Recipe: Relaxed Sound Predictive Analysis

Abstract—**The abstract goes here.**

I. Introduction

A race occurs when two concurrent threads access the same variable and when (i) at least one access is a write, (ii) the threads take no explicit mechanism to prevent the accesses from being simultaneous [7]. More intuitively, a race occurs when two accesses of the same variable (one is write) can happen in parallel. Race detection is a fundamental analysis for concurrent programs, which underlies many client applications including the atomicity/order violation detection [5], [6], [8], the dataflow analysis [1] and the record and replay [2], [4].

Over the last decade, researchers have spent tremendous efforts on race detection and have gained relative success. The initial efforts center around the *synchronization primitives* including the locks and the scheduling control such as wait/notify and start/join. Lockset analysis, which considers only locks, produces many infeasible races that contradict the schedule control. Happens-before analysis reasons about all the primitives but imposes overly restrictive lock order constraints, i.e., two lock regions with the same lock should follow the execution order as observed. The following work further relaxes the lock constraints: Hybrid analysis allows the two lock regions to be ordered in either way; Universal Causal Graph (UCG) [3] allows them to be ordered in either way only when this relaxation does not contradict the schedule control.

However, these seminal work misses the important *value constraints*, leading to false positives. Consider the example in Figure 1, where `x` and `y` are shared variables and no synchronizations are employed. The work that considers only synchronization constraints would report a race between line 2 and line 5 on x, which appear in the observed run. However, the race is a false alarm because line 5 can be executed only after line 4 reads the value of `y` from line 3 (after line 2).

The false positives greatly reduce the usability of the detection tool as programmers need to spend great efforts in investigating each of them. Recently, the predictive analysis attracts increasing research attention for its soundness guarantee, i.e., no false alarms. Smagardakis et al. pioneered by proposing first of such work. Starting from an original trace, their sound predictive analysis "predicts" a new trace by relaxing the execution order while preserving the inter-thread dependence constraints and the synchronization constraints. The inter-thread dependence constraints require the predicted trace and the original trace to exhibit the same set of dependences between conflicting accesses. Preservation of them leads to the soundness as every value remains the same. Huang et al. [] further optimizes by preserving only the inter-thread

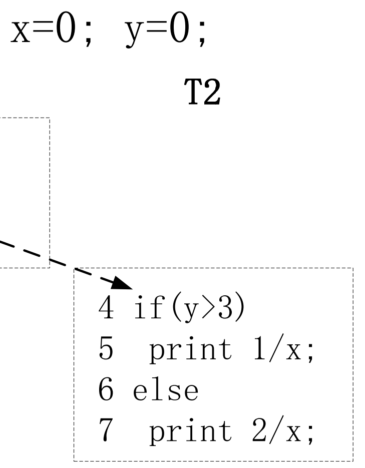

Fig. 1. Running Example

dependences that could affect the branch decisions in the following executions.

Although showing promises, existing sound predictive analyses are conservative as they satisfy the value constraints imposed by the branch conditions by preserving the inter-thread dependences and reading the same values at the branches. In this work, we propose the relaxed sound predictive analysis, which relaxes the value constraints to find more races without sacrificing the soundness. Our relaxation follows two key insights. First, we satisfy the value constraints computationally, rather than recording and preserving the dependences. The values read at the branches are allowed to change in the prediction as long as the change does not affect the branch decisions. We realize the value constraints through the satisfiability checker. Second, even if the change of the read values affects the branch decisions, we can still explore the un-executed branch if it is "simple" enough. The underlying rationale is, when the executed branch contains the shared accesses, the sibling (un-executed) branch is likely to access the same shared variable. Reasoning about the un-executed branch is enabled by flexibility of our computation-based approach, i.e., we just need to reverse the branch condition and resend it to the checker, while existing approaches cannot support such reasoning due to the lack of the computation power and the conservativeness of perservation-based approaches.

We highlight our approach with the example [1] in Figure 1. Suppose the branch at line 4 changes to `if(y>=3)`. In the original run, line 4 reads from the value defined at line 3. Therefore, Huang et al.'s approach preserves the same

[1]We omit the synchronization primitives purposely for simplification.

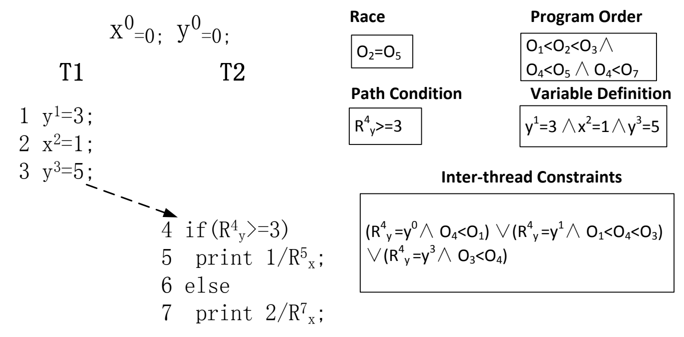

Fig. 2. Overview

dependence in the predicted run, and therefore misses the race on x between line 2 and line 5. In contrast, our approach reports two more real races on x, (2,5) and (2,7).

We first demonstrate how we find the race (2,5). As shown in Figure 2, we first encode trace in the SSA form, where each variable is defined exactly once (line number as the superscript). Besides, we introduce a new symbol, e.g., R^4_y, to denote each shared read, e.g., the read of y at line 4. We also list the constraints for a valid predicted run. Race constraints specify line 2 and line 5 can run in parallel. Here O_2 and O_5 are variables introduced to denote the orders. Program order constraints specify the intra-thread execution order. The rest constraints model the value constraints: path condition constraints specify the branch decision for the branches preceding the potential racy event line 5; variable definition constraints specify the intra-thread value flow due to each assignment; Inter-thread constraints specify the inter-thread value flow which is conditioned on the scheduling constraints. For example, $R^4_y = y^1 \wedge O_1 < O_4 < O_3$ specifies that line 4 reads y defined at line 1 if and only if line 4 happens after line 1 and before line 3. Combined together, the constraints are sent to off-the-shelf solver such as z3 or yices, which then returns a solution: line 4 reads y (y=3) defined at line 1, which satisfies the branch condition.

Finding the race (2,7) requires a slightly different set of constraints. Specifically, the race constraint becomes $O_2 = O_7$, and the path condition constraint becomes $R^4_y < 3$. The checker also returns a solution for these constraints: line 4 reads y (y=0) from the initial value, which steers the execution towards the unexecuted path at the branch line 4.

a) Contributions: The principal contributions of this paper are the following:

II. Technique

A. Basics

Our analysis starts with a trace τ, a sequence of events, $e_0, e_1, \ldots, e_n$. 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 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., $lock(o)$

The event is derived by instrumenting the three-address code and monitoring the instrumented execution. Therefore, each event can involve at most three operands.

Each event $e_i \in \Sigma$ contains the thread information $T_{e_i} \in \mathcal{T}$, the unique index of the event in the trace I_{e_i}, the address of the object or object field (if any) accessed in the event $A_{e_i} \in \mathcal{A}$, and the value of the definition (if any) inside the event $V_{e_i} \in \mathcal{V}$. 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 $id(o)\_id(f)$.

The trace supports its standard operations as follows.

- projection, e.g., $\tau|t$ returns [2] 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 $|\tau|$.
- selector element. $\tau[0]$ and $\tau[|\tau| - 1]$ represents the first and last event in τ.

In addition, we maintain auxiliary information as follows.

- $Acc : \mathcal{A} \times \mathcal{T} \rightarrow \gamma$ 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: $\forall 0 \leq i \leq |\tau|, \tau[i] \in \Sigma, and \forall i \neq j, \tau[i] \neq \tau[j]$.

1) Method Calls:

B. Time Window

C. Heap Invariant

III. 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.

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

[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.