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 preceds line `5`.

```
          x = 0; y = 0; z = 3;
                 z = 2;
----------------------------------------
        T1                  T2
----------------------------------------
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 ≡ 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:

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:

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

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:

$$\mathrm{x}^0 = 0 \bigwedge \mathrm{y}^0 = 0 \bigwedge \mathrm{y}^1 = 3 \bigwedge \mathrm{x}^2 = 1 \bigwedge \mathrm{y}^3 = 5$$

As an example, $\mathrm{y}^3 = 5$ 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

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

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, $(R^4_{\mathrm{y}} = \mathrm{y}^1 \wedge O_1 < O_4 < O_3)$ 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:

$$(R^4_{\mathrm{y}} > 2$$

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 $R^4_{\mathrm{y}} \leq 2$ in place of $R^4_{\mathrm{y}} \leq 2$.

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. If successful, the solver returns a solution for the specified constraints, and in particular, we obtain a feasible trace that gives rise to the race at hand.

III. Basics

A. Trace Terminology

Our analysis starts with a trace τ, a sequence of events, $e_0, e_1, \dots, e_n$. There are three 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 synchronization event, which includes start/join, wait-/notify, lock/unlock events, e.g., $lock(o)$

Each event $e_i \in \Sigma$ is a tuple, $< t, id, a, v, ins >$, 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), $a\mathcal{A}$ 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, and $ins \in \mathcal{INS}$ denotes the three-address instruction generating 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 $id(o)\_id(f)$. Besides, as the event is derived by instrumenting the three-address code and monitoring the instrumented execution. Therefore, each event can involve at most three operands.

The trace supports its standard operations as follows.

- projection, e.g., $\tau|t$ returns [1] the events from the thread t, $\tau|a$ returns the event involving the address a.
- concatenation. $\tau' = \tau e$ represents the new trace by appending the event e to τ.
- length $|\tau|$.
- selecting an element. $\tau[0]$ and $\tau[|\tau| - 1]$ represents the first and last event in τ.

In addition, we maintain auxiliary information as follows.

- $AT : \mathcal{A} \times \mathcal{T} \to \gamma$ is a function that returns a trace $\tau \in \gamma$ that contains only the accesses of the address $a \in \mathcal{A}$ by the thread $t \in \mathcal{T}$. 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: $\forall 0 \leq i \leq |\tau|, \tau[i] \in \Sigma, and, \forall i \neq j, \tau[i] \neq \tau[j]$.
- $R : \mathcal{A} \to \gamma$ is a function that returns the read accesses of the address $a \in \mathcal{A}$.
- $W : \mathcal{A} \to \gamma$ 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 $a \in \mathcal{A}$.

B. Symbolic Trace

To facilitate the symbolic analysis, we need to introduce symbols to represent the operands in each event. Symbols allow us to overcome the limitation of concrete dependences and allow us to explore more dependences symbolically.

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 make sure each use still reads from the same definition thread-locally.

```
                x = 0; y = 0;
         T1                        T2
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. 2. Running Example (shared variables are in bold font).

The simple procedure shows the construction of the symbolic trace for local variables defined or used. The symbols are constructed by combining the static instruction and the runtime event id. Lines 6-10 handles the local variable definition. We build a symbolic variable s for it by combining the variable name and the event id. The uniqueness of the event id guarantees that each symbolic variable is defined exactly once. In addition, we replace the variable to the symbolic variable in the instruction and record the replacement in $table$. Lines 3-5 updates the local variable used in each event so that it is replaced with the symbolic variable for the corresponding definition. Here, the corresponding definition and the use share the same variable name, therefore, we can easily find out the symbolic variable through looking up the $table$.

The SSA form of the trace is different from the SSA form of the instruction as the SSA instruction can only distinguish definitions at different program points but cannot distinguish the definitions at different execution points that share the same program point.

[3] $e : \tau$ $ins \leftarrow e.ins$ $ins.use : ins.uses$ $ins.use \leftarrow table(ins.use)$ $ins.def \neq null$ $s \leftarrow ins.def^{e.id}$ $table[ins.def \to s]$ $ins.def \leftarrow s$

Shared Accesses Besides, we introduce symbols to represent shared reads and shared writes which leave the inter-thread dependence between reads/writes undetermined. For each read (or write) of shared variable x, we introduce R_x^{id} (or W_x^{id}) to denote it, where id is the event id. Consider the code in Figure 2, which resembles the example in Figure 1 except that it includes a for loop at lines 1-2. The symbolic trace is produced in Figure 3. For simplicity, we only show the symbolic variables, while omitting other information such as thread information.

Method Calls The key to supporting the method calls is to capture the value flow the actual argument to the formal argument, and the flow from the return statement to the LHS variable of the method call. To explicitly model the value flow, record two additional events for each method call. Consider the

[1]This is the abbreviation for the complete form $\tau|Thread = t$

$Trace$	$Symbolic\ Trace$
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. 3. Trace

example in Figure 4, we record the local access event $y1 = y$; for the argument value flow and record the local access event $x = i2$;. Recording the additional events is achieved through instrumenting the call site and the callee method statically.

The above simple strategy however hides the complexity of the virtual method calls. At a call site of a virtual method, the static instrumentation cannot know precisely which method would be called. Therefore, we do not know what formal argument the actual argument flows to. Consider the example in Figure 4, suppose another implementation of the virtual method exists (in the comments). We do not know how to instrument the code statically, $y1 = y$ or $y2 = y$.

To avoid the problem, we have to combine the runtime knowledge. Our strategy is as follows: rather than directly record direct value flow from actual argument to formal argument, we introduce an artificial variable during the static instrumentation. Then we insert the instrumentation $record(ARG0 = y;)$ at call site, and insert $record(y1 = ARG0)$ at the entry of the method $func$ declared in the first class, and insert $record(y2 = ARG0)$; at the entry of the method $func$ declared in the second class. At runtime, depending on which $func$ method is invoked, we record either the event sequence $ARG0 = y; y1 = ARG0$ or the sequence $ARG0 = y; y2 = ARG0$, which precisely captures the value flow. We model the return value flow similarly.

Note that although different methods use the same names for the artificial variable, they are translated to different variables after we get the SSA form of symbolic trace.

C. Constraints

The constraints fall into the following categories:

IV. Time Window

```
x=o.func(y)
func(y1){ // class O1
  i=y1;
  i2=2*i;
  return i2;
}
// func(y2){ // class O2
//   j=y2;
//   j2=3*j;
//   return j;
// }
```

Fig. 4. Method Calls

V. Heap Invariant

VI. 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.