of achieving soundness as dynamic analyses and reasoning about properties that did not occur during 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 reaons 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 accsses. 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 pertubation*. The rationale is that it becomes as difficult as static analysis if we allow these critical runtime information change. Preditive 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).

Despite its effectiveness, we observe that existing predictive analyes 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.

```
        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

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. the same branch is taken). However, the relaxation would allow us to execute lines 4 and 5 before line 2, 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 the else branch was not executed originally, its effect can be precisely modeled as the value 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 XXX times more races than existing predictive analysis, while guaranteeing soundness.

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

Our contrituions 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 show XXX.

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

To facilitate our presentation, we first introduce some terminologies used throughout this paper. Intuitively, a trace is a sequence of events recorded during the observation run.

Event An event, $e =< t, id, inst >$, is a concrete representation of the runtime execution of a static instruction $inst$.

- t refers to the thread that issues the event e, denoted as t^e.
- id refers to the id associated with each event. The key property of id is *uniqueness*, i.e., any two events in the trace own different ids. Unless otherwise specified, we use the index of an event in the trace as its id, which satisfies the above property. Throughout this paper, we denote an event as e_{id} with the id as the subscript.
- $inst$ is the static instruction. The instructions are three-address instructions involving at most three operands, which modern compilers commonly support. Specifically, we are interested in the types of instructions listed in Table I. When the variable does not appear on the left hand of an equation, such as y in $x.f = y$, it may refer to a variable, a constant or event object creation expression $new(...)$. The *bop* stands for the binary operator, which may refer to $+, -, *, /, \%, \wedge, \vee$ in the assignment, or refer to $<, >, =, \wedge, \vee$ in the branch. The target of the branch event is not important in our scope, therefore, we may abbreviate the branch instruction as the boolean expression afterwards. The listed instructions suffice to represent all trace events of interest. This is because a concrete finite execution trace can be reduced to a straight-line loop-free call-free path program (argument passing of the method call is modeled as assignments and the virtual call resolution is modeled as branches). This standard form of simplification preserves all the data-race-related information contained in the original trace.

Trace A trace τ is a sequence of events. It can also be represented more comprehensively as, $\tau =< \Gamma, \{\tau_{t_1}, \tau_{t_2}, \ldots \tau_{t_n}\}, O, \theta >$.

- Γ refers to the set of threads involved in the trace, which include t_1, t_2, …, t_n.

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

- $\{\tau_{t_1}, \tau_{t_2}, \ldots \tau_{t_n}\}$ 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. For example, e_i is scheduled earlier than e_j if $O(e_i) < O(e_j)$.
- θ is a function that maps every variable to a value. For example, given an event e executing the instruction $x = y+z$, the mapping recorded in the trace may be $\theta(x) = 3$, $\theta(y) = 2$, $\theta(z) = 1$. If the variable x is defined in two events, the mapping for x becomes ambiguous. To avoid the ambiguity, we conduct a pre-processing (Section III) that produces the SSA form of the trace, in which each variable is defined exactly once.

B. Constraint System

The predictive analysis takes the program and a trace as the input. Suppose it starts with the following trace of the program in Figure 1: $e_1e_2e_3e_4e_5$, where the line number is used as the event id (subscript). A potential race is between the pair, (e_2, e_5), as they access the shared variable x from different threads and one is a write.

Given the trace, $\tau =< \Gamma, \{\tau_{t_1}, \tau_{t_2}, \ldots \tau_{t_n}\}, O, \theta >$, the predictive analysis computes a new trace, $\tau' =< \Gamma, \{\tau_{t_1}, \tau_{t_2}, \ldots \tau_{t_n}\}, O', \theta' >$. τ' preserves the same event sequence for each thread, i.e., T_1 still executes $e_1e_2e_3$ and T_2 still executes e_4e_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 them briefly in the following. The full details are explained in Section III and the feasibility 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 The potential race pair involves two accesses of the same shared location from different threads, where at least one access is a write. Given any potential race

[2]We omit the synchronization constraints in this paper, which are already well explained in existing techniques [14], [**?**].

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 The predictive analysis needs to preserve the same event sequence for each thread, which further requires

- **Control Flow Constraints** Each thread takes the same control flows (or branch decisions) to reproduce the events. For the running example (ignoring the statements in red), this yields:

  $$\theta'(R^4_{\mathrm{y}}) > 2 \equiv \mathbf{true}$$

  That is, in the predicted trace τ', the value of variable 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 flows. We yield the following formula:

  $$O'(e_1) < O'(e_2) < O'(e_3) \wedge O'(e_4) < O'(e_5)$$

  That is, in the predicted trace τ', the three events from T_1 are ordered in the same way as in τ, and similarly, the branch event e_4 in T_2 still executes before the event e_5 inside the branch body.
- **Intra-thread Value Constraints** The events should respect the value constraints imposed by each instruction. For our running example, we obtain:

  $$\theta'(W^0_{\mathrm{x}}) = 0 \wedge \theta'(W^0_{\mathrm{y}}) = 0 \wedge \theta'(W^1_{\mathrm{y}}) = 3$$
  $$\wedge\theta'(W^2_{\mathrm{x}}) = 1 \wedge \theta'(W^3_{\mathrm{y}}) = 5$$

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

Inter-thread Value Constraints The inter-thread value 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

$$\begin{array}{ll} & (\theta'(R^4_{\mathrm{y}}) = \theta'(W^0_{\mathrm{y}}) \wedge O'(e_4) < O'(e_1)) \\ \bigvee & (\theta'(R^4_{\mathrm{y}}) = \theta'(W^1_{\mathrm{y}}) \wedge O'(e_1) < O'(e_4) < O'(e_3)) \\ \bigvee & (\theta'(R^4_{\mathrm{y}}) = \theta'(W^3_{\mathrm{y}}) \wedge O'(e_3) < O'(e_4)) \end{array}$$

Notice, importantly, that the formula associates the value constraints with the scheduling order constraints. As an example, $\theta'(R^4_{\mathrm{y}}) = \theta'(W^1_{\mathrm{y}}) \wedge O'(e_1) < O'(e_4) < O'(e_3)$ 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.

Unexplored Branches Beyond the encoding steps so far, which focus on the given trace, we can often encode constraints along unexplored branches. In our running example, this is essential to discover the race between lines 2 and 7. We conduct the symbolic execution to exercise the unexplored branches, which returns a set of traces representing possible executions of the unexplored branch. Each trace is combined with the original trace to find races involving the accesses in the unexplored path. For the racy events, e_2 and e_7, we first specify the race condition constraint $O'(e_2) = O'(e_7)$. In addition, we negate the path condition, thereby obtaining $\theta'(R^4_{\mathrm{y}}) > 2 \equiv \mathbf{false}$ in place of $\theta'(R^4_{\mathrm{y}}) > 2 \equiv \mathbf{true}$. We also model the other necessary constraints such as the intra-thread order constraint $O'(e_4) < O'(e_7)$, similar to the above analysis.

Constraint Solving The formuals from the different encoding steps are conjoined and sent to the 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.

Highlights In this example, our analysis finds two more pairs of races than existing analyses [14], [**?**], which include (e_2, e_5) and (e_2, e_7). The key insight for finding the first race is, we do not require e_4 to read from e_3 to retain the original value, as what existing approaches do, rather, we allow e_4 to read from e_1 while still preserving the feasibility of the branch. 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. The key insight for finding the second race is, we explore the unexplored branches and ask the solver to compute a feasible schedule that exercises them. The neighboring branches are likely to contain racy events if the executed branches contain racy events.

III. Relaxation of Flow Dependencies

Given the 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 (the order of events inside each thread should remain unchanged) to witness the race. The key insight underlying our first relaxation is that, the read events in τ' are allowed by our analysis to read different values from different writes (than in trace τ), but they are required by existing approaches to read the same values as in τ. Therefore, our relaxation allows more schedules, which are likely to expose more races. 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 in 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. Another side effect is that the def/use chains become explicit in the SSA form, which simplifies the following analysis steps.

RECIPE handles the local assignment and heap accesses differently.

- **Local Assignment** The SSA form for local assignment resembles the SSA form of static instructions in compiler optimization, except that the loops and recursions are fully resolved in a concrete trace, obviating the need for the Phi node. 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. Consider the example in Figure 2, where the line number is also used as event id, the local variable x defined at line 4 is renamed as x^4 in the trace, while x defined at line 9 is renamed as x^9, which is used in the following two events. At line 5, where the object $o3$ is created, we define a reference variable w^5. Internally, the object $o3$ is encoded as a unique integer that indicates its heap address.
- **Heap accesses** The def/use relation between local heap accesses can be locally determined. Therefore, we treat them in the same way as local assignment. 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 access to `y` at line 4 in Figure 1 can either obtain the definition before line 1 or at line 1 or at line 3. Therefore, we introduce two symbols to represent the shared read and shared write respectively and use them to establish different links flexibly. 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 the x. Similarly, the subscript $o.f$ is interpreted as a heap location, i.e., the field f of the object o.

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

B. Constraint System

Based on the SSA form of the original trace τ, we build the constraints to compute a new trace τ' with the new schedule O' and 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, therefore, e_{11} must happen after e_2. Existing approaches enforce the same value dependence between e_3 and e_8, leading to the miss of the race between e_2 and e_{11}. Our analysis can derive a new trace τ' to find the race by relaxing the dependence. In τ', e_8 reads from a different event e_1, producing the new value mapping, $\theta'(y^8) = 3$. Accordingly, at the event e_9, the value mapping for x^9 is also updated as 13 ($\theta'(y^8)$=3, $\theta'(z^7) = 10$), enabling the 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 intentionally, as they are well explained in all existing predictive analysis techniques [14], [4].

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 Following the standard definition, pair (e_i, e_j) of events forms a race iff (1) e_i and e_j are accesses of the same location ℓ by different threads, (2) at least one of them writes to ℓ, and (3) e_i and e_j run concurrently. We refer to the candidate pair throughout this section as (candidate) *racy events*.

We first identify all candidate pairs of events that access the same location from different threads (and at least one is a write). We then check each pair separately. The checking is achieved by encoding all necessary constraints and invoking a constraint solver. The first constraint asserts the feasibility of the concurrent execution of the racy events:

$$O'(e_i) = O'(e_j)$$

In Figure 2, the race condition for the race pair, (e_2, e_{11}), is $O'(e_2) = O'(e_{11})$

Intra-thread Constraints Given the candidate race pair, e_i and e_j, suppose e_i is after e_j in the original trace τ. The predictive analysis reschedules the events in the prefix of e_i, i.e., prior 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 the events in the prefix.

The predictive analysis by design requires that each thread should follow the same event sequence prior to e as in the

original run. The preservation of the event sequence 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 set of events. Specifically, we only need to reason about the branches prior to e_i.
  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 run τ' as in the original trace τ. Importantly, Unlike existing analyses [14], [4], we do not pose the requirement that the values flowing into branching statements remain the same, but adopt the relaxed requirement that the evaluation of branching expressions remains the same. Consider the example in Figure 2, given the branch if(x^9¿11) at $e_1$0, the original mapping is $\theta(x^9) = 12$. Existing analyses require the same value mapping for x^9 in the predicted run, while we allow a different value mapping, e.g., $\theta'(x^9) = 13$, which retains the same truth value for the branch expression.
- **Intra-thread Order Constraints** The events in the sequence should follow the same order as in the original trace. More formally, we require that

$$\forall e_m, e_n \in prefix(\tau, e_i), s.t., \quad t^{e_m} = t^{e_n}. \\ O(e_m) < O(e_n) \quad \Rightarrow O'(e_m) < O'(e_n)$$

  This constraint specifies that two events from the same thread should follow the same order as reflected by the ids of the events.
- **Intra-thread Value Constraints** The value mapping of the variables should not contradict the value constraints imposed by each instruction. Without loss of generality, suppose the instruction is a local assignment in the form of $x = y + z$, then we require that

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

  Remember 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 consistently. In the above, we use the local assignment event for demonstration, in general, the intra-thread value constraints should also be specified for the heapr/heapw events.

Inter-thread Value Constraints for Relaxation We now move to the first novel feature of RECIPE, which is its ability to explore execution schedules that depart from the value flow exhibited in the original trace. More precisely, RECIPE is able to relax value flow dependencies in the original trace: a read access may read a different value from other write events, as long as the read value enforces feasibility of the following execution. This is strictly beyond the coverage potential of existing predictive analyses, which restrict trace transformations to ones where any read access must read the same value (often from the same write event) as in the original trace.

T_1	T_2
1: x^1 = new();// creates o1	
2: x^2= new();// creates o2	
3: $W_y^3 = x^1$;	
4: $W_y^4 = x^2$;	
5: $W_{o2.f}^5 = 5$;	
	6: $z^6 = R_y^6$;
	7: $w^7 = R_{o2.f}^7$;

Fig. 3. A trace.

To ensure feasibility under relaxation of flow dependencies, we need to secure the flow between the read/write with the execution schedule. For example, in Figure 2, for the read at e_8 to read from the write t 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_ℓ^m of location ℓ at the event e_m with set $\mathcal{W}$ of matching write events (i.e., events including write access to ℓ), takes the following form:

$$\bigvee_{e_n \in \mathcal{W}} \quad (\theta'(R_\ell^m) = \theta'(W_\ell^n)) \\ \bigwedge \quad O'(e_n) < O'(e_m) \\ \bigwedge_{e_p \in \mathcal{W} \setminus \{e_n\}} \quad (O'(e_p) < O'(e_n) \vee O'(e_m) < O'(e_p))$$

This disjunctive formula iterates over all matching write events, and demands for each 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 relaxation of flow dependencies 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 the 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. Suppose the event e_6 reads from e_3 due to the relaxation. Then e_7 and e_5 no longer share the same base object (or the same location). Even worse, we do not know which events e_7 matches, because the base object is no longer known.

Invariant Base Object Constraints To address this challenge, we enhance the constraint system with the requirement that heap objects that are dereferenced in field access statements before the racy events retain their original address in the predicted trace. This achieves 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. Third, sharing between threads of heap locations remains unchanged because each location is accessed by the same number of threads given that the base object is the same.

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. Exploration of Unexecuted Branches

We now switch to the second feature of RECIPE, which is its ability to reason about neighboring branches that are not executed in the original trace τ. At the high level, we replace the constraints that model the executed branch to the constraints that conservatively model the possible executions in both branches. We leverage the static analysis to realize this relaxation.

We start with a branch event e_b in the trace, without loss of generality, suppose its true branch is taken, we need to explore the false branch (thread-locally with the same thread as e_b) and add constraints to model it. As the static analysis has some known challenges, we make several simplifying assumptions: (1) the unexplored branch should not contain loops or recursions because the loop bound may depend on some shared variable which is statically unknown; (2) every base reference variable should be fully resolved to a concrete object, as required by our analysis (Section III). We apply dataflow analysis (reaching definitions) to judge: If the base reference depends on some shared read in the explored branch, of which the value is statically unknown, it cannot be statically resolved to an object; If the base reference depends on multiple definitions from different paths, it cannot be statically resolved to an object. Otherwise, it can be resolved. (3) We assume the unexplored branch does not contain array access because the array accesses often use the index expressions that are statically unknown. Similarly, as the collections often use array for implementation, we assume they are also absent. (4) We inline the method calls (for two levels) in the unexplored branch, similar to the base object, we assume the caller object

```
6: if(x^5<10)
7: W_S^7=0;
8: else
9:    z^9=R^9_{o1.f};
10:   if(x^5<z^9)
11:    W_S^11=0;
12:   else
13:    W_S^13=l^id_{o2.f};
```

Fig. 4. Running Example (shared variables are in bold font).

can be fully resolved statically. If any of the above assumptions is violated, we avoid exploring the unexplored branches and adopt the purely dynamic solution in Section **??**sec:relax1. We use the example in Figure 1 for demonstration.

Events . We traverse the control flow graph of the unexplored branch and generate an event, $< t, id, inst >$, to represent the possible execution of each instruction $inst$. The id can be any unique value, where the uniqueness is the requirement. We simply use the line number as the event id in our example. As the unexplored branch does not contain loop/recursion, we establish the one-to-one 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 unexplored branch and combine its SSA form with the existing SSA form of the explored branch to produce the final SSA form. The SSA transformation is standard, but special issues need to be considered.

- **Local primitive variables** Some local variable is defined before the branch event e_b, of which the SSA form is already in the trace τ, we reuse the SSA form. In addition, the SSA transformation may introduce Phi node, e.g., $y = Phi(y_1, C_1, y_2, C_2, \ldots y_i, C_i)$, meaning that y is equal to y_i in the condition C_i. The combination of SSA forms of both branches may require further Phi node when both branches define a variable. We insert the Phi node to the trace and replace the following use accordingly.
- **Local Heap Accesses** Given the heap access of $x.f$, remember 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 the special symbols to denote the shared read/write.

The SSA form for the branch at line 6 is as follows, where the base variable w at line 13 is replaced to an object $o2$ and the local heap access $w.f$ is encoded as a local primitive variable.

Constraints During the traversal, we also record the shared access events. For the set of accesses S_1 of the location ℓ, we also find the accesses S_2 of it 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 constraint is the race condition constraint, e.g., $O'_4 = O'(e_1 1)$ specifies that the event at line 4 may form a race with the event at line 11, which is not executed in the original trace.

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 all branches inside the branch structure may be taken, which is trivially $true$. The solver automatically reasons about which branch should be taken, as indicated in the solution. Note that only one of the two branches can be taken because the conditions for them contradict.

Intra-thread Value Constraint Intra-thread value constraint is the core constraint that realizes the exploration different branches. It asserts the value constraints imposed by the assign/heapr/heapw events take effect only when the guarding condition is satisfied. More formally, we have

$$cons(n) = \begin{cases} (\bigwedge_{n_1 \in n_{if}} (expr(header_n) \rightarrow cons(n_1))) \\ \bigwedge(\bigwedge_{n_2 \in n_{else}} (\neg expr(header_n) \rightarrow cons(n_2))) \\ \qquad\qquad \text{if n is a branch} \\ equation(n) \\ \qquad\qquad \text{otherwise} \end{cases}$$

Here, $expr(header)$ is the helper function that returns the condition associated with the header. $equation(n)$ returns the equation inside each event, which imposes the value constraint. In the first case when the instruction n is a branch, we assert the value constraint imposed by each instruction in the if-branch can take effect only when the if-branch is taken, i.e., when $expr(header_n)$ is evaluated as true, and the value constraint imposed by instruction in the else-branch can take effect only when the else-branch is taken. In the second case, when the instruction n is a normal instruction that contains the equation, we assert the equation as the value constraint. Note that the constraint is defined recursively because a branch may contain another branch structure.

Consider the example in Figure 4, the constraint is,

$$\begin{aligned} &(\theta'(x^5) < 10 \rightarrow \theta'(W_S^7) = 0) \wedge (\theta'(x^5) \geq 10 \rightarrow \theta'(z^9) = \theta'(R_{o1.f}^9) \\ &\wedge(\theta'(x^5) \geq 10 \rightarrow \\ &\quad (\theta'(x^5) < \theta'(z^9) \rightarrow \theta'(W_S^{11}) = 0 \\ &\quad \wedge\theta'(x^5) \geq \theta'(z^9) \rightarrow \theta'(W_S^{13}) = \theta'(l_{o2.f}^{id})) \\ &) \end{aligned}$$

Phi node

Intra-thread Order Constraints The intra-thread order constraint captures the program order imposed by the branch structure and the instruction sequence. More formally, for each branch n, we have,

$$cons(n) = \begin{cases} (\bigwedge_{n_1 \in n_{if}} O'(header_n) < O'(n_1) < O'(end_n)) \\ (\bigwedge_{n_1 and n_3 are adjacent in n_{if}} O'(n_1) < O'(n_3)) \\ (\bigwedge_{n_2 \in n_{else}} O'(header_n) < O'(n_2) < O'(end_n)) \\ (\bigwedge_{n_2 and n_4 are adjacent in n_{else}} O'(n_2) < O'(n_4)) \end{cases}$$

Here, end_n is a node that denotes the first event after the branch. The order constraints are imposed by the program, and therefore, are unconditionally valid. That is why we do not specify the order constraints under specific branch conditions.

Inter-thread value constraints The inter-thread value constraints are identical to those in Section III. Interesting, we do not need to specify the value constraints under the specific branch condition. The underlying reason is that, the intra-thread value constraint already encodes the branch condition information, if the branch condition is not satisfied at runtime, the value constraint does not take effect and the read of a shared variable cannot propagate its value downstream, even if it is assigned to a value.

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 [8].

Theorem 1 (Soundness)**.** *The trace τ returned by our 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 [**?**], [13] 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 lock acquire and release of a lock should not be interleaved by the lock operations of the same lock, the begin 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 and the $inst^e$ should be evaluated as the boolean value indicated by $inst^{e'}$. We derive the trace through either concrete execution or symbolic execution, which satisfies the requirements and guarantees the correctness of the claim.

□

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

First, to the best of our knowledge, none of the existing techniques automatically explore the un-executed branches by relaxing the schedules, while ours 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 pictorially, where two vertical lines represent the progress of two threads and the circles with numbers denote the events with ids. Existing techniques require the reads to read the same values, therefore, 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, our technique does not identify the maximal set of traces from a single trace. The underlying reason is that, we still enforce the schedule order 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 unknown. We plan to explore this idea in future work.

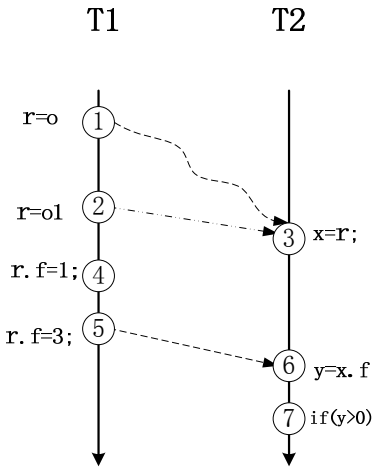

Fig. 5. Illustration of Detection Capability.

VI. Evaluation

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

Prototype Implementation In our prototype implementation of RECIPE, we used the Z3 SMT solver for constraint resolution [**?**]. For static analysis, to discover unexecuted branches, we utilize the SOOT compiler framework [**?**].

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 address the scalability limitations of the constraint solver (Z3 in our case), 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 inputs to the next window. For lack of space, we omit the concrete details of this optimization, and instead refer the reader to Huang et al. [5], who apply a similar optimization.

The trace is typically long and the constraint solver cannot scale to the whole trace. This is a known limitation to solver-based predictive analysis. Instead, we adopt the notion of time window. We divide the trace into N traces of the same length, where N is configurable parameter (1000 in our experiment). To support the time window, we need to store the values of the shared variables and local variables at the end of a window and encode such values as the inputs of the next window.

We maintain two maps, $sstore$ and $lstore$, for the shared variables and local variables respectively. Suppose the trace for the last window is τ. The following code illustrates how we maintain the $sstore$, i.e., storing the value written by the last write with each address. Beside, the variable $e.ins.left$ represents the left hand variable inside the instruction $e.ins$.

for $a \in \mathcal{A}$ **do**
  $\tau_a = W(a)$
3: $sstore.put(a, \tau_a[|\tau_a| - 1].v)$
end for
for $e \in \tau$ **do**
  if $e.isLocalAcc()$ **then**
3: $lstore.put(e.\_def, e.v)$
  else
    if $e.isRead()$ **then**
6: $lstore.put(e.\_def, e.v)$
    end if
  end if
9: **end for**

When constructing the constraints for the next window, we specify the following constraints to encode the input values.

The following formula enhances the thread interfernce constraint to include the case that the read reads from the initial values.

$$\Phi_{TI} = \Phi_{TI} \wedge R = sstore.get(e_R.a) \wedge \bigvee_{e \in S} O(e_R) < O(e)$$

.

The following formula encodes the input values of local variables.

$$\Phi_{IN} = \bigvee_{l \in lstore.keys()} l = lstore.get(l)$$

.

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 [], which replaces reflective calls with the concrete method invocations recorded in the observed run for the purpose of effective static analysis. 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 our tool to such abnormal execution, we identify only 3 races, similarly to the report of RV []. 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.

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 identification of the branches, 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.[3]

Our experiments and measurements were all conducted on an x86 64 Thinkpad W530 worksta- tion with eight 2.30GHz Intel Core i7-3610QM processors, 16GB of RAM and 6M caches. The workstation runs version 12.04 of the Ubuntu Linux distribution, and has the Sun 64-Bit 1.6.0_26 Java virtual machine (JVM) installed.[4]

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 the 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 the artificial branch immediately in front of each shared access (and array accesses). We do not use such artificial branches.

We make 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 missed by Recipe.

First of all, Recipe finds $>$150 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 breaks the inter-thread read/write dependence structure 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) An 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 **??**). Our strategy reduces the number of branches greatly and amplifies the effectiveness of the optimization. We also carried out case studies (Section **??**) 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 for this is limitations of the underlying constraint solver: (1) the solver cannot compute constraints with very complex arithmatic operations, and (2) the solver does not

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

[4]We note that Sun JDK 1.7 does not support the transformation of dacapo applications with tamiflex.

support some program constants such as the scientific notation $3E-10$. In our empirical evaluation, we encounted only one benchmark, account, where these issues manifested.

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 our approach 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.

Summary of advantage and weakness In general, our approach is flexible, which allows the relaxation of schedules and paths based on fine-grained reasoning at the level of data values. As a result, it detects many more races compared to existing approaches. On the other hand, our approach is more heavyweight. An interesting topic for future research is how to combine our approach with a lightweight approach with soundness guarantees like RV. One option is to create a staged analysis, wherein RV is first run. RECIPE is then left to run only on racy pairs not confirmed by RV. In this way, we can also tackle misses due to practical limitations of the constraint solver.

B. Case studies

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

Relaxing Inter-thread dependencies Figure 6 demonstrates the relaxation of inter-thread dependence enabled by our approach. The code is from the benchmark bbuffer, where the line number is marked. Huang et al. [] 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. [] 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, $400 \rightarrow 291$, which further enforces the order $400 \rightarrow 291 \rightarrow 294$. 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.

Relaxing the Paths Figure 7 demonstrates how our approach relaxes the scheduling to account for unexecuted paths. All threads invoke the method Sorting, which recursively starts two child 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.

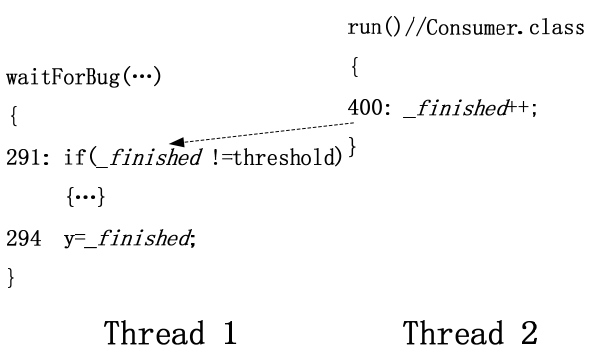

Fig. 6. Relaxation of Inter-thread dependence

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. [] 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 [5].

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 65525, we avoid instrumenting the method. The consequence is that we will potentially 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.

A second point regards support for certain operators. In our current implementation, we do not support the boolean operators, including e.g. &, as well as the bit operator $<<$, which accounts for most of our misses.

VIII. RELATED WORK

In this section, we survey related research on race detection with special emphasis on predictive trace analysis (PTA).

[5]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	Recipe-s	Recipe	RV	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	2	5	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	13	25	9	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	4	10	2	8	0	63	9
mergesort	5	38	33	5	15/472	28	1.7K	3	10	3	10	0	37	5
raytracer	2	31	5	12	314/8,2K	676	94.5K	4	6	4	2	0	47	2
montecarlo	2	5	86	2	1.9K/38.2K	21.1K	1.9M	1	4	1	3	0	1	17
moldyn	2	605	61	104	19.6K/52.6K	62	203.4K	6	14	2	12	0	2842	1
ftpserver	28	684	299	71	4.4K/233.3K	78.2K	3.9M	99	152	57	108	13	811	153
jigsaw	12	525	702	211	63.2K/467.9K	86.7K	5.5M	17	23	8	15	0	33	7
sunflow	9	2.1K	1.3K	473	201.3K/827.0K	50K	7.1M	38	78	20	69	11	4520	22
xalan	9	1.4K	0.9K	209	15.7K/103.2K	190.1K	6.6M	2	6	2	4	0	5317	10
lusearch	10	2.3K	0.5K	715	22.2K/164K	93.2K	9.1M	27	49	14	38	5	5430	8

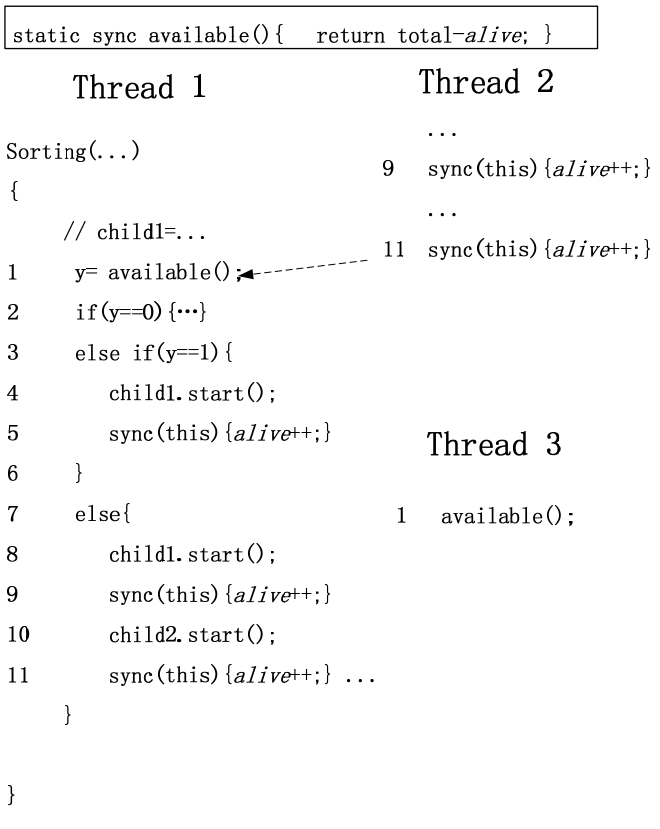

Fig. 7. Relaxation of Paths

A. Race Detection Techniques: Broad Survey

Initial attempts to address the challenge of race detection focused on built-in synchronization primitives [**?**], [**?**], [15], [1]. These include locks as well as the wait/notify and start/join scheduling controls. Notable among these efforts is *lockset* analysis, which considers only locks [**?**]. Because the derived constraints are partial, permitting certain infeasible event reorderings, lockset analysis cannot guarantee soundness [10].

A different tradeoff is struck by the *happens-before (HB)* approach [**?**], [2], [9], which is inspired by and based on Lamport's HB relation [7]. In this style of analysis, all synchronization primitives are accounted for, though reordering constraints are conservative . As an example, HB inhibits reordering of two synchronized blocks governed by the same lock.

Recently there have been successful attempts to relax HB constraints. Among these are hybrid analysis [**?**], which permits both orderings of lock-synchronized blocks, as well as the *Universal Causal Graph (UCG)* representation [6], which also enables both orderings but only if these are consistent with wait/notify- and start/join-induced constraints.

B. Predictive Trace Analysis

Given concurrent execution trace t, a *maximal and sound causal model* based on t defines the set of all traces that a program that is able to generate t can generate (maximality), and only those traces (soundness). 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.

The first to propose PTA are Smagardakis et al. [**?**]. 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 in tact. This ensures that the values of shared memory locations remain the same, which secures the soundness argument, though maximality is not guaranteed.

Said et al. [11] describe a PTA that is also sound albeit not maximal. Similarly to our analysis, Said et al. perform symbolic analysis of the input trace, and then utilize an SMT solver to search for interleaved schedules that establish the presence of data races. Soundsness is guaranteed by their

ability to precisely encode the semantics of sequential consistency. ExceptionNULL [3] is another example of a sound PTA without maximality guarantees, where the goal is to detect null dereferences rather than data races.

Serbanuta et al. [12] have recently shown that it is possible to build a model that is not only sound but also maximal: Any extension of the model with a new trace renders the model unsound. The theoretical framework created by Serbanuta et al. provides a foundation for different forms of PTA, including data races, but also atomicity, serializability and other properties.

Inspired by this result and closer to RECIPE is the PTA technique of Huang et al. [5], which — like our technique — is sound and also maximal, but only under the assumption that local assignments are not modeled. Huang et al.s technique, too, makes use of an SMT solver based on symbolic encoding of the input trace. As part of the encoding, control-flow information is taken into account to enable reorderings that do not violate control dependencies. Still, the two relaxations that RECIPE features, which enable (i) value- rather than dependence-based reasoning about data flow and (ii) consideration of unexplored branches, are strictly outside the scope of Huang et al.'s technique. In Section VI, we demonstrate via direct comparison with Huang et al. the dramatic improvement in coverage thanks to these relaxation methods, which we prove to handle in a sound manner.

IX. CONCLUSION

The conclusion goes here.

ACKNOWLEDGMENT

The authors would like to thank...

REFERENCES

[1] 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 *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 258–269, New York, NY, USA, 2002. ACM.

[2] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles &Amp; Practice of Parallel Programming*, PPOPP '90, pages 1–10, New York, NY, USA, 1990. ACM.

[3] A. Farzan, P. Madhusudan, N. Razavi, and F. Sorrentino. Predicting null-pointer dereferences in concurrent programs. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 47:1–47:11, New York, NY, USA, 2012. ACM.

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

[5] J. Huang, P. O. Meredith, and G. Rosu. Maximal sound predictive race detection with control flow abstraction. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 36, 2014.

[6] V. Kahlon and C. Wang. Universal causality graphs: A precise happens-before model for detecting bugs in concurrent programs. In *Proceedings of the 22Nd International Conference on Computer Aided Verification*, CAV'10, pages 434–449, Berlin, Heidelberg, 2010. Springer-Verlag.

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

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

[9] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 24–33, New York, NY, USA, 1991. ACM.

[10] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 308–319, New York, NY, USA, 2006. ACM.

[11] 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*, NFM'11, pages 313–327, Berlin, Heidelberg, 2011. Springer-Verlag.

[12] T. Serbanuta, F. Chen, and G. Rosu. Maximal causal models for sequentially consistent systems. In *Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers*, pages 136–150, 2012.

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

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

[15] C. von Praun and T. R. Gross. Object race detection. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 70–82, New York, NY, USA, 2001. ACM.