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.

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

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 ::=		
	m(x̄) { s }	**(mdcl)**
s ::=		
	x = y | x = c |	**(asgn)**
	x = new() |	**(alloc)**
	x = f(ȳ) |	**(call)**
	return x |	**(return)**
	z = x { +,−,×,/ } y |	**(aexp)**
	b = x { <,≤,≡,≥,> } y |	
	b = bx ∧ by | b = bx ∨ by |	**(bexp)**
	y = x.f | x.f = y |	**(heap)**
	if (b) { s } | while (b) { s } | s ; s |	**(ctrl)**
	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 $\mathtt{m}^2{:}\,\mathtt{v}^7$ is interpreted as the definition of local variable `v` within the second invocation of method `m` that occurs at trace index 7.

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. Constraint Derivation and Resolution

We now describe in detail the constraint encoding and solving steps.

A. Same Trace

We begin with the basic setting, which enforces strict reordering of the events along the input trace without diverging into new execution paths. We relax this requirement later, such that different branches can be followed as long as feasibility is retained.

o) Intra-thread Order: The first set of constraints reflects control flow within the individual threads, which must remain unchanged under the reordering transformation. Given input trace t, this is expressed as the following formula:

$$\begin{aligned}\forall \tau, \tau' \in t.\quad & \mathsf{thread}\ \tau \equiv \mathsf{thread}\ \tau'.\\ & \mathsf{index}\ \tau < \mathsf{index}\ \tau' \Rightarrow O_\tau < O_{\tau'}\end{aligned}$$

The logical variables O_x express ordering constraints. The requirement, as stated above, is that these variables reflect the same order as the projection of index onto individual threads.

p) Race Condition: Given pair τ and τ' of events that both access a common memory location ℓ, we demand that

$$\begin{aligned}& \mathsf{thread}\ \tau \neq \mathsf{thread}\ \tau'\\ \bigwedge\ & (\mathsf{writes}\ \tau\ \ell \vee \tau'\ \mathtt{s}'\ \ell)\\ \bigwedge\ & O_\tau = O_{\tau'}\end{aligned}$$

That is, (i) events τ and τ' are executed by different threads, (ii) at least one of the events performs write access to ℓ, and (iii) the events occur at the simultanesouly. This is a direct logical encoding of the definition of a race condition.

q) Path Constraints: The requirement with respect to branching is that the prefix of the original trace t up to the pair τ and τ' of candidate racing events remains identical in the predicted trace t'. More accurately, under the assumption that $\mathsf{index}\ \tau < \mathsf{index}\ \tau'$, we require that

$$\bigwedge_{\tau'' \in t \cap \mathbf{bexp}.\ \tau'' \in \mathsf{pre}\ \tau'} [\![\mathsf{stmt}\ \tau'']\!]\ t \equiv [\![\mathsf{stmt}\ \tau'']\!]\ t'$$

where stmt is a helper function that obtains the code statement incident in a given transition. That is, all branching transitions up to τ' (which occurs after τ) preserve their boolean interpretation under t'. This ensures that there are no divergences from the path containing the racing events, though beyond that path any feasible continuation is permitted. Importantly, contrary to [**?**], we do not pose the requirement that the values flowing into branching statements remain the same, but suffice with the relaxed requirement that the evaluation of branching expressions is invariant under the input and predicted traces.

r) Variable Definitions: The final requirement for the basic setting is that left-hand variables are defined according to the same right-hand variables as before. That is, version i of variable `u` is defined as version j of variable `v` in input trace t, then the same remains true in predicted trace t'. This is enforced as the formula

$$\bigwedge_{\tau'' \in t \cap \mathbf{asgn}.\ \tau'' \in \mathsf{pre}\ \tau'} \mathsf{stmt}\ \tau'' \in t'$$

where we again assume that $\mathsf{index}\ \tau < \mathsf{index}\ \tau'$. That is, the same statement occurring in t is also present in t' (though the transitions may differ). Since the statements of t' are a permutation of the statements of t, we are assured that use/def flow is constrained appropriately.

B. Unexplored Paths

V. 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 V-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 **??**. 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.

One important requirement is that we need to preserve the same branch decisions for all preceding branches, rather than those of the guarding branches. Consider the following code, where sx and sy are shared variables with the initial values 0 and tmp is a local variable with initial value 0, there are two branches, one is the guarding branch for update of the shared variable at line 4, the other is not. Suppose in the observation run, the first branch takes the false branch (hence, $tmp = 5$ is not included in the trace), and the second takes the true branch. If the solver only guarantees the same branch decision for the guarding branch, i.e., the second, it risks reporting the false feasibility of line 4. Specifically, line 4 is infeasible if the first branch is after some remote update that forces it to take the true branch. However, in this case, the solver still judges line 4 as feasible as it does not model the true branch of the first branch, which is not in the trace. Therefore, we need to respect all branch decisions so that the set of events under reasoning are all captured in the trace.

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,

```
                    // sx=4;
if(sx>3)
   tmp=5;
if(tmp≠ 5 )
sy=0;
```

Fig. 3. Method Calls

as illustrated at line 5 in Figure **??**. 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 V-B will relax these constraints to explore more event sequences). The constraints are encoded as follows.

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

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 **??**, 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 **??**, 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

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. 4. Heap Invariant

following formula. Here, W_e denotes the shared write access included in the event e.

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

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.

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(e_1)|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.

VI. 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. We ensure the predicted run sees the same index at the same event by specifying the additional constraint $i == v$, where v is the concrete index value observed.

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.

B. 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,

VII. Evaluation

Our evaluation focuses on the effectiveness and scalability of our approach. To measure the effectiveness, we compare with the predictive analysis, RV []. We choose 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.

Evaluation Method We conduct our experiment on a large set of applications, which are also used to evaluate RV. Specifically, the set includes large applications such as Jigsaw, Xalan, Lusearch. For the large applications that use the reflection, we adopt Tamiflex [] to to support the static analysis, which replaces the reflection calls with the concrete method calls recorded in the observation run. We omit the benchmark eclipse because the current version of Tamiflex leads to abnormal execution after the instrumentation, which throws exception during starting the main service. By applying our tool to such abnormal execution, we identify only 3 races, similar to the report of RV []. Besides, the benchmark montecarlo requires the input, which we downloaded from internet and simplified. For the large applications, we use the most lightweight configuration if possible.

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 set of races, (2) the original implementation of RV contains a bug in identification of the branches, which leads to the misses of many branches and the incorrect reduction of dependences during the analysis. We confirmed the bug with the author and fixed the bug in our experiments [1].

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 [2].

A. Effectiveness

Table II shows the main results of our analysis, which includes four sections: Trace (details about the trace), Races

[1]We report the bug in details with the test case https://sites.google.com/site/recipe3141/

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

(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, for the branch, we report it in the form of 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 benchmarks, which have relatively more complex logic. The reads of the shared base objects occupy a small portion ($1/3$-$1/10$) of the total shared reads. The rest shared reads read only the primitive values or the references that are not the base references, e.g., the references involved only in the nullness check. Besides, our analysis involves significantly less branch events as compared to the RV approach. The 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 fully-fledged version, and RV. By comparing the Recipe-s and Recipe, we find 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 the Recipe version with RV version, as illustrated in the $Difference$ section, where $Diff$ shows the races found by Recipe but missed by RV, $Diff'$ shows the races found by RV but missed by Recipe.

First of all, Recipe finds many 150+ more races than RV. The reasons are multifold: (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 allows the relaxation of the scheduling even if it breaks the inter-thread read/write dependence in the observed run. Recipe allows the majority of the shared reads, i.e., the reads of primitive values, to freely read from 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 conduct case studies (Section **??**) to better illustrate the scenarios.

Another interesting observation is, 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 underlying reason is due to the limits of the constraint solver: (1) the solver cannot compute constraints with very complex arithmatic operations (2) the solver does not support some program constants such as the scientific notation, $3E-10$.

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 the fine-grained reasoning of value flows. As a result, it detects many more races compared to existing approaches. On the other hand, it is heavy weighted. The suggestion is to combine it with the lightweight approach with soundness guarantee such as RV, by treating RV as the preprocessing and instructing Recipe to skip those confirmed by RV. In this way, we can complement Recipe by finding those missed by it due to the limit of the constraint solver.

B. Case studies

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

Relaxing the Inter-thread dependence Figure 5 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

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

the two forms a race. We re-replay such race easily using the eclipse IDE breakpoints.

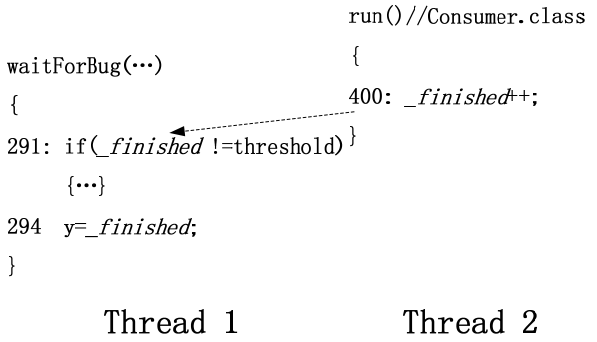

Fig. 5. Relaxation of Inter-thread dependence

Relaxing the Paths Figure 6 demonstrates how our approach relaxes the schedulings to account for the unexecuted paths. Each thread invokes the method Sorting, which recursively starts two children threads if there are two more available entries in the thread pool (lines 8-11), or starts one child thread if there is only one available entry (lines 4-5). The availability is computed through the static method available with the constant `total`, which is equal to 5. The shared variable `alive` denotes the used entries.

Initially there is one sorting thread. After it starts Thread 1 and Thread 2, there are three threads alive and only two more entries are available. Suppose in the observation run, Thread 2 consumes both thread entries and starts the children 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 the 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 have such 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 [3].

VIII. 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.

IX. 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.

[3] We abbreviate the `synchronized` keyword as `sync`.

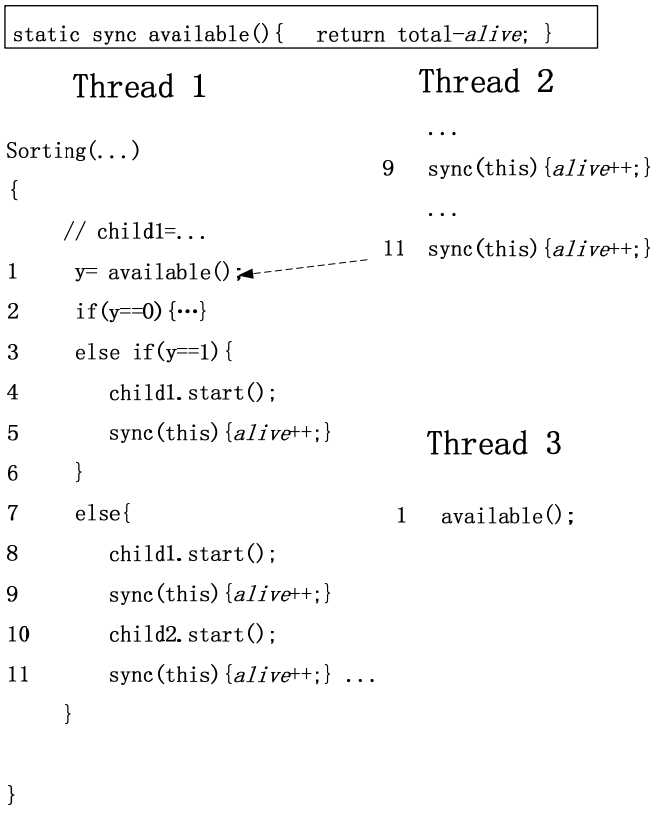

Fig. 6. Relaxation of Paths

[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.