RECIPE: Relaxed Sound Predictive Analysis

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

I. INTRODUCTION

In a multithreaded application, a data race occurs when two concurrent threads access the same memory location, such that (i) at least one of the accesses is write access, and (ii) no explicit mechanism is enforced to prevent the threads from simultaneous access to the location [7]. Race detection is important not only because races often reveal bugs in the implementation, but also as the basis of other techniques and analyses like atomicity checking [5], [6], [8], data-flow analysis [1] and record/replay [2], [4].

In this paper, we focus on *sound* race detection, whereby reported races are guaranteed to be real. This is the key requirement for adoption of the tool by developers [**?**]. Ensuring soundness is a difficult challenge, which mandates dynamic forms of analysis. Indeed, extensive research has been carried out on dynamic race detection. The general goal has been to derive constraints from a given execution trace on event reordering, and check for the remaining reorderings whether they disclose data races. The key question then becomes about constraint extraction.

a) Existing Approaches: Initial attempts to address this challenge focused on built-in synchronization primitives. 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.

A different tradeoff is struck by the *happens-before (HB)* approach. 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, and the *Universal Causal Graph (UCG)* [3] representation, which also enables both orderings but only if these are consistent with wait/notify- and start/join-induced constraints.

Unfortunately, even full consideration of synchronization constructs is insufficient. Branching decisions must also be respected to ensure the feasibility of a reported race. As an example, we refer to the trace in Figure 1. (For now, ignore the statements in red.) This trace is free of any explicit synchronization statements, and so a race between the assignment to variable `x` at line `2` and the use of `x` at line `5` may appear possible. This race is, however, spurious, since the use of `x` at line `5` is governed by positive evaluation of the test at line `4`, which is conditioned on the assignment at line `3`. Hence, line `2` necessarily precedes line `5`.

```
x = 0; y = 0; z = 3;
       z = 2;
```

T_1	T_2
`1: y = 3;`	
`2: x = 1;`	
`3: y = 5;`	
	`4: if (y > z)`
	`5:  print(1/x);`
	`6: else`
	`7:  print(2/x);`

Fig. 1. Example illustrating ordering constraints beyond synchronization primitives

As an alternative, driven by the requirement for soundness, Smagardakis et al. have recently proposed *predictive analysis*. In this approach, 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. As a soundness-preserving relaxation, Huang et al. [**?**] permit reorderings as long as control dependencies are respected.

While being sound and able to find real races, existing predictive analyses are conservative in two respects:

1) Constraints imposed by branch conditions are abided by preserving the exact values flowing into the condition as in the original trace.
2) A second limitation, implied by the first, is that traces that diverge from the original trace in their branching history are strictly out of scope.

Due to these restrictions, enforced as a conservative means of ensuring soundness, existing predictive analysis suffer from limited coverage.

b) Our Approach: The departure point of our approach, yielding strictly better coverage, is to consider the explicit values of shared memory locations instead of dependencies between statements. The gain in coverage is twofold:

1) Reorderings that violate the original dependence structure but preserve the branching history become possible. The values read by a branching statement may change so long as the branch condition evaluates to its original truth value.
2) In addition, in certain cases we can guarantee soundness while directing execution toward branches that diverge from the original trace. This mandates that the effects of

an unexplored branch can be modeled precisely, which holds frequently in practice because the original and new branchees often access the same set of shared memory locations.

Returning to the example in Figure 1, this time also with reference to the statements in red, we demonstrate how value-based reasoning improves coverage. First, unlike predictive analysis, which is unable to detect the race between lines `2` and `5` because of the dependence between lines `3` and `4`, value-based reasoning reveals that trace [`1`, `4`, `5`, `2`, `3`] is feasible, since the first assignment, `y = 3`, satisfies the condition `y > z`. Hence the race is discovered. A second race, between lines `2` and `7`, is detected by negating the condition. This is possible because the effects of the `else` branch can be modeled precisely. Negation leads to an execution starting at line `4` (where `y` $\equiv$ `0`). The race between lines `2` and `7` then becomes visible.

Concretely, we achieve value-level (rather than dependence-level) granularity computationally via a unique encoding of the input execution trace as a constraint system. The constraints are then processed by a satisfiability checker. This gives us the flexibility to explore nontrivial constraints, such as forcing a context switch after the first assignment to `y` or negating the condition in Figure 1. Importantly, these manipulations are beyond what dependence-based reasoning can achieve, since this view of the trace is too conservative.

c) Contributions: This paper makes the following principal contributions:

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 dynamic execution trace of P in Static Single Assignment (SSA) form, such that every variable is defined exactly once.

We begin with an explanation of the encoding process. The resulting formula is provided in Figure 2. We describe the conjuncts comprising the formula one by one.

d) Program Order: The first set of constraints reflects ordering constraints between program statements. We use the symbol O_i to denote the ith program statement, which yields the following formula for Figure 1:

$$O_1 < O_2 < O_3 \wedge O_4 < O_5 \wedge O_4 < O_7$$

That is, the first 3 statements are totally ordered, and the `if` statement executes before the body (either O_5 or O_7).

e) Variable Definitions: The next set of constraints, denoted $\mathtt{z}^i = k$, capture variable definitions: Variable `z` is assigned value k at statement i. For our running example, we obtain:

$$\mathtt{x}^0 = 0 \wedge \mathtt{y}^0 = 0 \wedge \mathtt{y}^1 = 3 \wedge \mathtt{x}^2 = 1 \wedge \mathtt{y}^3 = 5$$

As an example, $\mathtt{y}^3 = 5$ denotes that the value assigned to variable `y` at line `3` is 5.

f) Thread Interference: To express inter-thread flow constraints, we utilize expressions of the form $R^i_{\mathtt{z}} = \mathtt{z}^j$, which denotes that line i reads variable `z`, and the definition it reads comes from line j. The resulting formula for our example is

$$\begin{array}{ll} & (R^4_{\mathtt{y}} = \mathtt{y}^0 \wedge O_4 < O_1) \\ \bigvee & (R^4_{\mathtt{y}} = \mathtt{y}^1 \wedge O_1 < O_4 < O_3) \\ \bigvee & (R^4_{\mathtt{y}} = \mathtt{y}^3 \wedge O_3 < O_4) \end{array}$$

Notice, importantly, that the formula combines flow constraints with order constraints, which are essential to determinize the value read at a given statement. As an example, $(R^4_{\mathtt{y}} = \mathtt{y}^1 \wedge O_1 < O_4 < O_3)$ means that the value of `y` read at line `4` is that set at line `1` assuming an execution order whereby the first statement executes followed by the fourth then third statements.

g) Path Conditions: To preserve path conditions while potentially permitting dependence-violating reordering (e.g., a context switch after line `1` in Figure 1), we model explicitly the condition. For the running example (ignoring the statements in red), this yields:

$$R^4_{\mathtt{y}} > 2$$

That is, the value of variable `y` read at line `4` is greater than 2. Indeed, this constraint is satisfied by the assignments to `y` both at line `1` and at line `3`.

h) Race Condition: The final constraint, forcing the check whether a particular race is feasible, is to demand that two conflicting statements occur at the same time. For the potential race between lines `2` and `5`, we obtain:

$$O_2 = O_5$$

This asserts that both statements occur simultaneously, which — together with the other constraints — guarantees the feasibility of the predicted race if a solution is found for the overall constraint system.

i) 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`.

The conflicting accesses in this case, determined based on static analysis of P, are expressed as $O_2 = O_7$. In addition, we negate the path condition, thereby obtaining $R^4_{\mathtt{y}} \leq 2$ in place of $R^4_{\mathtt{y}} \leq 2$.

j) Constraint Solving: Having conjoined the formuals from the different encoding steps into (i) a global representation of all feasibility constraints (path, ordering, assignment and other constraints) and (ii) the requirement for a given race to occur (expressed as simultaneous execution of the conflicting accesses), we discharge the resulting formula to an off-the-shelf constraint solver, such as Z3 or Yices. If successful, the solver returns a solution for the specified constraints.

In particular, the solution discloses a feasible trace that gives rise to the race at hand. The trace is identified uniquely via the order enforced in the solution over the variables O_i, which represent scheduling order.

$$
\begin{array}{lcll}
 & (O_1 < O_2 < O_3 \wedge O_4 < O_5 \wedge O_4 < O_7) & & \textbf{(program order)} \\
\bigwedge & \left(\mathtt{x}^0 = 0 \wedge \mathtt{y}^0 = 0 \wedge \mathtt{y}^1 = 3 \wedge \mathtt{x}^2 = 1 \wedge \mathtt{y}^3 = 5\right) & & \textbf{(variable definitions)} \\
\bigwedge & \left((R^4_{\mathtt{y}} = \mathtt{y}^0 \wedge O_4 < O_1) \bigvee (R^4_{\mathtt{y}} = \mathtt{y}^1 \wedge O_1 < O_4 < O_3) \bigvee (R^4_{\mathtt{y}} = \mathtt{y}^3 \wedge O_3 < O_4)\right) & & \textbf{(thread interference)} \\
\bigwedge & R^4_{\mathtt{y}} > 2 & & \textbf{(path conditions)} \\
\bigwedge & O_2 = O_5 & & \textbf{(race condition)}
\end{array}
$$

Fig. 2. RECIPE encoding of the trace in Figure 1 as a constraint system

III. TRACE ENCODING

In this section, we describe how a concrete execution trace is encoded as a symbolic trace, which enables RECIPE to derive reordering constraints.

A. Preliminaries

Throughout this paper, we assume a standard operational semantics, which defines (i) a mapping thr between execution threads, each having a unique identifier $i \in \mathbb{N}$, and their respective code, as well as (ii) per-thread stack and shared heap memory.

The code executed by a given thread follows the syntax in Table I. The text of a program is a sequence of zero or more method declarations, as given in the definition of symbol p. Methods accept zero or more arguments $\overline{\mathtt{x}}$, have a body s, and may have a return value (which we leave implicit). For simplicity, we avoid from static typing as well as virtual methods. The body of a method consists of the core grammar for symbol s. We avoid from specifying syntax checking rules, as the grammar is fully standard.

For simplicity, we assume that in the starting state each thread points to a parameter-free method. In this way, we can simply assume an empty starting state (i.e., an empty heap), and eliminate complexities such as user-provided inputs and initialization of arguments with default values. These extensions are of course possible, and in practice we handle these cases, but reflecting them in the formalism would result in needless complications.

As is standard, we assume an interleaved semantics of concurrency. A *transition*, or *event*, is of the form $\sigma \xrightarrow{i/\mathtt{s}} \sigma'$, denoting that thread i took an evaluation step in prestate σ, wherein atomic statement s was executed, which resulted in poststate σ'. We refer to a sequence of transitions from the starting state to either an exceptional state (e.g., due to null dereference) or a state where all the threads have reduced their respective code to ϵ as a *trace*.

We make use of the following helper functions:

- proj t i projects trace t onto all transitions involving thread i.
- $t[k]$ obtains the kth transition within trace t.
- index t τ retrieves the index, or offset, of transition τ within trace t. When simply writing index τ (while omitting the trace parameter) we refer to the index of τ within the original trace.
- pre t τ is the prefix of trace t preceding transition τ. For the suffix beyond τ, we use post t τ. Finally, bet t τ_1 τ_2 returns the transitions delimited by τ_1 and τ_2.

p ::=	$\overline{\texttt{m(x) \{ s \}}}$	**(method)**
s ::=	x = y | x = c |	**(assignment)**
	x = new() |	**(allocation)**
	x = f($\overline{\texttt{y}}$) |	**(invocation)**
	return x |	**(return)**
	z = x { +, −, ×, / } y |	**(aexp)**
	b = x { <, ≤, ≡, ≥, > } y |	**(comparison)**
	b = bx ∧ by | b = bx ∨ by |	**(bexp)**
	y = x.f | x.f = y |	**(heap)**
	if (b) { s } | while (b) { s } | s ; s |	**(control)**
	lock(x) | unlock(x)	**(sync)**

TABLE I
LANGUAGE SYNTAX

B. Basic Encoding: Local Accesses

We describe how RECIPE encodes a trace into symbolic form in two steps. We first start with "local" intraprocedural events — i.e., transitions that manipulate memory accessed by at most a single thread during an invocation-free run — and then extend our encoding scheme to the entire set of possible events.

k) SSA Form: The fundamental encoding transformation is to induce Static Single Assignment (SSA) form on the raw trace, such that a variable is defined exactly once. In this way, def/use chains become explicit, and encoding of trace events as constraints is simplified. As an illustration, trace

```
1: x=1; 2: x<3; 3: x=3;
4: y=1;
5: z=x+y
```

becomes

```
1: x¹=1; 2: x¹<3; 3: x³=3;
4: y⁴=1;
5: z⁵=x³+y⁴
```

(For readability, we version variables according to the line number of their definition.)

l) Local Heap Accesses: Beyond the default SSA rewriting algorithm, we apply a specialized transformation to handle local heap accesses. While in general determining whether a given heap access is local is an undecidable problem, in the context of a concrete execution trace this determination is straightforward. We can thus improve upon the baseline SSA form, where def/use chains over heap accesses are ignored. Indeed, to ensure feasibility, we must account for such accesses. Hence, we replace accesses to field f of local

object o with a fresh local variable $\mathtt{l}_{o.\mathtt{f}}$, where $o.\mathtt{f}$ denotes the memory location itself and not its value. This is done prior to the standard SSA transformation, and under the assumption that in the predicted run, the base object o remains the same.

C. Full Encoding: Method Calls and Shared Accesses

To complete the encoding algorithm, we next explain how shared heap accesses and method calls are dealt with. Similarly to the case of local heap accesses, we take advantage of the fact that the full execution trace is available for analysis to detect shared accesses (the dual of local accesses) as well as call-site resolutions (which is more relevant in practice, where we support virtual method calls).

m) Method Invocations: To account for method invocations, we induce additional context on variable accesses. Instead of recording only the version of a variable, we also represent as part of the variable's identifier its enclosing method and its invocation counter. Thus, symbol $\mathtt{m}^2{:}\,\mathtt{v}^3$ is interpreted as the definition of variable `v` at offset 3 within method `m` during its second invocation.

In this way, the trace is flattened. Method invocations are substituted with qualified variable names, and the rest of the encoding steps, described above, remain unchanged. Because the trace is finite and fully resolved, challenges such as looping, recursion and mutual recursion are all obviated.

n) Shared Heap Accesses: The final aspect of our encoding process is representation of shared heap accesses. As illustrated in Section II, this is done via designated symbols:

- $W^k_{\mathtt{v}}$ denotes write access to variable `v` at trace index k (where `v` is assumed to be a qualified identifier to account for the interprocedural setting).
- Analogously, $R^k_{\mathtt{v}}$ denotes that `v` is read at index k.

The main motivation to encode shared heap accesses using special symbols is to simplify downstream processing, and in particular, the definition of candidate races. We could encode the question of shared versus local accesses as additional constraints (requiring that a variable be accessed by more than one thread), but that would be more complicated and less efficient.

IV. Trace Terminology

Our analysis starts with a trace τ, a sequence of events, $e_0, e_1, \ldots, e_n$. There are types of events in general.

- the shared access, which includes the read and write of the shared fields, e.g., $o.f$=x and x=$o.f$.
- the local access, which includes only the access of the local variable, e.g., $x = y + z$ or $v.f = x$ (where v is a thread-local object).
- the branch event, which evaluates the branch condition to true/false, e.g., $x > 3$.
- the method call event, which includes the method call, e.g., $o.method()$.
- the synchronization event, which includes start/join, wait-/notify, lock/unlock events, e.g., $lock(o)$.

Each event $e_i \in \Sigma$ is a tuple, $< t, id, a, v, type, ins, \_def, \_uses, \_ins >$, where $t \in \mathcal{T}$ denotes the thread generating the event, $id \in \mathcal{ID}$ denotes the unique integer assigned to the event (event id). We say $e_1 < e_2$ iff $e_1.id < e_2.id$. $a \in \mathcal{A}$ denotes the address of the object or field (if any) accessed in the event, $v \in \mathcal{V}$ denotes the value of the definition (if any) in the event, $type$ denotes the type of the event.

Specifically, the address of the object o is denoted as $id(o) \in \mathcal{ID}$, which is a string value representing o uniquely, the address of the static field f is $id(f)$ and the address of the instance object field $o.f$ is $id(o).id(f)$.

$ins \in \mathcal{INS}$ denotes the static three-address instruction corresponding to the event. $\_def \in \mathcal{S}$ is the symbolic representation of the variable definition in the event (We will explain the symbolic representation in Section V). $\_uses$ are the symbolic representations of the variable uses. $\_ins$ is the symbolic form of the instruction after changing the variables into symbolic variables.

The trace supports its standard operations as follows.

- projection, e.g., $\tau|Thread = t$ returns the events from the thread t, $\tau|Address = a$ returns the event involving the address a, $\tau|Type = branch$ returns all branch events.
- concatenation. $\tau' = \tau e$ represents the new trace by appending the event e to τ.
- length $|\tau|$.
- selection. $\tau[i]$ and $\tau[j]$ denotes the ith and jth event in τ. $\tau[i \ldots j]$ denotes the event sequence from the ith event to the jth event. $\tau.before(e)$ and $\tau.after(e)$ are the helper function for selecting the event sequences before or after e. $\tau.between(e_1, e_2)$ selects the event sequence between e_1 and e_2 (inclusively).

In addition, we maintain auxiliary information as follows.

- $R : \mathcal{A} \to \Gamma$ is a function that returns the read accesses of the address $a \in \mathcal{A}$. Each trace τ in Γ is defined over the alphabet of events Σ, specifically, the trace is an empty trace ϵ or defined in this way: $\forall 0 \leq i \leq |\tau|, \tau[i] \in \Sigma, and, \forall i \neq j, \tau[i] \neq \tau[j]$.
- $W : \mathcal{A} \to \Gamma$ is a function that returns the write accesses of the address $a \in \mathcal{A}$.
- $Sync : \mathcal{A} \to \Gamma$ is a function that returns the synchronization events involving the address $a \in \mathcal{A}$.

V. Symbolic Trace

To facilitate the symbolic analysis, we need to introduce symbols to represent the the variables in each event. Symbols allow us to overcome the limitation of concrete dependences and allow us to explore more dependences symbolically. We refer to this process as the symbolicization.

Local Variables Like other symbolic analysis [**?**], [**?**], the symbolic trace should be in the SSA form, i.e., each variable is defined exactly once. This is because the constraint solver employed by the analysis requires each variable to hold only one value. Besides, we need to preserve the local data dependences, i.e., to guarantee each use still reads from the same definition thread-locally.

The simple procedure shows the construction of the symbolic trace for local variables defined or used. The symbols

x = 0; y = 0;

T_1	T_2
1: s=0;	
2: for(i=1;i<3;i++)	
3: s+=i;	
4: **y** = s;	
5: **x** = 1;	
6: **y** = 5;	
	7: if (**y** > 2)
	8: print(**x**+1);
	9: else
	10: print(**x**+2);

Fig. 3. Running Example (shared variables are in bold font).

are constructed by combining the name of the variable in the static instruction $e.ins$ and the runtime event id $e.id$. Lines 6-10 handles the local variable definition. The symbolic representation $e.\_def$ for a variable definition is the combination of the variable name and the event id, where the unique event id guarantees the uniqueness of the symbolic representation. In addition, we record the mapping in $table$. Lines 3-5 build the symbolic representations for the variables used in the instruction, which are computed through looking up $table$ with the variable name as the key. For a local stack variable, its uses and definition involve the same variable name. However, for a local heap variable, i.e., a field, its uses and definition may not involve the same variable name. For example, $o.f = 5$ and $y = r.f$ $(o = f)$ involve the same field but involve different names. To simplify the representation, we introduce a stack variable $l_{address(o.f)}$ to replace the heap variable. The replacement is conducted before the construction of the symbolic trace.

An important assumption for the replacement is that the address of the heap variable will remain the same in the predicted run. We will guarantee the validity of the assumption, as explained in Section.

$ins = e.ins$
for $e : \tau$ **do**
3: **for** $i : 0 \ldots ins.uses.length - 1$ **do**
$e.\_uses[i] \leftarrow table.get(ins.uses[i])$
end for
6: **if** $e.\_def \neq null$ **then**
$e.\_def \leftarrow ins.def^{e.id}$
$table.put(ins.def, e.\_def)$
9: **end if**
end for

Shared Variables Besides, we introduce symbols to represent shared variables read or written. For each shared variable x read (or written), we introduce R_x^{id} (or W_x^{id}) to denote it, where id is the event id. We denote the read and the write with different symbols, which allow us to reason about different dependence relations among them flexibly.

Consider the code in Figure 3, which resembles the example in Figure 1 except that it includes a for loop at lines 1-2.

$Trace$	$Symbolic\ Trace$
0: **x**=0	0: W_x^0=0
1: **y**=0	1: W_y^1=0
2: s=0	2: s^2=0
3: i=1	3: i^3=1
4: i<3	4: i^3<3
5: s=s+i	5: s^5=s^2+i^3
6: i=2	6: i^6=2
7: i<3	7: i^6<3
8: s=s+i	8: s^8=s^5+i^6
9: i=3	9: i^9=3
10: i<3	10: i^9<3
11: **y** = s;	11: W_y^{11} = s^8;
12: **x** = 1;	12: W_x^{12} = 1;
13: **y** = 5;	13: W_y^{13} = 5;
14: **y** > 2	14: R_y^{14} > 2
15: tmp=**x**+1;	15: tmp=R_x^{15}+1;
16: print(tmp);	16: print(tmp);

Fig. 4. Trace

```
x=o.func(y)
func(y1){ // class O1
  i=y1;
  i2=2*i;
  return i2;
}
// func(y2){ // class O2
// ...
```

Fig. 5. Method Calls

Figure 4 shows the instruction $e.ins$ of each event e and its symbolic instruction $e.\_ins$, which is essentially replaces the variables in $e.ins$ to the symbolic variables.

Method Calls To support the method calls, we need to capture the value flow from the actual argument to the formal argument, and the flow from the return variable to the variable definition at the call site. We capture the value flow by recording additional events for each method call. Consider the example in Figure 5, we need to record events to capture the implicit assignment $y1 = y$ and $x = i2$ due to the method call.

Our strategy is as follows. To capture the argument flow $y1 = y$, we record two events, one capturing $ARG0 = y$ and the other capturing $y1 = ARG0$, where $ARG0$ is an artificial variable denoting the first argument of any method. This simple strategy helps handle the virtual method calls. At the call site, we do not know which target method would be called, the first event records information about the actual argument. Then, a target method is invoked, the second event records information about the formal argument of the resolved target method. Combined together, the two events record the argument value flow. The recorded events also undergo the above symbolicization process for gaining the SSA form.

Until now, we get the trace with the symbolic variables.

The following section uses these symbolic representations to compute races. Hereafter, we always refer to the symbolic form, such as symbolic variable or symbolic instruction, unless otherwise specified.

VI. Constraints

First we collect the candidate racy pairs and apply the constraint solver to verify each of them. In general, the candidate racy pairs are collected from the trace as pairs of shared accesses (one is a write) of the same variable from different threads.

$$\{(e_1, e_2 | e_1.a = e_2.a \vee e_1.t \neq e_2.t \vee (e_1 \in W(e_1.a) \wedge e_2 \in W(e_2.a)))\}$$

For each race pair (e_1, e_2), we construct the constraints Φ that encode the feasibility requirement and apply the solver to verify the feasibility. The constraints Φ are the conjunction of the following categories of constraints: local order constraints Φ_{Local}, variable definition constraints Φ_{VDef}, the thread interference constraints $\Phi_{Interference}$, the path condition constraints Φ_{Path} and the race condition constraints Φ_{Race}, and the synchronization constraints Φ_{Sync}. In the following, we discuss each of them in order. We omit the synchronization constraints as they are identical to [**?**].

Race condition constraints The race condition specifies that the racy pair may happen at the same time, i.e., they own the same order. Given two events in the racy pair, (e_1, e_2), suppose $O(e)$ denotes the order variable associated with the event, we have,

$$\Phi_{Race} = (O(e_1) = O(e_2))$$

Path condition constraints The path condition constraints specify that the predicted run should follow the same paths to produce the same set of events (Section VI-B will relax these constraints to explore more paths). It is directly derived from the branch conditions taking the symbolic variables, as illustrated at line 10 in Figure 4. Given a racy pair, (e_1, e_2), suppose e_1 is after e_2 in the trace, i.e., $e_2 < e_1$.

$$\Phi_{Path} = \bigwedge_{e \in (\tau.before(e_1) | Type = branch)} e.\_ins$$

The branches do not need to include those after both racy events as we only need to guarantee the feasibility of the execution leading to the racy events. The branches include not only those from the two threads executing e_1 and e_2 but also those from other threads, as other threads the feasibility may require the interference from other threads.

Variable definition constraints The variable definition constraint captures the value flow due to each statement from the right hand side to the left hand side. It is directly derived from the symbolic form of the instruction in the each event, as illustrated at line 5 in Figure 4. Suppose e_1 is after e_2 in the trace,

$$\Phi_{VDef} = \bigwedge_{e \in (\tau.before(e_1) | (Type = shardAccess \vee Type = localAccess))} e.\_ins$$

Similar to the branches, we include also the accesses from other threads that precede e_1.

Intra-thread order constraints During the predictive analysis, each thread produces the same event sequence in the original run and the predicted run (Section VI-B will relax these constraints to explore more event sequences). The constraints are encoded as follows.

$$\forall i, 0 < i < |\tau'| - 1, where\ \tau' = \tau | t, O(\tau'[i]) < O(\tau'[i+1])$$

Intuitively, the formula specifies that there is an order between any two consecutive events from the same thread.

In our settings, the order variable is denoted with the event id, e.g., the order variable for the event with id 15 is denoted as O_{15} For the example in Figure 4, inside the execution of thread $T1$, we have $O_1 < O_2 < \ldots < O_{13}$. Inside the execution of thread $T2$, we have $O_{14} < O_{15} < O_{16}$.

A. Relaxation for the schedules breaking dependences

In addition to the above constraints, we also need to reason about the dependences between the shared reads and shared writes. Previous predictive analysis requires the predicted run to share many scheduling decisions with the original run so that each shared read reads from the same write (or the same written value) in two runs. Our approach achieves the relaxation, without the need for preserving such scheduling decisions. In our settings, each shared read can read from any write of the same variable from a different thread, or from the preceding write from the same thread, as long as the resultant execution is feasible.

Thread interference constraints Thread interference constraints model the value flows between the writes and reads of the same shared variable. Consider Figure 4, the read R_y^{14} may read from the writes W_y^1, W_y^{11} or W_y^{13}. Each read-write correlation further requires certain scheduling constraints. For example, to read from the write W_y^1, the read should happen after W_y^1, and no other writes of the same variable interleave between them. The constraint can be captured as $R_y^{14} = W_y^1 \wedge O_1 < O_{14} \wedge (O_{14} < O_{11} \vee O_{11} < O_1) \wedge (O_{14} < O_{13} \vee O_{13} < O_1)$.

In general, given an event e_R that contains the read R, and the set S of candidate matching write events, we have the following formula. Here, W_e denotes the shared write access included in the event e.

$$\Phi_{Interference} = \bigvee_{e \in S} (R = W_e) \wedge O(e) < O(e_R) \wedge \bigwedge_{e' \in S \setminus e} (O(e') < O(e) \vee O$$

The candidate matching operations S include both all the writes of the same variable/address from other threads and the preceding write of the same variable from the current thread.

T_1	T_2
1: o_2 = $newClassA()$;	
1: **A** = o_1;	
2: **A** = o_2;	
3: o_2.**f** = 5;	
	4: **r** =**A**;
	5: x=**r**.f;

Fig. 6. Heap Invariant

Heap invariant In the above, we find the matching operations for the predicted run based on the address collected in the original run. However, the matching may be invalid because the address of a field may change if the base reference variable reads a different value, which is allowed in our relaxed analysis.

Consider the following example, in the original run following the line number order, thread T_2 reads o_2 at line 4 and uses the field $o_2.f$ at line 5. Therefore, following the address value, the write at line 3 is a candidate match for the read at line 5. However, in the predicted run, the match may be invalid if the relaxation allows line 4 to read from a different write at line 1.

We need to guarantee that the address collected for each event in the original run remains the same in the predicted run. Therefore, we add the heap invariant constraint Φ_{Heap}. Given an event e that defines the object reference that is used as the base for a field reference or a method call (before the racy pair), we have,

$$\Phi_{Heap} = (e.\_def = e.v)$$

$base(e_{inv}) = baseDefEvent(e_{inv}).v$, requires the symbol representing the base object to read the original value.

In this way, we guarantees three properties: (1) the matching events in the original run are also the matching events in the predicted run, i.e., our above strategy for finding candidate matching events is valid. (2) the shared accesses forming a racy pair in the original run also form a racy pair in the predicted run as the base object references do not change from original run to predicted run. (3) the dispatching of method calls remains the same in both runs as the base object reference remains the same.

Suppose e_1 is after e_2 in the trace, the following algorithm judges whether an event e defines such a base reference. The branch at line 4 returns false because we encounter a redefinition before the use as the base reference.

```
o = e.ins.def
τ' = τ.before(e1)|e.t
3: for e_i ∈ τ'.after(e) do
     if e_i.ins.def = o then
       return false
6:   else
       if e_i.ins references o then
         return true
9:     end if
   end if
end for
12: return false
```

B. Relaxation for the un-explored branches

To reason about the un-explored branches, we conduct the symbolic execution to collect the symbolic trace for the unexplored branches. At the high level, the symbolic execution starts from any branch event e in the original trace τ, and symbolically executes the un-executed branch. The un-explored branch may contains branches itself, therefore, we conduct the depth-first search (DFS) to collect the symbolic traces Γ_e for all paths inside the branch. We truncate the original trace at the branch event, append it with the negation of the branch event and the symbolic trace derived during the DFS. The new trace is $\tau.before(e)neg(e)\tau_e$, where τ_e is any symbolic trace from Γ_e. We apply the constraint solver to the new trace to find potential races.

Loop issues and object creation issues are known limitations of symbolic execution. For simplicity, we assume the un-explored branch is simple enough, i.e., it does not contain the loops and every object reference is fully resolved at the branch point. In case that the assumption does not hold, we avoid exploring such a branch, sacrificing the coverage in general.

Besides, as mentioned above, we need to know the address which guides the matching of reads and writes. As we assume the unexplored branch has no object creation, all addresses are fully resolved.

VII. Implementation

A. Time Window and Initial Value constraints

The trace is typically long and the constraint solver cannot scale to the whole trace. 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 ∈ A do
  τ_a = W(a)
3:  sstore.put(a, τ_a[|τ_a| − 1].v)
end for
for e ∈ τ 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.

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

.

Object Creation Given the event in the form of $x = newObject()$, we encode the created object as a unique integer and treats the instruction as a normal integer assignment. The unique integer is the address of the object, calculated by $System.identityHashcode(x)$.

Array access For array access event $arr[i] = x$, we treat them as the field access event $arr.i = x$, where the index i is treated as a field.

Shared Analysis We distinguish the accesses of the shared fields and local fields. To collect the shared fields, we preprocess the trace and map each field to a set of accessing threads. If a field is written by two or more threads, it is shared. If a field is written by one thread and read by other threads, it is shared. Otherwise, it is a local field.

Optimization . Before the symbolic analysis, we conduct the backward slicing by treating the racy pair as the seeds. Only those selected in the slices are preserved while the rest events are removed.

VIII. Evaluation

Evaluation Method We conduct our experiment with a set of large applications, e.g., those from the Dacapo benchmark suite. The benchmarks are shown in Table. We compare our approach with the state-of-the-art technique proposed by Huang et al. As the capability of finding races depends on the monitored run, for fair comparison, we apply both techniques to the same monitored run. Specifically, we use two threads in the monitored run as two threads are sufficient to manifest most of races [**?**].

The Dacapo relies on the reflection functionality to execute, which imposes challenges to static analysis and transformation. We apply the tool Tamiflex [] to overcome the problems, which resolves the reflection using the classes observed in the dynamic run. The window size is 10000.

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 [1].

A. Scalability Issues

Our approach suffers from scalability issues for two reasons. First, as we record local access events in the monitoring run, the monitoring run takes long time, e.g., more than 2 hours, and our trace is typically large, e.g., at the GB level. To avoid the out of memory issue during the monitoring run, we store the event to the database on the fly, rather than storing them in a buffer. Besides, we terminate the monitoring run with a shutdown hook if it is beyond 30 minutes. Second, the prediction phase needs to load the trace to memory, which easily causes out of memory problem. To solve the problem, we separate the analysis into several runs, each run restarting the JVM and resuming from the window of last run. Different runs communicate the window id using the external file. These runs are organized together automatically using the ant. The outputs of the files are merged into one summary finally. Besides,

TABLE II
HAHA

ad	fda
1	2

IX. Case studies

We manually inspect the reported races in small applications to gain better understanding about our approach.

Relaxing the Inter-thread dependence Figure 7 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. [] detects the race between line 291 and line 400, but fails to detect the race between line 294 and line 400. The reason is as follows. In the observation run, the execution follows the order, lines 400, 291 and 294. For the event at line 294, its preceding branch at line 291 reads from line 400. Therefore, Huang et al. [] requires the predicted run to preserve the dependence between line 291 and line 400 so that line 291 reads exactly the same value and the branch takes the same branch decision. 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 forms a race. We re-replay such race easily using the eclipse IDE breakpoints.

[1]Note that Sun JDK 1.7 does not support the transformation of dacapo applications with tamiflex.

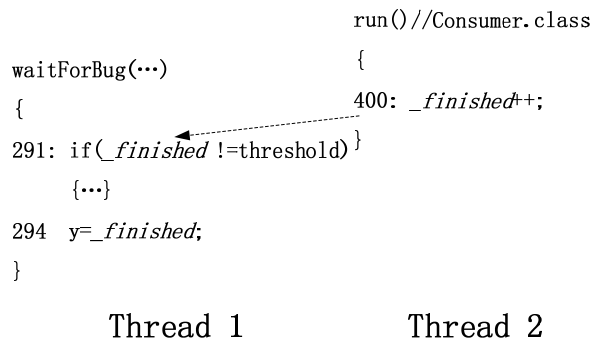

Fig. 7. Relaxation of Inter-thread dependence

X. Validity Threat

Java method can have 65535 bytecode instructions maximally. Therefore, we count the number of bytecode instructions inside each method, if the number exceeds 65525, we avoid instrumenting the method. The consequence is that, we will miss the races inside the method. 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.

We do not support the boolean operations such as &, bit operators $<<$, which contributes to most of our misses.

XI. Conclusion

The conclusion goes here.

Acknowledgment

The authors would like to thank...

References

[1] R. Chugh, J. W. Voung, R. Jhala, and S. Lerner. Dataflow analysis for concurrent programs using datarace detection. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 316–326, New York, NY, USA, 2008. ACM.

[2] J. Huang, P. Liu, and C. Zhang. Leap: Lightweight deterministic multi-processor replay of concurrent java programs. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 207–216, New York, NY, USA, 2010. ACM.

[3] 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.

[4] D. Lee, P. M. Chen, J. Flinn, and S. Narayanasamy. Chimera: Hybrid program analysis for determinism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 463–474, New York, NY, USA, 2012. ACM.

[5] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 329–339, New York, NY, USA, 2008. ACM.

[6] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: Detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 37–48, New York, NY, USA, 2006. ACM.

[7] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 27–37, New York, NY, USA, 1997. ACM.

[8] Y. Shi, S. Park, Z. Yin, S. Lu, Y. Zhou, W. Chen, and W. Zheng. Do i use the wrong definition?: Defuse: Definition-use invariants for detecting concurrency and sequential bugs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 160–174, New York, NY, USA, 2010. ACM.