Persuasive Prediction of Concurrency Access Anomalies

Jeff Huang, Charles Zhang
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
{smhuang, charlesz}@cse.ust.hk

ABSTRACT

Predictive analysis is a powerful technique that exposes concurrency bugs in un-exercised program executions. However, current predictive analysis approaches lack the *persuasiveness* property as they offer little assistance in helping programmers fully understand the execution history that triggers the predicted bugs. We present a persuasive bug prediction technique as well as a prototype tool, PECAN, for detecting general access anomalies (AAs) in concurrent programs. The main characteristic of PECAN is that, in addition to predict AAs in a more general way, it generates "bug hatching clips" that deterministically instruct the input program to exercise the predicted AAs. The key ingredient of PECAN is an efficient offline schedule generation algorithm, with proof of the soundness, that guarantees to generate a feasible schedule for every real AA in programs that use locks in a nested way. We evaluate PECAN using twenty-two multi-threaded subjects including six large concurrent systems, and our experiments demonstrate that PECAN is able to effectively predict and deterministically expose real AAs. Several serious and previously unknown bugs in large open source concurrent systems were also revealed in our experiments.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids; Tracing; Diagnostics*

General Terms

Algorithms, Performance, Reliability

Keywords

Persuasive, Access Anomaly, Bug Detection

1. INTRODUCTION

Concurrent programs are plagued by a class of bugs called access anomalies (AAs), characterized by criteria such as data race [36], atomicity violation [10], and atomic-set serializability violations (ASV) [42]. Among the broad spectrum of concurrency bug detection techniques that have proliferated in recent years [40, 19, 26, 39, 2, 22, 1, 28, 27, 25, 11, 13, 33, 37, 32, 16], the technique of predictive trace analysis (PTA) has drawn the significant research attention [43, 46, 45, 9, 5, 47].

Generally speaking, a PTA technique records a trace of execution events, statically (often exhaustively) generates other permutations of these events under certain scheduling constraints, and exposes concurrency bugs unseen in the recorded execution. PTA is a powerful technique as, compared to dynamic analysis, it is capable of exposing bugs in unexercised executions and, compared to static analysis, it incurs much fewer false positives for the fact that its static analysis phase uses the concrete execution history.

In our mind, a bug detection technique is more useful if it is *persuasive*. This new criterion emphasizes that a bug detection technique should not only localize the bug in the source code but also, and more importantly, help programmers in fully understanding how the bug has occurred, to provide good fixes.[1] We characterize persuasiveness by two key properties. First, a persuasive technique should report violations with *no false positives*. Since it is non-trivial to manually verify the false alarms in large sophisticated concurrent systems, the perceived usefulness of the technique quickly deteriorates with even a small number of false positives. Second, a persuasive technique should also show programmers how the detected bugs or violations can occur by accompanying each violation with a "*bug hatching clip*", which essentially instructs the program to deterministically execute the bug hatching process step by step. We believe that allowing programmers to deterministically trigger the bug is one of the most effective ways to achieve the complete bug comprehension.

Assessed by the persuasiveness criterion, the state of the art PTA techniques [43, 46, 45, 9, 5, 47] are unsatisfactory in generally addressing access anomalies in real-life complex concurrent programs. Although several recent work [46, 45, 43] has also pointed out the usefulness of persuasiveness, it is still not clear how to efficiently create a concrete execution that can expose the predicted anomalies in real programs. In addition, despite the much improved soundness compared to static analysis, current PTA techniques still report quite a number of false positives, either due to the inadequacy of their prediction models or the incompleteness of the col-

[1] A report [14] shows that as much as 9% bug fixes are bad fix, either failing to fix a bug or creating new bugs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSTA '11, July 1721, 2011, Toronto, ON, Canada
Copyright 2011 ACM 978-1-4503-0562-4/11/07 ...$5.00.

lected traces. For example, as detecting data races in general is NP-hard [30], for efficiency reasons, many race detectors [31, 35, 6, 37] employ an overly approximated prediction model that combines the lockset-based algorithms [36] and the happens-before based approaches [23]. Moreover, for PTA techniques, a certain type of false positives simply cannot be avoided when programmers use application level synchronization mechanisms, such as barrier and flag operations. These "non-standard" synchronization mechanisms are difficult to be automatically discovered [41] and, in turn, result in incomplete traces.

In this paper, we present PECAN, a novel persuasive PTA technique that detects general access anomalies (AAs) in concurrent programs. Unlike other PTA techniques [5, 40, 9] that cater specific types of concurrency bugs, PECAN offers a general prediction model that addresses a much broader class of concurrent access anomalies. Moreover, for each predicted AA, PECAN generates "bug hatching clips" that deterministically instruct the input program to exercise the predicted AAs. PECAN does not present false positives to programmers as we guarantee that each clip represents a feasible concrete execution. Since all AAs reported are real and the programmers are given the full history and context information to understand the bug, we believe PECAN can dramatically expedite the process of bug fix.

The key technical challenge that we are faced with is how to statically generate a *feasible thread execution schedule* to expose the predicted AAs. We present an algorithm, with the proof of its soundness, that guarantees to generate a feasible schedule for every real AA for programs that use locks in a nested way, i.e., releasing locks reverse to the acquisition order. Moreover, to predict AAs in a general way, we present a general specification model of AAs and reduce the AA prediction problem to a graph pattern search problem. With compact encoding of the happens-before relationship between the events and the scheduling order of memory accesses in the trace, the graph supports efficient pattern search of AAs, enabling PECAN to scale well to large traces.

The salient property of persuasiveness is also highly valued and explored by other classes of techniques such as active testing [37, 32, 20, 22] and model checking [38, 21, 27, 44, 3]. In particular, RaceFuzzer and Atomfuzzer [37, 32] dynamically explore and, thus, is capable of creating concrete executions to expose real races by actively controlling the race-directed and randomized thread scheduler. Chess also systematically explores the thread scheduling space at runtime to find concurrency bugs. As a PTA technique, the goal of PECAN is to provide the generalized support of the persuasiveness for concurrency access anomalies.

We have implemented PECAN for Java programs and conducted extensive experiments for evaluating it. Three common types of AAs are investigated: data races, atomicity violations, and ASVs. Our evaluation results show that PECAN is able to effectively and efficiently predict and deterministically create real AAs in all the twenty evaluated subjects including six large multi-threaded applications. PECAN achieves a 100% success ratio of creating the predicted AAs in more than half of the subjects. For the other subjects, the success ratio is from 0.25 to 0.93 (due to the reported false AAs). Several serious and previously unknown bugs were also revealed by PECAN in large open source concurrent systems such as `OpenJMS` and `Jigsaw`. Moreover, PECAN has good scalability that can, for instance, analyze a trace in `Derby` with more than 447K events in around 6 seconds. The PECAN prototype and the detected replayable bugs in our experiments are publicly online at `http://www.cse.ust.hk/prism/pecan/`.

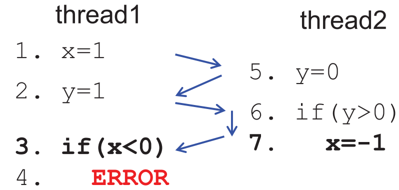

Figure 1: A real race involving two shared variables

In summary, this work makes the following contributions:

- We present a persuasive PTA technique as well as a prototype tool PECAN for detecting general access anomalies in concurrent Java programs. PECAN not only predicts access anomalies, but also generates "bug hatching clips" that deterministically instruct the input program to exercise the predicted AAs.
- We present an efficient static thread schedule generation algorithm, with proof of the soundness, that guarantees to generate a feasible schedule for every real AA in programs that use locks in a nested way.
- We present a general specification model of access anomalies and a prediction model that models the problem of access anomaly prediction as a graph pattern search problem. The graph compactly encodes the happens-before relationship between the events and the scheduling order of memory accesses in the trace, and supports efficient pattern search of AAs to enable PECAN to scale well to large traces.
- We evaluated PECAN using twenty-two multi-threaded subjects including six large concurrent systems and our experiments demonstrate that PECAN is able to effectively predict and deterministically expose real AAs.

2. PECAN IN A NUTSHELL

To make our technique more comprehensible, we first use a simple example to illustrate the AA detection process of PECAN. The simple code in Figure 1 contains three races on the two shared variables `x` and `y`. Let us use the line number as the identifier of the statement. The races are between statements `(2,5)`, `(2,6)` and `(3,7)`. Among the three real races, detecting the race `(3,7)` is comparably more important because it might trigger the `ERROR` at line 4.

PECAN addresses the above problem using the following steps: 1. We first collect traces of interesting events during the program execution. 2. We extracts from the trace a *partial and temporal order graph* (`PTG`) that encodes the information about the happens-before relationship between the events, the atomic blocks, and the scheduling order of memory accesses. 3. We perform a pattern-directed search on the `PTG` for the matching of the general AA patterns w.r.t. the program constraints. 4. Taking the original trace and the search results as the input, we statically generates a thread schedule for each predicted AA. 5. We use a deterministic replayer [17] to re-execute the program to expose the predicted AAs according to the generated schedules.

	Examples		
	data race	atomicity violation	ASV
E	e1 – e2	e1 – e2 – e3	e1 – e2 – e3 – e4
T	t1 – t2	t1 – t2 – t1	t1 – t2 – t2 – t1
SV	s1 – s1	s1 – s1 – s1	s1 – s1 – s2 – s2
AR	u1 – u2	u1 – u2 – u1	u1 – u2 – u2 – u1
AT	r – w	r – w – r	w – r – r – w

Figure 2: General access anomaly patterns

Coming back to our example, suppose the collected execution trace is `<1,5,2,3,6,7>`. In Step 3, PECAN will detect that `(3,7)` is a possible race and then, in Step 4, PECAN is able to generate the thread schedule `<1,5,2,6,7,3>` that deterministically directs the replayer to expose this race and to trigger the `ERROR` in Step 5. From the user's perspective, the whole process is autonomous and requires no additional user intervention. We note that, like other PTA techniques, our analysis requires the fact that the error inducing events `(3,7)` appear in the input trace, which might not always happen. In practice, we can use techniques such as RaceFuzzer [37] to compensate this difficiency[2].

In the following sections, we go under the hood of our technique to discuss the pattern language we use to specify the general AAs (Section 3), the graph prediction model (`PTG`) we use to represent the AA prediction problem (Section 4), the pattern search algorithm for locating the AAs on the graph model (Section 5), and the schedule generation algorithm for generating the thread schedules for each predicted AAs (Section 6).

3. PATTERN SPECIFICATION OF ACCESS ANOMALIES

The most commonly known AAs include data race, atomicity violation, and atomic-set serializability violations (ASV). These anomalies are sequences of 2 - 4 events generated by two different threads on one or two shared variables. In our prediction model, we generalize the concept of AA to allow arbitrary number of events, threads and shared variables, and we describe each type of AA as an event sequence pattern.

An AA pattern `p` is comprised of a group of equal-length sequences `[E,T,SV,AR,AT]`. The meaning of each symbol is described as follows:

- `E`: the event sequence defined by the pattern.
- `T`: the thread scheduling order corresponding to `E`, i.e., the event `E[i]` is by the thread `T[i]`.
- `SV`: the accessed shared variable sequence corresponding to `E`, i.e., the event `E[i]` accesses the shared variable `SV[i]`.
- `AR`: the atomic region sequence corresponding to `E`, i.e., the event `E[i]` belongs to the atomic region `AR[i]`.
- `AT`: the access type sequence corresponding to `E`, i.e., the access type of `E[i]` is `AR[i]` which is either a read or a write: `AR[i]` $\in$ {`r`,`w`}.

[2]We come back to this issue in Section 8.5.

Figure 2 shows example patterns of the three commonly known AAs. Clearly, the specification of AA patterns above is general enough to describe all the three commonly known AAs. Moreover, this general pattern model allows the users to define their own AA patterns that may contain much more complex thread interleavings. Nevertheless, since in fact all complex AA patterns can be composed by these three basic ones[3], we focus on explaining them in this section.

We next discuss these three basic AAs and describe them using the general pattern specification. Since they in total contain a dozen of patterns, for brevity, we only show one representative pattern for each of them. The others patterns are similar.

Data race A data race occurs when two threads are concurrently accessing the same data without the proper synchronization and at least one of these accesses is a write [36]. We thus can describe it as: `E`=e1-e2, `T`=t1-t2, `SV`=s1-s1, `AR`=u1-u2, and `AT`=`r-w`, meaning that the first thread reads a shared variable and immediately the second thread writes to it. Note that data race patterns require the two events happen consecutively to each other, while this condition is unnecessary for atomicity violation and ASVs.

Atomicity violation An atomicity violation happens when the desired serializability among multiple memory accesses to a single memory location is violated [10]. Suppose a memory location is accessed by three consecutive events e_i, e_k, and e_j in this written order, and e_i, e_j belong to the same atomic region, e_k belong to another. An atomicity violation with the three access type"*write-read-write*" can be written as `E`=e1-e2-e3, `T`=t1-t2-t1, `SV`=s1-s1-s1, `AR`=u1-u2-u1, and `AT`=`w-r-w`.

ASV Atomic-set serializability is a criterion for enforcing the serializability of units of work that deal with atomic sets. An atomic set is defined to be a set of memory locations together satisfy some consistency property. Vaziri et al. [42] summarized a set of problematic data access patterns that violate atomic-set serializability. For example, let $W_u(m)$ ($R_u(m)$) represent a write (read) access to a memory location, m, by a unit of work, u, and suppose m_1 and m_2 belong to the same atomic set. The execution sequence "$W_u(m_1) - R_{u'}(m_1) - R_{u'}(m_2) - W_u(m_2)$" causes an ASV as the two consecutive writes to m_1 and m_2 by u are interleaved by two reads to these memory locations by u', another unit of work, resulting in inconsistent reads. We describe this pattern as `E`=e1-e2-e3-e4, `T`=t1-t2-t2-t1, `SV`=s1-s1-s2-s2, `AR`=u1-u2-u2-u1, and `AT`=`w-r-r-w`.

In our implementation, we consider each atomic region as a unit of work and all memory locations accessed in the same atomic region belong to the same atomic set.

4. GRAPH PREDICTION MODEL

Our approach to the general prediction problem of AAs is to reduce it to a graph search problem. We start by formalizing the representation of the trace and the permutation constraints. We then describe our formulation of the AA prediction problem as a graph mutation and pattern search problem.

4.1 Trace

[3]As proved in [42], a set of eleven ASV patterns forms a complete set of all the problematic thread interleaving scenarios w.r.t. atomic sets and units of work.

A *trace* captures a multi-threaded program execution as a sequence of events $\delta = \langle e_i \rangle$. We associate each event e_i with the following attributes:

- i: the global order of e_i in δ;
- t: the thread executing e_i;
- m: the memory location accessed by e_i;
- a: the access type of e_i, where $a \in$ {READ, WRITE, LOCK, UNLOCK, WAIT, NOTIFY, FORK, JOIN};
- l: the locks held by the thread executing e_i when e_i is executed;
- u: the atomic region to which e_i belongs.

An atomic region is defined as a region of code fragments that preserves certain consistency properties w.r.t. the program states. Similar to [47] and with no loss of generality, we consider every synchronized method and every synchronized block as an atomic region. In addition, the *fork*, *join*, *wait*, *notify* operations are considered to be region boundaries. In the case of nested regions, an event e_i belongs to the outermost one.

In our presentation, we use $t(i)$, $m(i)$, $a(i)$, $l(i)$, and $u(i)$ to denote the attributes t, m, a, l, u associated with the event e_i respectively.

4.2 Constraint Model

Precisely detecting access anomalies in general is computationally intractable [30]. To achieve efficiency in predicting AAs, similiar to many race detection techniques [6, 37], we use a hybrid constraint model [31] that combines the lockset condition [36] and the happens-before relation [23]. Specifically, the hybrid model defines that two events e_i and e_j are *independent iff*

1. they do not hold a common lock ($l(i) \cap l(j) == \emptyset$);
2. they do not have a partial order relation (POR) between each other ($e_i \nrightarrow e_j$ and $e_j \nrightarrow e_i$).

The POR is defined as follows:

Definition 1. *The* ***Partial Order Relation*** *(POR)* $e_i \rightarrow e_j$ *holds whenever* e_i *occurs before* e_j *and one of the following holds:*

- **Program order***: both events are by the same thread.*
- **Fork-join order***:* e_i *is the event that starts the thread* $t(j)$ *or* e_j *is the event that blocks until the thread* $t(i)$ *terminates.*
- **Wait-notify order***:* e_i *is the event that sends some signal that the event* e_j *waits for.*

Notice that the hybrid constraint model we use is a conservative approximation of the precise model for checking the independence between events [29]. Therefore, it is a possible source for PECAN to report false warnings during the pattern search. Nevertheless, these false warnings can be automatically pruned during the re-execution phase (see Section 6.3), hence, do not affect the final results delivered to the end user.

4.3 The AA Prediction Problem

The essential idea behind the AA prediction is that the *independent* events in the trace can be rearranged, simulating the thread scheduling effects. Therefore, even if a AA is not directly witnessed in the trace, as long as it can be manifested in any feasible permutation of the trace, we can locate it and expose it with a concrete execution. This idea is initiated in Lipton's theory of reduction [24] and has been exploited by many concurrency bug detection approaches [10, 13, 47].

Our general objective is to search all the AAs that satisfy some given patterns on an execution trace or on any of its feasible permutations allowed by our constraint model defined in Section 4.2. We model this problem as a graph pattern search and mutation problem. Before giving a formal problem definition, let us first define the graph model:

Definition 2. *The* ***Temporal Order Relation*** *(TOR)* $e_i \dashrightarrow e_j$ *holds if events* e_i *and* e_j *are consecutive accesses on the same shared memory location and* e_i *occurs before* e_j*.*

Definition 3. *A* ***Partial and Temporal Order Graph*** (`PTG`) *is a graph* $G(V, E)$ *where* V *is a set of nodes and* E *is a set of edges. Each* $v_i \in V$ *corresponds to the event* e_i *in the trace. Each edge e is either solid* ($\rightarrow$) *or dashed* ($\dashrightarrow$)*, corresponding to the POR and TOR between the events, respectively.*

The `PTG` can be mutated by interchanging the nodes connected by dashed edges w.r.t. the POR and the lockset condition. For brevity, we call these two conditions as mutation condition, and we refer to these mutated `PTG`s as `vPTG`*s*.

Based on the `PTG`, the AA patterns can be conveniently formulated as propositional formulas between the nodes in the `PTG`. Our goal is to *find all the AAs on the vPTGs that satisfy the user specified patterns.*

5. GRAPH PATTERN SEARCH

Since the number of `vPTG`s is exponential and the size of trace could be very large, it is inefficient to perform pattern search on every individual `vPTG`. We use two primary techniques to achieve the efficiency. First, we have developed a compact encoding of the `PTG`. Second, we perform pattern-directed graph mutations on the fly based on the intermediate search results, hence, does not require separate mutation steps.

5.1 Compact Encoding of PTG

We have two main techniques for the compact encoding of the `PTG`. First, to facilitate the efficient pattern search, we build separate indices of events based on the thread ID, the memory location, the access type and the atomic region. Second, to scale to large traces, we do not maintain the full POR but, instead, maintain only the relations between the thread communication (TC) events, i.e., *fork*, *join*, *notify*, and *wait* events. Since the TC events are the only sources of the POR between events across different threads, we use them to compute the POR for all the other events on demand. By this approach, we reduce the space cost from quadratic to the trace size to linear to the trace size and quadratic to the number of TC events. The number of TC events are usually much smaller compared to the entire trace size.

5.2 Pattern-Directed Search

In general, given a pattern described in the specification model in Section 3, our pattern search algorithm first decides in the pattern the number of threads, the number of shared variables, and the number of events by each thread in the same atomic region on each shared variable. Our algorithm then uses this information to search on the indexed `PTG` to obtain a set of candidate AAs. The candidate AA may not match the thread scheduling order `T` specified in the pattern, in which case the mutation condition is applied to check whether there exists certain allowed permutation of nodes in the `PTG` that makes the matching possible. We next give detailed explanation for data race, atomicity violation, and ASV patterns.

Data race Recall that each pattern of data race contains two events satisfying the conditions defined in Section 3. We thus follow the dashed edges on the `PTG` and examine every candidate node pair that could possibly satisfy the conditions. If a node pair (v_i,v_j) matches the temporal order (i.e., the two nodes are connected by a dashed edge), we just report it as a real AA. Otherwise, we check if the `PTG` can be mutated for the node pair to match the temporal order. The function *canSatisfyByMutation*(v_i,v_j) (Algorithm 1) is used to check this condition.

Algorithm 1 canSatisfyByMutation(v_i,v_j)

Ensure: $i < j$
1: return $(l(i) \cap l(j) \neq \emptyset$ && $!POR(v_i, v_j))$

Algorithm 2 canSatisfyByMutation(v_i,v_k,v_j)

Ensure: $i < j < k$
1: **for all** $v_x \in [v_{i+1}, v_{i+2}, \ldots, v_j]$ **do**
2:   **if** $canSatisfyByMutation(v_x, v_k)$ **then**
3:     return true
4:   **end if**
5: **end for**
6: return false

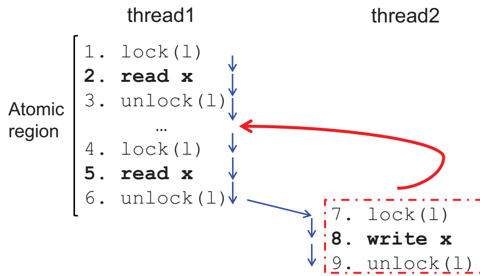

Figure 3: Example of searching atomicity violations

Atomicity violation and ASV The search algorithms for atomicity violation and ASV patterns are similar to that of data race, with the main difference in checking the mutation condition. Because each atomicity violation (ASV) pattern contains three (four) nodes, we need to check the mutation condition for more pairs of nodes compared to that data race. Without losing of generality, we use the example in Figure 3 to illustrate the mutation condition (Algorithm 2) for checking candidate atomicity violations. Suppose we have already found the candidate triple (v_2, v_8, v_5) by traversing the events by the two threads on the shared memory location `x`. As the temporal order of this triple does not directly satisfy the atomicity violation pattern, we next check if it can be satisfied in any of the other `vPTG`s, i.e., if v_8 can be placed *in any position* between v_2 and v_5 without violating the POR and the lockset condition. Our algorithm thus tries to find a position between v_2 and v_5, say v_x, such that there is no POR between v_8 and v_x and they are not protected by a common lock, i.e., the lockset condition. Finally we find $v_x = v_4$ and thus report this AA.

6. SCHEDULE GENERATION

For each predicted AA, PECAN statically generates a corresponding thread schedule that is used to deterministically direct an execution for exposing the AA. This problem is highly nontrivial and there are several challenges to be addressed:

1. Given an AA, regardless of real or false, how to generate a schedule that can manifest it?
2. For each AA, there might be multiple corresponding schedules. Which one should we generate?
3. For real AAs, how to make sure the generated schedules are feasible, i.e., can expose the real AAs?

In the following text, we first present our schedule generation algorithm and discuss how it addresses the above challenges. Then we formally prove that, for programs using nested locks, our algorithm guarantees to generate a feasible schedule for every real AA. For false AAs, although our algorithm may also generate infeasible schedules, we show in Section 6.3 that how these false AAs can be automatically pruned away during the re-execution phase.

6.1 How to Generate a Feasible Schedule?

The basic idea of the our schedule generation algorithm is to transform the original trace by changing the relative order of independent events, i.e., moving the related events to different positions in the trace. The main challenge is that we need not only to make sure the transformed trace can manifest the AA, but also to guarantee it is feasible (i.e., does not violate the program constraints). However, as there are exponential number of ways to transform the trace, it is very inefficient to exhaustively generate every possible schedule and verify its feasibility by checking the constraints. Figure 4 shows a simple trace in which the nodes v_1,v_2,v_3 and v_4,v_5,v_6 belong to two different threads, and the POR and TOR are represented by solid and dashed edges respectively. Suppose (v_2,v_5) is a real race pair. There are many possible rearrangements of the nodes in which we can place v_2 and v_5 next to each other, but only some of them are feasible schedules. For instance, if we naively move v_2 to the position before v_5, we will get an infeasible schedule δ', in which the relative order between v_2 and v_3 violates the POR.

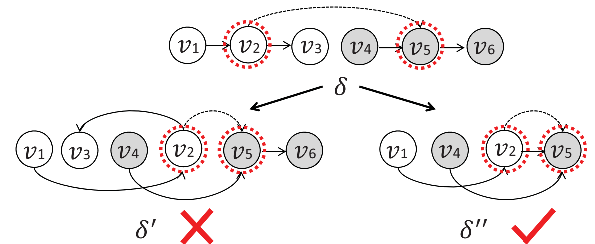

Figure 4: An example of schedule generation

We have the following tactics to reduce the computational complexity of the schedule generation: First, although there might be many feasible schedules that manifest a real AA, it is sufficient for us to generate one of them. Second, since the original trace is a feasible schedule (i.e., satisfies the program constraints), when we permute the original trace (e.g., move a node to a different position), we only need to make sure the changed portion does not violate the constraints w.r.t. the entire trace. Third, since it is sufficient for the resulting schedule to manifest the violation, we can remove from the schedule the nodes that are placed beyond the violation creation point.

With these tactics, the whole schedule generation process becomes clear and straightforward. Now the key problem is how to satisfy the program constraints when permuting the nodes. There are basically three types of program constraints: the POR, the lock constraint, and the program control constraint. The lock constraint requires that, at any time of the program execution, a lock cannot be held by more than one thread. The program control constraint is related to the execution order determined by the evaluation results of program control statements. For real AAs, we can ignore the program control constraint as the evaluation results of program control statements should be unchanged if we move the violation node to a correct position that manifests the AA; otherwise, the AA is not real. We next discuss how our algorithm respects the POR and the lock constraint.

Satisfying the POR is relatively simple. The key point is that, we should not only move the violation node to the correct position such that the violation pattern can be satisfied, but also move the nodes that are dependent on, or having PORs with, the violation node, to their correct positions. Back to the example in Figure 4, we generate a correct schedule δ'' by first moving v_2 and v_3 (because v_3 is dependent on v_2) to the position next to v_5, and then removing v_3 and v_6 from the schedule (because v_3 and v_6 are beyond the violation creating point).

Satisfying the lock constraint is much more complicated. We next first use an example to elucidate the challenge and then describe our approach for addressing it.

Example In Figure 5, the race pair (v_3,v_8) satisfies our relaxed mutation constraints, i.e., v_3 and v_8 are not protected by a common lock and there is also no POR between them. Therefore, it would be reported as a possible race pair by our pattern search algorithm. However, it is a false warning: it is impossible for v_3 and v_8 to happen next to each other in any feasible schedule, as there is a POR between v_2 and v_5. For this false violation, if we only consider the POR in the schedule generation, we would generate an infeasible schedule <v_1,v_2,v_5,v_6,v_7,v_3,v_8> that violates the lock constraint. This is fine as this false violation can be pruned in the re-execution phase. The problem is, however, if we remove the partial order relation from v_2 to v_5 and (v_3,v_8) becomes a real race, this schedule is still infeasible.

The root cause of the problem above is that, by moving the dependent nodes on the to-be-moved violation node (v_3 in Figure 5), we have moved an *unlock* node (v_4) but not its corresponding *lock* node (v_1), causing the resulting schedule to violate the lock constraint. To address this problem, *whenever we move a unlock node, we should also make sure its corresponding lock node is moved to a correct position.* Thus, in addition to the steps illustrated in Figure 4, our algorithm also looks for the *outermost lock* (OML) node

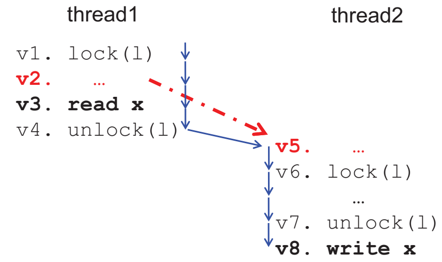

Figure 5: An example for illustrating the difficulty of satisfying the lock constraint for schedule generation. The race pair (v_3,v_8) is a false warning, though it satisfies both the POR and the lockset condition.

protecting the to-be-moved violation node, and moves all the dependent nodes on the OML node to their correct positions. For the example in Figure 5, we first find v_1 (the OML node) and move v_1, v_2 and v_3 (the nodes dependent on the OML node) to the positions before v_8, then we move v_4 to the position after v_8 and remove v_4 afterwards. Finally we get a feasible schedule <v_5,v_6,v_7,v_1,v_2,v_3,v_8>.

Algorithm 3 ScheduleGeneration(v_i,v_j)

Require: $i < j$
1: Let v_l be the outermost *lock* node that is protecting v_i
2: Move all the nodes dependent on v_i to the positions after v_j
3: **if** v_l is not *NULL* **then**
4: Move v_l and all the nodes from v_l to v_i that are dependent on v_l to the positions before v_j
5: **else**
6: Move v_i to the position immediately before v_j
7: **end if**
8: Remove all nodes after v_j

Algorithm 3 summarizes our schedule generation algorithm for data race patterns. Since all it does is moving a sequence of nodes to different positions, the worst case time complexity of this algorithm is linear in the length of the trace. The algorithms for the other AA patterns, such as atomicity violation and ASV patterns, are in a similar style, though may require to move more nodes if the pattern contains three or more events. For example, Algorithm 4 shows our algorithm for atomicity violation patterns, which contain an event triple (v_i,v_k,v_j). The goal of the algorithm is to generate a feasible schedule in which v_k is placed between v_i and v_j. With no loss of generality, let us consider the case $i < j < k$. Recall that in reporting every potential atomicity violation in the pattern search phase, we have found a node v_x which is between v_i and v_j and satisfies the mutation condition with v_k. This means that in some feasible schedule v_k can be placed before v_x. We thus generate such a feasible schedule by the safest and simplest way: move all the nodes from v_x to v_j in the original trace that are dependent on v_x to the position after v_k. The movement of nodes simply follows the same rule as that in the algorithm for data race patterns.

6.2 What Can Our Algorithm Guarantee?

Theorem 1. *For programs that use locks in a nested way, i.e., releasing locks reverse to the acquisition order, our schedule generation algorithm guarantees to produce a feasible schedule for every real AA.*

Algorithm 4 ScheduleGeneration(v_i,v_k,v_j)

Require: $i < j < k$
1: Find v_x in $canSatisfyByMutation(v_i, v_k, v_j)$
2: Let v_l be the outermost *lock* node that is protecting v_x
3: **if** v_l is not $NULL$ **then**
4: Move v_l and all the nodes from v_l to v_{x-1} that are dependent on v_l to the positions before v_k
5: **else**
6: Move all the nodes from v_x to v_j that are dependent on v_x to the positions after v_k
7: **end if**
8: Remove all nodes after v_j

PROOF. Since the essential idea of the schedule generation is event permutation: moving events or event sequences in the original trace from one place to another, to prove the correctness in general (for any AA), it is sufficient to prove the correctness for the most basic step: moving a single event. Now let us pick a race pair (v_i,v_j) with $i < j$ for the proof. Suppose (v_i,v_j) is a real race but the schedule generated by Algorithm 3 is infeasible. Following we prove it is impossible by contradiction.

Because the schedule is infeasible, it must have either violated the POR, the program control constraint, or the lock constraint. For the POR, because Algorithm 3 only changed the temporal order between v_j and the nodes that were moved to the positions after v_j, i.e., nodes dependent on v_i, the only possible POR the generated schedule may violate is between v_j and the nodes that are dependent on v_i. However, for any of such PORs, say $v_x \to v_j$, we must have $v_i \to v_x$ and then $v_i \to v_j$ that contradicts to the condition that there is no POR between v_i and v_j, which must be satisfied for our algorithm to report this AA. Besides, it cannot violate the program control structure neither; otherwise the race is infeasible. Thus, it is impossible for the generated schedule to violate the POR or the program control constraint.

We next prove that it is also impossible to violate the lock constraint. If the schedule violates the lock constraint, then there must exist an unmatched *lock* and *unlock* node pair, i.e., the *lock* node and its corresponding *unlock* node is interleaved by another *lock* or *unlock* node. However, because the original trace satisfies the lock constraint, there are only two possible reasons for this result: **(I)** we incorrectly moved the interleaved *lock* or *unlock* node to a position between the *lock* and the *unlock* node; **(II)** we incorrectly moved the *unlock* node to a position after the interleaved node. Case **I** is impossible because it violates the lockset condition which should be satisfied for our algorithm to report this AA. For case **II**, we show it is also impossible if there are only nested locks in the original trace. First, because our algorithm only moves those nodes that are dependent on the outermost *lock* (OML) node that is protecting the violation node, if we had ever moved an *unlock* node, this *unlock* node should be dependent on the OML node. Additionally, if there are only nested locks in the trace, the corresponding *lock* node of this *unlock* node should also be dependent on the OML node, otherwise the OML node would not be the outermost *lock* node. Thus, if we had ever moved an *unlock* node, we should have also moved its corresponding *lock* node to a correct position. So case **II** is also impossible.

6.3 Pruning False Warnings

Note that our schedule generation algorithm is sound but incomplete, i.e., it may generate infeasible schedules for false violations. Nevertheless, we are able to automatically prune all the false AAs away during the re-execution phase. Specifically, during the re-execution, we control the thread scheduling of the re-execution to strictly follow an input generated schedule by matching the events between the two schedules. When we observe that some thread has executed a new event that does not match the corresponding event in the input schedule, which means the thread has taken a different branch from the original observed execution, or the re-execution hangs due to a deadlock, we immediately stop the re-execution and report the AA is a false violation. In this way, as we only report successful re-executions, we are able to prune all the false violations.

7. IMPLEMENTATION

We have implemented PECAN based on our LEAP record and replay framework [17]. In addition to the graph pattern search and the schedule generation algorithms presented in Section 5 and Section 6, the other three technical components of PECAN include the instrumentation, the trace collection, and the deterministic scheduling. We next briefly describe them.

Instrumentation The instrumentation phase takes the bytecode of an arbitrary Java program and produces two versions: the record version and the replay version. The record version is executed for the trace collection, and the replay version is used in the deterministic scheduling phase to create a concrete execution that exposes the predicted AA.

To allow us to collect all the needed trace information at runtime, we instrument all the program points that may emit relevant events in the trace. Specifically, we instrument all the possible shared variable access points[4] and all the monitor entry and exit points, to track the shared variable access events and the lock acquisition and release events, during the program execution. To track the thread communication events, we instrument thread *fork* and *join*, thread *start* and *exit*, and object *wait* and *notify* points. To get the atomic region information of each event, we also instrument the entry and exit points of synchronized methods.

The only difference between the instrumentation of the record version and of the replay version is the treatment of the synchronization points. For the replay version, to enable the application level thread scheduling (discussed later in this section), we insert the instrumentation *before* the original synchronization points in the program, and transform all synchronized methods of the application classes to synchronized blocks.

Trace collection To represent the trace, we maintain a vector to record a global order of all the events. For all the events, we record their access type, thread ID and the accessed memory ID at runtime. The lock set and the atomic region information are computed offline to save runtime cost. For re-entrant locks, like [12], we process them internally in the trace collection phase and will not expose them to the resultant trace.

Deterministic scheduling The deterministic scheduler

[4]To remove unnecessary recording cost on local variables, we first perform a static thread escape analysis [15] to identify all the possible shared variables in the program.

in PECAN is based on our previous record/replay work [17]. Following the transformed problematic schedule, the deterministic scheduler controls the scheduling of threads to enforce a deterministic execution order of all the events and to create the predicted AA. To support the deterministic scheduling, there are two issues to be addressed. First, we need a way to enable the application level thread scheduling. Second, we have to match the thread identity at the deterministic scheduling phase with the correct one during the trace collection phase.

To enable the application level thread scheduling, we associate each thread in the deterministic scheduling phase with a semaphore maintained in a global data structure, so that each thread can be suspended and resumed on demand. In each execution step, we make sure that only the thread with same ID and event attributes as the ones in the input schedule is allowed to proceed. We also use an extra thread to actively check whether the predicted AA has been created or not. If all the threads are suspended before the AA is created, we report the predicted schedule has failed, indicating a false violation.

To make sure that a thread at the trace collection phase is correctly recognized during the deterministic scheduling phase, we introduce additional synchronization operations at the thread creation time to ensure the same thread creation order across these two phases. More specifically, at the trace collection phase we maintain a list that records the IDs of the parent threads in the global order of their thread creation operations. The list is used to direct the same thread creation order at the deterministic scheduling phase.

8. EVALUATION

8.1 Evaluation Methodology

We use a set of popular subjects (Table 1), used in benchmarking the concurrency defect analysis techniques [32, 37, 22], and a number of large multi-threaded Java applications to evaluate PECAN. The first seven subjects (*Account*, *BuggyPrg*, *Critical*, *Loader*, *Manager*, *MergeSort* and *Shop*) are from IBM ConTest benchmark suite [8], the five subjects in the middle (*StringBuf*, *ArrayList*, *LinkedList*, *HashSet* and *TreeSet*) are open libraries from Suns JDK 1.4.2. *Moldyn*, *RayTracer*, and *MonteCarlo* are standard benchmarks from the Java Grande Forum, *Cache4j* is a thread-safe implementation of cache for Java objects, *SpecJBB*-2005 is SPEC's benchmark for evaluating the performance of server side Java, *Hedc* is a web-crawler application kernel, and *Weblech*-0.0.3 is a multi-threaded web site download and mirror tool. To assess the ability of PECAN in predicting real concurrency bugs in large mature multi-threaded applications, we also include in our experiments *Derby*-10.3.2.1, Apache's widely used open source Java RDBMS, *Jigsaw*-2.2.6, W3C's leading-edge web server platform, and *OpenJMS*-0.7.7, an open source implementation of Sun Microsystems's Java Message Service API 1.1 Specification.

In all our experiments, we collect a normal execution trace for each program with the fixed configuration setting and program input. For each generated schedule, we re-execute the program once to verify whether the corresponding predicted AA is created or not. Because of the concurrency bugs, some subjects may throw uncaught exceptions in certain problematic schedules. It is clearly a highly desirable and useful characteristic if a technique is able to predict these concurrency bugs from a normal execution trace, and generate the corresponding schedules to cause the program to raise uncaught exceptions. Thus, in our evaluation, we also report the number of re-executions in which the program raise uncaught exceptions, out of all the schedules generated by PECAN, for each evaluated program.

To remove the nondeterminism caused by random numbers, we replace all the random seeds in the evaluated programs with a constant. For those open libraries, we use the drivers from [18] to close them. All the experiments were conducted a 8-core 3.00GHz Intel Xeon machine with 16GB memory and Linux version 2.6.22. The VM configuration is standard Java HotSpot (TM) 64-Bit Server VM with version 1.6.0_10 with 10G heap space, which is sufficient for all our experiments.

8.2 Experimental Results

Table 1 summarizes the results of our experiments. For each program, Column 2 reports its size in lines of source code (*LOC*), Column 3-5 report the number of threads (*Thread*), the number of real shared memory locations that contain both read and write accesses from different threads (*SM*), and the number of events in the trace (*Event*) that we analyzed, respectively. The thread number ranges from 2 in *RayTracer* to 24 in *OpenJMS*, the number of shared memory locations ranges from 1 to 399, and the trace size ranges from 19 to 447,392.

Column 6 reports the runtime overhead (*Overhead*) of our trace collection[5]. The runtime overhead ranges from 0.00x in *Account* and *Manager* to 7.84x in *Moldyn*. Columns 7-8 report the pattern search time (*Analysis*) and the average schedule generation time (*Transform*). The pattern search time ranges from 3ms in *StringBuf*, with 86 events in the trace, to around 5 minutes in *OpenJMS* with 180,887 events in the trace. The average schedule generation time ranges from 2ms in *Critical* to 1.473s in *Derby*.

Columns 9-11 report the number of predicted data races (*Race*), atomicity violations (*AV*), and ASVs (*ASV*), respectively, in each program. PECAN has predicted a number of data races and atomicity violations in almost all the traces we analyzed. The number of predicted ASVs is often zero or very small except for *Jigsaw*, in which PECAN predicted 684 ASVs. Note that each AA reported by PECAN is unique in terms of the source code line numbers on which the violation events are triggered. We do not report duplicate AAs that have the same line number combinations in the source.

Columns 12-14 report the number of created real AAs (*REAL*), the number of re-executions that raise uncaught exceptions (*EXP*), and the number of re-executions that fail (*F*). For the three large programs (*OpenJMS*,*Jigsaw*,*Derby*) marked with '*', because they contain too many predicted AAs (from 437 to 2,076), we only generate the schedules for 100 randomly selected AAs. PECAN created real AAs for all the evaluated programs and, for most of them, PECAN caused the program to throw uncaught exceptions, which is a strong symptom of real concurrency bugs. PECAN also reported a number of failed re-executions in several subjects, especially those large programs. We manually inspect those failures and find that the only reason why PECAN fails to create these AAs is that these AAs are false violations, due

[5]The overhead was averaged over 10 runs for each subject.

Table 1: Experimental results

Program	LOC	Trace			Computation			Violation			Result		
		Thread	SV	Event	Overhead	Analysis	Transform	Race	AV	ASV	REAL	EXP	F
Account	148	3	4	66	0.00x	6ms	7ms	7	2	6	15	8	0
BuggyPrg	385	4	5	225	0.67x	12ms	8ms	9	1	0	10	1	0
Critical	70	3	1	19	0.33x	2ms	2ms	16	14	0	28	2	0
Loader	148	4	1	48	0.01x	4ms	8ms	2	4	0	5	1	0
Manager	212	5	3	158	0.00x	30ms	14ms	15	0	0	14	1	1
MergeSort	456	6	2	1,587	0.50x	22ms	10ms	8	10	0	8	1	10
Shop	280	4	1	429	0.20x	101ms	32ms	4	0	0	4	1	0
StringBuf	1,320	3	2	86	1.40x	3ms	43ms	0	1	0	1	1	0
ArrayList	5,866	3	3	294	0.14x	6ms	8ms	6	1	4	11	2	0
LinkedList	5,979	3	9	357	0.13x	8ms	7ms	6	1	7	13	3	0
HashSet	7,086	3	11	404	0.62x	12ms	8ms	4	3	4	10	5	0
TreeSet	7,532	3	23	475	0.21x	30ms	7ms	4	2	4	10	4	0
Moldyn	1,352	2	11	134,375	7.84x	5.622s	265ms	4	0	0	1	0	3
RayTracer	1,924	2	399	15,140	1.25x	1.034s	68ms	2	2	0	4	0	0
MonteCarlo	3,619	2	9	7,650	1.69x	309ms	20ms	1	0	0	1	0	0
Cache4j	3,897	5	19	1,077	0.25x	11ms	12ms	5	2	0	5	0	2
SpecJBB-2005	17,596	4	116	60,775	0.07x	79ms	53ms	24	1	0	18	2	7
Hedc	29,949	7	10	3,117	0.32x	5ms	9ms	25	3	0	12	2	16
Weblech-0.0.3	35,175	3	26	10,640	0.14x	57ms	24ms	10	0	0	4	0	6
OpenJMS-0.7.7*	154,563	24	185	180,887	0.38x	298.6s	350ms	207	226	4	34*	8*	66*
Jigsaw-2.2.6*	381,348	12	307	275,128	3.61x	177.7s	578ms	665	727	684	36*	15*	57*
Derby-10.3.2.1*	665,733	4	99	447,392	1.90x	6.184s	1.453s	144	319	0	38*	6*	62*

to the conservativeness of the hybrid constraint model (recall Section 4.2) we use for AA prediction.

Our experiment results clearly demonstrate the performance and effectivenss of PECAN. First, PECAN has predicted real AAs for all the evaluated subjects and achieves a 100% success ratio of creating the predicted AAs in more than half of the subjects. For the other subjects, the success ratio is from 0.25 to 0.93 (due to the reported false violations). Second, the pattern search and the schedule generation are both relatively fast. For *Derby*, which has more than 447K events in the trace, PECAN predicted 463 AAs in around 6 seconds and generated the corresponding schedule for each AA in around 1.5 seconds on average. For *OpenJMS*, the trace of which contains more than 180K events, PECAN predicted 2,076 AAs in less than 5 minutes. For the other cases with smaller trace size, such as *ArrayList* that contains several hundred events, the pattern search time and the schedule generation time are only several milliseconds. These results clearly demonstrate the efficiency of our pattern search and schedule generation algorithms. Moreover, since we compute most of the event attributes offline, the runtime overhead of PECAN is relatively small, with slowdown factors ranging from 0.00x to 7.84x.

8.3 Detected Real Bugs

We investigated the uncaught exceptions and real AAs that PECAN has created and have confirmed a number of real concurrency bugs in almost all the subjects, and several bugs are previously unknown. We next describe a couple of previously unknown bugs in two large projects *OpenJMS-0.7.7* and *Jigsaw-2.2.6*.

Figure 6 shows a destructive data race predicted by PECAN in *OpenJMS-0.7.7*. The race happens on the field `multiplexer` of the class `MultiplexedManagedConnection`. When a thread first read the shared field at line 2 before it is initialized by another thread at line 1, the thread will throw a `ResourceException` that crashes the program.

Figure 7 shows a predicted real bug in *Jigsaw-2.2.6*. In the

MultiplexedManagedConnection.java

```
setInvocationHandler(…)
{
   …
1. _multiplexer = createMultiplexer(…);
   …
}
```

```
invoke(…)
{
     synchronized (this) {
2.       multiplexer = _multiplexer;
     }
     if (multiplexer != null)
        …
     else
        throw new ResourceException(…);
}
```

Figure 6: A destructive race in OpenJMS

method `getNextEvent` of the class `EventManager`, a thread first checks in a while loop (line 1) until the event queue becomes non-empty, then the thread gets the first item in the queue (line 2) and removes it from the queue (line 3). This logic is correct in a single-threaded event manager. However, when there are multiple threads executing inside the `getNextEvent` method simultaneously, a thread might try to get an item from the queue that has already been removed by another thread, causing an `ArrayIndexOutOfBoundsException` at line 2.

EventManager.java

```
getNextEvent()
{
1. while (queue.size() == 0) {
        …
   }
2. Event e = queue.elementAt(0);
   …
3. queue.removeElementAt(0);
}
```

Figure 7: A predicted real bug in Jigsaw

8.4 Comparing to AssetFuzzer

AssetFuzzer [22] is an active randomized testing technique that also uses the trace information to detect and create

Table 2: Comparison of success ratios in creating real AAs

Program	AssetFuzzer	PECAN
StringBuffer	0.9	1.00
ArrayList	0.35	1.00
LinkedList	0.53	1.00
HashSet	0.35	1.00
TreeSet	0.11	1.00
Moldyn	1.00	1.00
RayTracer	0.998	1.00
MonteCarlo	0.990	1.00
Cache4j	0.933	1.00
Hedc	0.976	1.00
Weblech	0.261	1.00
Jigsaw	0.853	1.00

ASVs in Java programs. The key difference between AssetFuzzer and PECAN lies in their different ways of creating executions to manifest the predicted AAs. Because AssetFuzzer is essentially a randomized technique that dynamically explores certain specific thread schedules from an ocean of thread interleavings, its capability of exposing *real* AAs is subjected to the randomness and cannot provide the persuasiveness property. While PECAN statically generates a full deterministic schedule, it is able to deterministically expose every *real* AA. Table 2 shows a comparison between PECAN and AssetFuzzer on the success ratios in creating *real* AAs for the twelve subjects evaluated by both techniques. The success ratios of AssetFuzzer ranges from 0.11 to 1.00, while PECAN achieves a 100% success ratio for creating real AAs.

8.5 Limitations of PECAN

Our experimental results have clearly demonstrated the superior persuasive concurrency bug prediction capability of PECAN compared to the related approaches. Through our experiments with real world large multi-threaded applications, we also observed some limitations of PECAN that we plan to address in our future work.

Limited path exploration PECAN currently has only the information of a single trace, it can not predict access anomalies in different execution paths that are not present in the collected traces. We plan to enhance PECAN by combining it with approaches such as symbolic analysis [44, 34, 4] to systematically exercise more execution paths. Nevertheless, since the most likely schedules in practice are those that are close to a common execution trace, using a real trace (or, set of them) also allows PECAN to bias the results toward real AA problems that are most likely to cause real problems in practice.

Sensitivity to the original trace Both the pattern search and the schedule generation phases of PECAN are dependent on the original trace. For example, to create the race (3,7) in Section 2, PECAN needs the statements 3 and 7 are both exercised in the original trace. However, such a schedule, e.g., `<1,5,2,3,6,7>` or `<5,1,2,3,6,7>`, could be difficult to manifest in either real executions or test runs. Techniques such as RaceFuzzer is effective in generating error-inducing traces by intelligently exploring thread schedules based on statically detected race pairs. As the future work, we plan to integrate PECAN with this school of techniques to tackle the trace sensitivity issue and to improve the bug detection capability of PECAN.

9. OTHER RELATED WORK

Active testing PENELOPE [40] is a recent technique for testing concurrent programs that exposes atomicity violations by re-executing the program under the atomicity-violating schedules. The atomicity-violating schedules in PENELOPE is generated using a cut-point based theoretical scheduling algorithm that addresses the single variable atomicity problem. Different from PENELOPE, we fundamentally address all known AAs using a general prediction model and AA schedule generation algorithm. The technique of PENELOPE cannot be trivially generalized to address general AAs as its prediction model considers only the locking constraints in the program, which is suitable for atomicity violations but may not for other types of AAs.

Symbolic analysis Wang et al. [46, 45, 43] developed a symbolic analysis model for finding concurrency errors, such as atomicity violations, based on the execution trace. The model encodes the causal dependencies between events, the program control structure, and the property of concurrency errors in a uniform way of symbolic constraints and uses a satisfiability solver to verify the existence of property violations. This approach can statically check whether a property holds in all feasible permutations of events in the given execution trace. However, it still faces the inherent challenge of huge search space and is hard to scale to large traces. Moreover, although the symbolic model is able to exhaustively verify the feasibility of schedules, it is not clear how to efficiently generate a witness that manifests the detected concurrency errors using this approach.

Static analysis Chord [28] and RacerX [7] can explore concurrency bugs by statically analyzing the program. The primary advantage of these static detection techniques is that they can potentially explore all paths to find possible bugs. However, the results of static analysis are often limited in giving the programmers the complete comprehension of the defects because, without runtime information, to give an accurate defective execution history is usually difficult.

Runtime analysis Eraser [36], SVD [48], and the work by Hammer et al. [16] dynamically detect concurrency bugs using the lockset algorithms or some criteria based automata. Bodden and Havelund [1] uses AOP for race detection by providing three novel pointcuts to the AspectJ language. A problem with these techniques is that they are often limited to detect the defects manifested in a specific concrete execution. To address this problem, Goldilock [6], and the work by Choi and O'Callahan [31] use the hybrid model that combines the happens-before and the lock-based approaches to predict races based on an execution. By exploiting the insight that a full generality of the happens-before is unnecessary in most data accesses, FastTrack [11] proposes an adaptive representation for the happens-before relation to efficiently and precisely detect races at runtime.

Model checking [38, 21, 27, 44, 3] is an alternative way to find bugs in concurrent programs. By exhaustively exploring the thread scheduling space, they can also report counter examples for the detected concurrency defects. For example, CHESS dynamically explores the thread scheduling decisions to expose concurrency bugs using a context-bounded approach. Shacham et al. [38] also uses a model checker to construct the witness for data races reported by the lockset algorithm. Unfortunately, due to the exponential size of the search space, it is hard for them to scale to large programs without compromising the detection capability.

10. ACKNOWLEDGEMENT

We thank the anonymous reviewers and our ISSTA shepherd for their constructive comments. This research is supported by RGC GRF grants 622208 and 622909.

11. REFERENCES

[1] Eric Bodden and Klaus Havelund. Racer: effective race detection using aspectj. In *ISSTA*, 2008.
[2] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. Pacer: proportional detection of data races. In *PLDI*, 2010.
[3] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *PLDI*, 2007.
[4] Wang Chao, Limaye Rhishikesh, Ganai1 Malay, and Gupta Aarti. Trace-based symbolic analysis for atomicity violations. In *TACAS*, 2010.
[5] Feng Chen, Traian Florin Serbanuta, and Grigore Rosu. jpredictor: a predictive runtime analysis tool for java. In *ICSE*, 2008.
[6] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: a race and transaction-aware java runtime. In *PLDI*, 2007.
[7] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP*, 2003.
[8] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent bug patterns and how to test them. *IPDPS*, 2003.
[9] Azadeh Farzan, P. Madhusudan, and Francesco Sorrentino. Meta-analysis for atomicity violations under nested locking. In *CAV*, 2009.
[10] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, 2004.
[11] Cormac Flanagan and Stephen N. Freund. Fasttrack: efficient and precise dynamic race detection. In *PLDI*, 2009.
[12] Cormac Flanagan and Stephen N. Freund. Adversarial memory for detecting destructive races. In *PLDI*, 2010.
[13] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI*, 2008.
[14] Zhongxian Gu, Earl T. Barr, David J. Hamilton, and Zhendong Su. Has the bug really been fixed? In *ICSE*, 2010.
[15] Richard L. Halpert, Christopher J. F. Pickett, and Clark Verbrugge. Component-based lock allocation. In *PACT*, 2007.
[16] Christian Hammer, Julian Dolby, Mandana Vaziri, and Frank Tip. Dynamic detection of atomic-set-serializability violations. In *ICSE*, 2008.
[17] Jeff Huang, Peng Liu, and Charles Zhang. LEAP: Lightweight deterministic multi-processor replay of concurrent Java programs. In *FSE*, 2010.
[18] Pallavi Joshi, Mayur Naik, Chang-Seo Park, and Koushik Sen. Calfuzzer: An extensible active testing framework for concurrent programs. In *CAV*, 2009.
[19] Pallavi Joshi, Mayur Naik, Koushik Sen, and David Gay. An effective dynamic analysis for detecting generalized deadlocks. In *FSE*, 2010.
[20] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI*, 2009.
[21] Nicholas Kidd, Thomas Reps, Julian Dolby, and Mandana Vaziri. Finding concurrency-related bugs using random isolation. In *VMCAI*, 2009.
[22] Zhifeng Lai, S. C. Cheung, and W. K. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *ICSE*, 2010.
[23] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 1978.
[24] Richard J. Lipton. Reduction: a method of proving properties of parallel programs. *CACM*, 1975.
[25] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. Literace: effective sampling for lightweight data-race detection. In *PLDI*, 2009.
[26] Nicholas D. Matsakis and Thomas R. Gross. A time-aware type system for data-race protection and guaranteed initialization. In *OOPSLA*, 2010.
[27] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam A. Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.
[28] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *PLDI*, 2006.
[29] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. In *PPOPP*, 1991.
[30] R. H. B. Netzer and B. P. Miller. What are race conditions: Some issues and formalizations. *LOPLAS*, 1992.
[31] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *PPoPP*, 2003.
[32] Chang-Seo Park and Koushik Sen. Randomized active atomicity violation detection in concurrent programs. In *FSE*, 2008.
[33] Soyeon Park, Shan Lu, and Yuanyuan Zhou. Ctrigger: exposing atomicity violation bugs from their hiding places. In *ASPLOS*, 2009.
[34] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. Differential symbolic execution. In *FSE*, 2008.
[35] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. In *PPoPP*, 2003.
[36] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *TOCS*, 1997.
[37] Koushik Sen. Race directed random testing of concurrent programs. In *PLDI*, 2008.
[38] Ohad Shacham, Mooly Sagiv, and Assaf Schuster. Scaling model checking of dataraces using dynamic information. In *PPoPP*, 2005.
[39] 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 *OOPSLA*, 2010.
[40] Francesco Sorrentino, Azadeh Farzan, and P. Madhusudan. Penelope: Weaving threads to expose atomicity violations. In *FSE*, 2010.
[41] Chen Tian, Vijay Nagarajan, Rajiv Gupta, and Sriraman Tallam. Dynamic recognition of synchronization operations for improved data race detection. In *ISSTA*, 2008.
[42] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, 2006.
[43] Kahlon Vineet and Wang Chao. Universal causality graphs: a precise happens-before model for detecting bugs in concurrent programs. In *CAV*, 2010.
[44] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with java pathfinder. In *ISSTA*, 2004.
[45] Chao Wang, Sudipta Kundu, Malay K. Ganai, and Aarti Gupta. Symbolic predictive analysis for concurrent programs. In *FM*, 2009.
[46] Chao Wang, Rhishikesh Limaye, Malay K. Ganai, and Aarti Gupta. Trace-based symbolic analysis for atomicity violations. In *TACAS*, 2010.
[47] Liqiang Wang and Scott D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *PPoPP*, 2006.
[48] Min Xu, Rastislav Bodík, and Mark D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI*, 2005.