Detecting Atomic-Set Serializability Violations in Multi-threaded Programs through Active Randomized Testing*

Zhifeng Lai and S.C. Cheung†
Department of Computer Science and Engineering
Hong Kong University of Science and Technology
Kowloon, Hong Kong
{zflai, scc}@cse.ust.hk

W.K. Chan
Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Hong Kong
wkchan@cs.cityu.edu.hk

ABSTRACT
Concurrency bugs are notoriously difficult to detect because there can be vast combinations of interleavings among concurrent threads, yet only a small fraction can reveal them. *Atomic-set serializability* characterizes a wide range of concurrency bugs, including data races and atomicity violations. In this paper, we propose a two-phase testing technique that can effectively detect atomic-set serializability violations. In Phase I, our technique infers potential violations that do not appear in a concrete execution and prunes those interleavings that are violation-free. In Phase II, our technique actively controls a thread scheduler to enumerate these potential scenarios identified in Phase I to look for real violations. We have implemented our technique as a prototype system ASSETFUZZER and applied it to a number of subject programs for evaluating concurrency defect analysis techniques. The experimental results show that ASSETFUZZER can identify more concurrency bugs than two recent testing tools RACEFUZZER and ATOMFUZZER.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Software/Program Verification – *Reliability*; D.2.5 [**Software Engineering**]: Testing and Debugging – *Testing tools*.

General Terms
Languages, Algorithms, Reliability, Verification

Keywords
Software Testing, Atomicity, Serializability, Dynamic Analysis

1. INTRODUCTION
Multithreaded programs have gained more prominence since the advent of multi-core architecture. At the same time, they incur concurrency bugs that do not exist in sequential programs. These bugs are widespread in both proprietary software [8] and open-source software [17]. Concurrency bugs are difficult to detect because failures of these bugs often manifest themselves only under specific thread interleavings. However, the number of possible interleavings for a multithreaded program is often myriad.

Researchers have proposed various criteria for analyzing concurrency defects. One such criterion is data race freedom. This criterion is unsatisfactory because some data races can be intentional and benign [29]. Recently, a number of serializability criteria have been proposed for multithreaded programs, including atomicity [6], causal atomicity [5], and conflict/view serializability [31]. However, they commonly ignore the correlations among shared variables, such as invariants and consistency properties. These criteria, therefore, do not accurately reflect correct program behaviors, resulting in missed bugs and false warnings. More recently, Vaziri et al. [32] proposed another criterion, called atomic-set serializability. This criterion assumes that a consistency property exists between memory locations, which are grouped into an *atomic set*. Code fragments expected to preserve the consistency of an atomic set are called *units of work*. Atomic-set serializability requires that units of work must be serializable for all the atomic sets that they operate on. This criterion characterizes a wider range of concurrency bugs than many previously proposed criteria. Errors due to data races [26], high-level data races [1], and violations of standard notions of serializability [31] can all be treated as violations of atomic-set serializability. Besides, previous experiences of using this criterion show that it can be more accurate in discerning real concurrency bugs than other existing ones [9].

The atomic-set serializability criterion is useful, but verifying whether a program satisfies this criterion is challenging. Hammer et al. [9] proposed a runtime monitoring technique to detect atomic-set serializability violations based on a set of problematic access patterns proposed in [32]. This approach reports a violation when the execution being monitored matches any such patterns. Although the approach is precise, it suffers from at least one crucial problem from the viewpoint of bug detection. Suppose that a program has a bug that can result in an atomic-set serializability violation. The ability of this approach to detect the violation highly depends on the thread scheduling strategy of the underlying operating system or virtual machine. If the execution being monitored does not exhibit any violations, the approach cannot detect the bug even though other executions may exhibit such a violation. Repeating the monitoring without cautiously controlling thread schedules does not warrant producing a

* This research was partially supported by the Research Grants Council of Hong Kong under grant numbers 612108, 111107, 123207, National Science Foundation of China under grant number 60736015, National Basic Research of China under 973 grant number 2006CB303000, and a grant of City University of Hong Kong under grant number CityU 7008039.

† Correspondence author.

violating execution. Worse still, the chance of hitting certain serializability violations is empirically low even under stress testing [23]. One can alternatively use model checking techniques to enumerate all interleavings. However, model checking techniques for atomic-set serializability [13] are not scalable even for medium-sized programs [12] because the number of possible thread interleavings often grows exponentially with the length of an execution.

In this paper, we propose an active randomized testing technique, called *fuzz testing for **A**tomic-**S**et **SE**rializabili**T**y* (or ASSETFUZZER), to detect atomic-set serializability violations. Active randomized testing was originally introduced by RACEFUZZER to detect real data races [27] in two phases. First, it uses an imprecise race detection algorithm [21] to derive pairs of statements in a multithreaded program that may potentially be in race. These statement pairs are then used to guide a randomized scheduler to create real data races. In this paper, we extend the active randomized testing technique to detect atomic-set serializability violations. Our extension needs to address two new challenges: (1) how to infer potential violations from an execution that exhibits no serializability violations, and (2) how to manipulate executing threads so that we can generate effective thread schedules to detect real violations.

In the first phase, ASSETFUZZER derives a *relaxed partial order execution trace* from a concrete execution. The relaxed partial order execution trace captures interleavings that respect the control flows of the concrete execution [30]. If ASSETFUZZER infers that no interleavings in the same relaxed partial order may violate atomic-set serializability, then these interleavings must satisfy this criterion. Note that existing techniques [9, 13] need to fully explore all these interleavings to discover this fact. If ASSETFUZZER infers that there are potential violations in these interleavings, it records problematic access patterns that capture these violations. In the second phase, ASSETFUZZER controls a thread scheduler to explore only these potentially violating interleavings. This search is efficient because a large portion of violation-free executions have been pruned in the first phase. Further, since access events of a potential violation have been identified, ASSETFUZZER can concentrate on manipulating threads to discover their undesirable occurrence sequences. The combination of violation inference and thread manipulation enables ASSETFUZZER to detect interleavings that violate atomic-set serializability earlier than existing techniques.

We have implemented our techniques in a tool, called ASSETFUZZER, and applied it to 13 Java subjects that have been commonly used to evaluate concurrency defect analysis techniques [6, 22, 27, 31]. The experimental results show that ASSETFUZZER can detect ten times more atomic-set serializability violations than a runtime monitoring algorithm [9] running on top of a randomized scheduler [28]. On average, ASSETFUZZER increases the rate of detecting atomic-set serializability violations by at least threefold. Results of these experiments also show that ASSETFUZZER discovers more concurrency bugs than two recent active testing tools RACEFUZZER [27] and ATOMFUZZER [22]. Not all of these bugs can be discovered even by a combination of these two tools due to the use of atomic-set serializability.

In summary, this paper makes the following contributions:

- We develop a dynamic analysis technique that can infer potential atomic-set serializability violations from a concrete execution. Not only does the technique look for violations in the execution being monitored, but it also infers potential violations in other executions.
- Based on the potential violations derived from the dynamic analysis, we propose an active randomized testing technique which can effectively explore executions that violate atomic-set serializability. Since ASSETFUZZER actually creates executions that violate atomic-set serializability, it does not report false positives.
- We implement the techniques in a prototype system and have applied it to a number of Java subjects. The experimental results show that ASSETFUZZER can effectively detect a larger number of real atomic-set serializability violations than a runtime monitoring technique [9] over a randomized scheduler [28].

The rest of the paper is organized as follows. Section 2 uses an example to motivate our work. Section 3 presents the foundation of our techniques. Section 4 describes the dynamic analysis technique and the active randomized testing technique. Section 5 evaluates the techniques on 13 recently used Java subjects for benchmarking. Section 6 reviews related work and Section 7 summarizes this paper.

2. MOTIVATION

In this section, we use an example (Figure 1) to illustrate the challenge of detecting atomic-set serializability violations and the rationale of ASSETFUZZER to quickly explore violating executions from a violation-free execution. The example is adapted from [9] which illustrates that defect analysis techniques for atomicity [6] and conflict/view serializability [31] can report false warnings that do not reflect real concurrency bugs, while techniques for analyzing atomic-set serializability can filter out such warnings.

2.1 An Example

```
public class Account {
    double checking = 0;           // denoted as c
    double savings = 0;            // denoted as s
    Account(double c, double s) {  // constructor
        setChk(c); setSav(s);
    }
    Account(Account acc) {         // constructor: BUG
        setEqChk(acc); setEqSav(acc);
    }
    boolean isLegal() {
        return (getChk() >= 0) && (getSav() >= 0);
    }
    synchronized void creditInterest(double rate) {
        setChk(getChk() * (1.0 + rate));
        setSav(getSav() * (1.0 + rate));
    }
    synchronized void setEqChk(Account acc) {
        setChk(acc.getChk());
    }
    synchronized void setEqSav(Account acc) {
        setSav(acc.getSav());
    }
L1: synchronized double getChk() { return checking; }
L2: synchronized double getSav() { return savings; }
L3: synchronized void setChk(double c) { checking = c; }
L4: synchronized void setSav(double s) { savings = s; }
}
```

Figure 1. An example program of account system.

The example program is an account system containing a class `Account`, which declares two instance fields `checking` and `savings`. This class has two constructors and several methods operating on the two fields. The method `isLegal()` validates the legality of an `Account` instance. The method `creditInterest()`

t_1: $\overbrace{R_1(x.c)R_1(x.s)}^{\text{isLegal()}^1}\overbrace{\overbrace{R_{2,3}(x.c)W_{2,3}(y.c)}^{\text{setEqChk()}^3}\overbrace{R_{2,4}(x.s)W_{2,4}(y.s)}^{\text{setEqSav()}^4}}^{\text{Account()}^2}\,tr(S1)tr(S2)tr(S3)\cdots\cdots$ $\{y.c = 400, y.s = 700\}$

t_2: $\cdots\cdots tr(S4)tr(S5)tr(S6)\,\overbrace{R_5(x.c)W_5(x.c)R_5(x.s)W_5(x.s)}^{\text{creditInterest()}^5}$

Figure 2. A serial execution.

t_1: $\overbrace{R_1(x.c)R_1(x.s)}^{\text{isLegal()}^1}\cdots\cdots\overbrace{\overbrace{\mathbf{R_{2,3}(x.c)}W_{2,3}(y.c)}^{\text{setEqChk()}^3}\overbrace{\mathbf{R_{2,4}(x.s)}W_{2,4}(y.s)}^{\text{setEqSav()}^4}}^{\text{Account()}^2}\,tr(S1)tr(S2)tr(S3)$ $\{y.c = 404, y.s = 707\}$

t_2: $\cdots\cdots tr(S4)tr(S5)tr(S6)\,\overbrace{R_5(x.c)\mathbf{W_5(x.c)}R_5(x.s)\mathbf{W_5(x.s)}}^{\text{creditInterest()}^5}\cdots\cdots$

Figure 3. Another serial execution.

t_1: $\overbrace{R_1(x.c)R_1(x.s)}^{\text{isLegal()}^1}\overbrace{\overbrace{\mathbf{R_{2,3}(x.c)}W_{2,3}(y.c)}^{\text{setEqChk()}^3}\cdots\cdots\overbrace{\mathbf{R_{2,4}(x.s)}W_{2,4}(y.s)}^{\text{setEqSav()}^4}}^{\text{Account()}^2}\,tr(S1)tr(S2)tr(S3)$ $\{y.c = \mathbf{400}, y.s = \mathbf{707}\}$

t_2: $\cdots\cdots tr(S4)tr(S5)tr(S6)\,\overbrace{R_5(x.c)\mathbf{W_5(x.c)}R_5(x.s)\mathbf{W_5(x.s)}}^{\text{creditInterest()}^5}\cdots\cdots$

Figure 4. A non-serial execution that violates atomic-set serializability.

models crediting interests. The method `setEqChk()` sets the `checking` balance of the current instance to that of the formal parameter. The method `setEqSav()` has similar functionality to that of `setEqChk()`. The remaining methods are setters and getters for the two fields.

```
public class AccountTest {
    Account x, y;
    Thread t1 = new Thread() {
        public void run() {
            if (x.isLegal()) {
                t2.start();
                y = new Account(x);
                S1; S2; S3;  // stmts not accessing x & y
            }
        }
    };
    Thread t2 = new Thread() {
        public void run() {
                S4; S5; S6;     // stmts not accessing x & y
                x.creditInterest(0.01);
        }
    };
    public void testCase() {
        x = new Account(400, 700);
        t1.start();
    }
}
```

Figure 5. A test case for the account system.

The test case (Figure 5) for this example creates an `Account` instance `x`, and starts a thread t_1. Thread t_1 first checks whether `x` is legal or not. If `x` is legal, t_1 starts another thread t_2, creates another `Account` instance `y` from `x`, and then executes a sequence of statements `Si` ($1 \le i \le 3$) that do not access the fields of `x` and `y`. Once t_2 is started, it invokes the method `creditInterest()` on `x` after executing another statement sequence `Si` ($4 \le i \le 6$). Running the test case may reveal a bug in the class `Account`. Developers of this class do not guarantee the second constructor to be executed atomically. Such a bug is common in multithreaded programs and similar bugs are found in the JDK library [31]. Owing to this bug, t_2 can interleave with t_1 when t_1 partially updates the state of `x`. Such interleavings cause the state of `y` to be inconsistent with that of `x`.

Figures 2–4 explain the conditions under which the bug manifests itself. Each line in these figures corresponds to a sequence of memory access events in a thread and time advances from left to right. There are two types of memory access events, namely, read (R) and write (W), which operate on fields `checking` and `savings`. For brevity, let `c` and `s` denote these two fields, respectively. Each method invocation (e.g., `setEqChk()` in Figure 2) is uniquely labeled by an integer (e.g., `setEqChk()`^{3}). Each access event made by a method invocation is labeled by a sequence of integers corresponding to the calling context of that method invocation. For instance, $R_{2,3}(x.c)$ denotes a read event made by `setEqChk()`^{3} in the calling context `Account()`^{2} → `setEqChk()`^{3}. This read event accesses the field `checking` of instance `x`. To simplify the presentation, we do not draw the invocations of setter/getter methods and use the notation *tr*(`Si`) to denote the set of events generated by executing the statement `Si`. The executions in Figure 2 and Figure 3 are *serial* because `Account()`^{2} and `creditInterest()`^{5} execute contiguously without being interrupted. The execution in Figure 4 is *non-serial* because execution of `creditInterest()`^{5} interleaves the execution of `Account()`^{2}. This execution results in an error because the state of `y` after the execution is different from that of either of the two serial executions as shown in these figures.

This error corresponds to an atomic-set serializability violation, which can be characterized by a problematic access pattern [32] "$R_{2,3}(x.c)$ $W_5(x.c)$ $W_5(x.s)$ $R_{2,4}(x.s)$". That is, when an execution exhibits such a pattern, the result of the execution can be different from any serial execution. The chance of hitting such a violation using a randomly chosen schedule is low. It decreases exponentially with the number of statements following $W_{2,4}(y.s)$ in t_1. The runtime monitoring technique [9] detects an atomic-set serializability violation only for the execution being monitored, but does not leverages information of the current execution to increase the chance of finding such a violation subsequent test runs. This approach is unfavorable to hunting concurrency bugs because certain serializability-violating executions seldom occur even under stress testing [23].

2.2 Outline of ASSETFUZZER

Although no violations occur in the execution in Figure 3, we observe that it provides hints on how to find one. The execution contains an access pattern "$W_5(x.c)$ $W_5(x.s)$ $R_{2,3}(x.c)$ $R_{2,4}(x.s)$", which is a permutation of the problematic access pattern "$R_{2,3}(x.c)$ $W_5(x.c)$ $W_5(x.s)$ $R_{2,4}(x.s)$". ASSETFUZZER exploits this information to explore one violating execution.

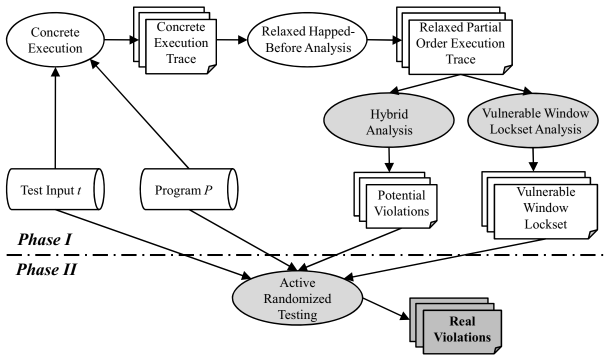

Figure 6. ASSETFUZZER's two-phase testing approach.

As shown in Figure 6, ASSETFUZZER first conducts a relaxed happens-before analysis to derive from the concrete execution (Figure 3) a *relaxed partial order execution trace*, which captures a set of thread interleavings that share the same control flows of the given execution in Phase I. ASSETFUZZER then uses a hybrid analysis to check whether there are linearizations of the relaxed partial order execution trace matching the problematic access pattern. Such an access pattern specifies a subset of executions in a relaxed partial order execution trace. In Phase II, ASSETFUZZER controls a thread scheduler to explore executions in this subset. Focusing only on this subset, *the size of the search space is reduced from 3432 to 30*, which represents the total number of possible thread interleavings and the number of possible thread interleavings that match the pattern, respectively.

However, not all linearizations in this subset are feasible. For instance, `creditInterest()`[5] cannot interleave between $R_{2,3}(x.c)$ and $W_{2,3}(y.c)$ because `t1` and `t2` hold the same lock `x` after entering `setEqChk()`[3] and `creditInterest()`[5]. Exploring such infeasible executions not only wastes time but may also make the whole system reach a deadlock. Such a deadlock due to active thread control is referred to as *thrashing* [11]. ASSETFUZZER conducts a vulnerable window lockset analysis in Phase I, and then uses the computed lockset information to prune these infeasible executions in Phase II. In this way, ASSETFUZZER *effectively reduces the search space to merely four interleavings*, and each of them violates the atomic-set serializability criterion. ASSETFUZZER explores all such interleavings in turn and reports real violations that it comes across in the produced executions. One such violating execution is shown in Figure 4.

ASSETFUZZER discovers the bug by exploring an execution that violates atomic-set serializability. However, this bug is missed by two recent active testing tools RACEFUZZER [27] and ATOMFUZZER [22]. RACEFUZZER uses potential data-race information derived from an imprecise race detection algorithm and generates thread schedules to find real data races. Since all accesses to shared fields are protected by the `synchronized` keywords, there are no data races in this example. RACEFUZZER therefore fails to discover the bug. ATOMFUZZER controls a thread scheduler to find an atomicity-violating locking pattern where a thread acquires and releases a lock while it is inside an atomic block, another thread subsequently acquires and releases the same lock, and then the first thread again acquires the same lock while inside the same atomic block. ATOMFUZZER uses the heuristics that any code block declared as `synchronized` is atomic. For instance, `setEqChk()`[3] in `t1` is inferred as atomic. Since `setEqChk()`[3], `setEqSav()`[4], and `creditInterest()`[5] hold the same lock `x`, they cannot interfere with one another. Therefore, no executions in the example exhibit the locking pattern, and ATOMFUZZER also fails to discover the bug.

3. PRELIMINARIES

This section introduces the background of our work. We assume that each statement in a concurrent thread accesses at most one memory location. This can be achieved by transforming a program into a three-address form. An execution of a statement changes a program from one state to another and generates different types of *events*, including memory access events, lock acquisition events, and lock release events. These events are described in detail in subsequent sections. An execution of a program thus outputs a sequence of events.

3.1 Atomic-Set Serializability

Table 1. Problematic access patterns [32].

	Access Pattern	Description
1	$R_u(m)W_{u'}(m)W_u(m)$	Value read is stale by the time an update is made in *u*.
2	$R_u(m)W_{u'}(m)R_u(m)$	Two reads of the same memory location yield different values in *u*.
3	$W_u(m)R_{u'}(m)W_u(m)$	An intermediate state is observed by *u′*.
4	$W_u(m)W_{u'}(m)R_u(m)$	Value read is not the same as the one last written in *u*.
5	$W_u(m)W_{u'}(m)W_u(m)$	Value written by *u′* is lost.
6	$W_u(m_1)W_{u'}(m)W_{u'}(M–m)W_u(m_2)$	Memory is left in an inconsistent state.
7	$W_u(m_1)W_{u'}(m_2)W_u(m_2)W_{u'}(m_1)$	(same as above)
8	$W_u(m_1)R_{u'}(m)R_{u'}(M–m)W_u(m_2)$	State observed is inconsistent.
9	$R_u(m_1)W_{u'}(m)W_{u'}(M–m)R_u(m_2)$	(same as above)
10	$R_u(m_1)W_{u'}(m_2)R_u(m_2)W_{u'}(m_1)$	(same as above)
11	$W_u(m_1)R_{u'}(m_2)W_u(m_2)R_{u'}(m_1)$	(same as above)

A set of memory locations *M* is grouped into an *atomic set* if there exists a consistency property between the memory locations in *M* [32]. A *unit of work u*, declared on a set of atomic sets, is an event sequence that is expected to preserve the consistency for each declared atomic set. Units of work can be nested. If an event *e* appears in *u* and *u* is not nested in another unit of work, then *e belongs to u*. If *u* is nested in another unit of work, then *e belongs to* the outermost unit of work where *u* is nested. A memory access event belonging to a unit of work *u* is of the form MEM(*m*, *a*, *u*), which indicates that *u* performs an access of type *a* (either R or W) to memory location *m*. For brevity, an event that reads *m* and belongs to *u* is denoted as $R_u(m)$. Similar notation $W_u(m)$ is defined for events of access type W.

A trace is a sequence of events. An execution (trace) *E* is a sequence of events from an actual program execution. Given an execution *E* and an atomic set *M*, the projection of *E* on *M* is a sequence of memory access events such that (1) events in the projection access memory locations in *M*, and (2) for each pair of events, their order in the projection is the same as that in *E*. The atomic sets of an execution *E*, *atomicSets*(*E*), consists of all atomic sets whose elements are accessed by events occurring in *E*, i.e., *atomicSets*(*E*) = {*M* | *m* ∈ *M* and *m* is accessed by an event in *E*}. An execution *E* is *atomic-set serializable* if the projection of *E* on each atomic set in *atomicSets*(*E*) is conflict-serializable [31]; otherwise, the execution is considered *non-serializable*.

An access pattern is a sequence of memory access events, specifying a subset of executions. A trace *matches* an access pattern if a substitution of the units of work and memory locations in the pattern can be found such that the trace contains events in the pattern instance and the order of these events is the same as that of the pattern instance. For example, the trace $\mathbf{R}_{u1}(\mathbf{x})$ $R_{u1}(y)$ $\mathbf{W}_{u2}(\mathbf{x})$ $W_{u2}(y)$ $W_{u1}(z)$ $\mathbf{W}_{u1}(\mathbf{x})$ matches the access pattern "$R_u(m)$ $W_{u'}(m)$ $W_u(m)$" with the substitution $\{u/u1, u'/u2, m/x\}$.

Vaziri et al. [32] identified eleven problematic access patterns (Table 1) such that an execution is atomic-set serializable *if and only if* it does not match any of these patterns. In Table 1, if m denotes one of $M = \{m_1, m_2\}$, the notion $M–m$ denotes the other memory location in M.

4. METHODOLOGY

As shown in Figure 6, ASSETFUZZER consists of two phases. Phase I (Section 4.1) computes a set of potential atomic-set serializability violations from a concrete execution. Phase II (Section 4.2) uses the elements of this set to guide a thread scheduler to explore executions that have real violations. We illustrate our techniques by finding a non-serializable execution that matches the 9th problematic access pattern in Table 1.

4.1 Phase I: Inferring Potential Violations

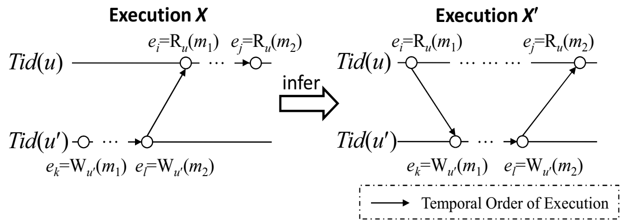

Figure 7. Visual illustration of ASSETFUZZER.

Our first challenge is *how to infer potential violations from a concrete execution exhibiting no serializability violations*. We illustrate how ASSETFUZZER addresses this challenge in Figure 7, which represents two executions X and X'. Each of them is enacted in two threads $Tid(u)$ and $Tid(u')$, where $Tid(u)$ returns the thread that executes u. X is a given violation-free execution, and X' is an inferred execution that violates serializability according to the 9th pattern in Table 1.

4.1.1 Relaxed Happens-Before Analysis

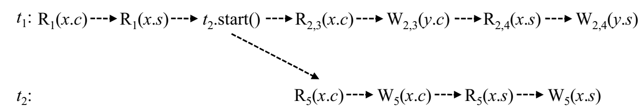

Figure 8. An rPOET of the execution in Figure 3.

Traditionally, the happens-before relation [15] induces a partial order structure, which characterizes a set of Mazurkiewicz-equivalent traces [19]. To address the challenge, we relax the happens-before relation by removing the happens-before relations of memory access events, lock acquisition and release events across threads in a concrete execution. On one hand, this relaxation allows the exploration of more classes of Mazurkiewicz-equivalent traces. On the other hand, checking explicit synchronization constraints can still prune large amounts of infeasible interleavings that do not respect these constraints imposed on a program. In other words, we only need to track explicit synchronization events, such as `start()`, `join()`, `wait()`, and `notify()` in a Java program execution. The relaxed happens-before relation (denoted by $\tilde{\rightarrow}$) is, therefore, a subset of the traditional happens-before relation. The resulting structure induced by a relaxed happens-before relation is called a *relaxed partial order execution trace* (*rPOET*).

An rPOET of a concrete execution facilitates the exploration of those thread interleavings that respect the control flows of the concrete execution [30]. An example rPOET of the execution in Figure 3 is shown in Figure 8, where events tr(`Si`) irrelevant to serializability violations are omitted. Basically, if one linearization of an rPOET matches a problematic access pattern, the program may violate the atomic-set serializability in those thread interleavings that share the same rPOET. Such detection can be achieved by checking the relaxed happens-before relation between events in the rPOET. For instance, if the four events e_k, e_l, e_i, and e_j of execution X can be reordered to become X' in Figure 7, they must satisfy the following formula F_{RHB}:

$$F_{RHB}: (e_i \tilde{\rightarrow} e_j) \wedge (e_k \tilde{\rightarrow} e_l) \wedge \neg(e_k \tilde{\rightarrow} e_i) \wedge \neg(e_j \tilde{\rightarrow} e_l).$$

4.1.2 Lockset Analysis

The relaxed happens-before analysis is useful to infer potential violations, but it can be imprecise and generate many false positives. This is because the relaxed happens-before analysis does not consider the following locking discipline: Each atomic set is protected by a common lockset, in the sense that every unit of work holds the lockset when it accesses the atomic set. For instance, execution X' of Figure 7 is infeasible if the two units of work u and u' hold the same lock during their entire execution periods. Our second challenge is thus on *how to prune infeasible executions that do not comply with the locking discipline*. To eliminate such false positives deduced by the relaxed happens-before analysis, we conduct a lockset analysis.

Determining whether an execution comply with this locking discipline requires computing the set of locks held by a thread at any given point. Given a lock l and a thread t, let ACQ(l, t) and REL(l, t) denote a lock acquisition event and a lock release event, respectively. Let events be uniquely labeled by their indices in an ascending order according to their order of occurrences. The locks held by thread t before the occurrence of event e_i in an execution can be given by:

$$L_t(e_i) = \{l \mid \exists x\, (x < i \wedge e_x = \mathrm{ACQ}(l, t)) \wedge \forall y\, (x < y < i \wedge e_y \neq \mathrm{REL}(l, t))\}.$$

If two units of work do not comply with this locking discipline, consistent accesses to memory locations in the same atomic set cannot be guaranteed. For instance, if the units of work u and u' in execution X of Figure 7 do not comply with the locking discipline, it is possible to reorder e_k and e_l so that they interleave between e_i and e_j as depicted by execution X'. Non-compliance can occur in two situations. First, units of work u and u' do not hold any common locks during their entire execution periods. Second, units of work u and u' hold common locks before the occurrence of e_i, e_j, e_k, and e_l, but u releases the common locks between e_i and e_j. The lock releases create a window for events e_k and e_l to occur between e_i and e_j. To eliminate these false positives algorithmically, we formalize the two situations as F_{LS}:

$$F_{LS}: (\exists e_x = \mathrm{REL}(lk, t) \wedge (i < x < j) \wedge (lk \in LS) \wedge (t = Tid(u)) \vee (LS = \emptyset),$$
$$\text{where } LS = L_{Tid(u)}(e_i) \cap L_{Tid(u)}(e_j) \cap L_{Tid(u')}(e_k) \cap L_{Tid(u')}(e_l).$$

4.1.3 Hybrid Analysis

The relaxed happens-before analysis and the lockset analysis capture different synchronization constraints imposed on a program. The use of lockset analysis alone can, therefore, also be imprecise and generate false positives. For illustration, let us consider the execution in Figure 3. The method `isLegal()`[1], which $R_1(x.c)$ and $R_1(x.s)$ belong to, does not hold any locks during its execution. According to the lockset analysis, the events $W_5(x.c)$ and $W_5(x.s)$ in `t₂` can be shifted earlier so that they occur between $R_1(x.c)$ and $R_1(x.s)$. If so, the resultant execution is non-serializable because it matches the 9th problematic access pattern. However, this execution is infeasible because `t₂` is started after `isLegal()`[1]. Fortunately, such false positives can be eliminated by the relaxed happens-before analysis. This suggests that we need to combine both the relaxed happens-before analysis and the lockset analysis by forming a logical conjunction of the two preceding formulae F_{RHB} and F_{LS}. We refer to such an analysis based on the conjunction of F_{RHB} and F_{LS} as *hybrid analysis*.

To illustrate the hybrid analysis, let us consider events $W_5(x.c)$, $W_5(x.s)$, $R_{2,3}(x.c)$, and $R_{2,4}(x.s)$ in Figure 3. $W_5(x.c)$ and $W_5(x.s)$ belong to the unit of work `creditInterest()`[5]; $R_{2,3}(x.c)$ and $R_{2,4}(x.s)$ belong to the unit of work `Account()`[2]. They access the same atomic set {*x.c*, *x.s*}. In the derived rPOET (Figure 8), $W_5(x.c)$ and $W_5(x.s)$ are concurrent to $R_{2,3}(x.c)$ and $R_{2,4}(x.s)$. Although threads `t₁` and `t₂` hold the common lockset {`x`} before the occurrence of these events, `t₁` releases `x` between $R_{2,3}(x.c)$ and $R_{2,4}(x.s)$ when `setEqChk()`[3] exits. The hybrid analysis therefore reports a potential violation with respect to the problematic access pattern "$R_{2,3}(x.c)$ $W_5(x.c)$ $W_5(x.s)$ $R_{2,4}(x.s)$".

4.2 Phase II: Detecting Real Violations

The objective of Phase II is to explore executions that refine the potential violations detected by the hybrid analysis. We achieve this by exploiting the potential violation information to direct a thread scheduler. This task further poses two challenges. First, *how to match events between the two phases so that we can suspend threads when relevant events occur*? Second, *how to choose proper suspension points in threads so as to alleviate the thrashing problem*?

4.2.1 Matching Contexts

Although the concept of finding violating executions by manipulating a thread scheduler has been proposed by RACEFUZZER [27], our thread manipulation technique needs to handle additional difficulties. The thread manipulation of RACEFUZZER is achieved by bringing two racing statements next to each other without considering the calling contexts of these statements. However, this mechanism is inadequate for exploring executions that violate atomic-set serializability. This is because finding such non-serializable executions requires identifying the units of work whose access events match some problematic access patterns. If we cannot map an access event in Phase I to the corresponding event in Phase II, the executions found in Phase II may not exhibit the intended violations derived in Phase I. For instance, suppose that the hybrid analysis in Phase I infers from the execution in Figure 3 that the program can potentially match the problematic access pattern "$R_{2,3}(x.c)$ $W_5(x.c)$ $W_5(x.s)$ $R_{2,4}(x.s)$". The events in this pattern are generated by executing statements `L1`, `L3`, `L4`, and `L2`, respectively. In Phase II, if we simply find an execution that matches this statement sequence, the execution may not correctly instantiate the inferred violation. Consider the serial execution in Figure 2, the events $R_{2,3}(x.c)$, $W_{2,3}(y.c)$, $W_{2,4}(y.s)$, and $R_5(x.s)$ are also generated by `L1`, `L3`, `L4`, and `L2`, respectively. However, this execution does not violate atomic-set serializability. In other words, the calling contexts of the events in a potential violation in Phase I need to match those of the concrete executions identified in Phase II. We use an MD5 hash algorithm to codify calling contexts [3] and match an access event in Phase I with another in Phase II if both their statement labels and their codified calling contexts are equal. This approach alleviates the tracing costs and speeds up the matching process.

4.2.2 Alleviating Thrashing

Event reordering requires us to carefully suspend and resume thread executions. RACEFUZZER [27], as well as some other testing tools (e.g., ConTest [2, 4]), suspends a thread whenever a suspicious access event occurs. This mechanism presents a problem for detecting atomic-set serializability violations.

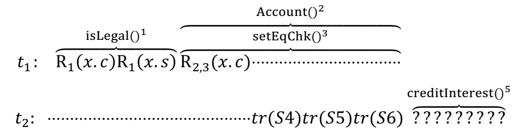

Figure 9. A problematic thread manipulation.

For instance, the hybrid analysis reports that a violation occurs if events $W_5(x.c)$, $W_5(x.s)$, $R_{2,3}(x.c)$, and $R_{2,4}(x.s)$ in Figure 3 can be reordered so that an execution of the program matches the problematic access pattern "$R_{2,3}(x.c)$ $W_5(x.c)$ $W_5(x.s)$ $R_{2,4}(x.s)$". Now, let us consider a thread manipulation in Figure 9. The thread execution is switched from `t₁` to `t₂` at the point after executing $R_{2,3}(x.c)$. The switching attempts to make the execution match the pattern. Unfortunately, the thread scheduler could not fire the target event $W_5(x.c)$ after thread `t₂` executes `S4`, `S5`, and `S6`. This is because the lock `x` required to enter `creditInterest()`[5] is still being held by thread `t₁`. Consequently, the thread scheduler could not make any progress. Such a deadlock due to active thread manipulation is referred to as *thrashing* [11]. RACEFUZZER uses a daemon thread to break the tie by randomly selecting a thread that is suspended by the testing engine to execute (`t₁` in this example). Although this mechanism resolves thrashing in this case, it increases testing time because the thread scheduler needs to wait for some timeouts before breaking the tie. The mechanism may also miss some non-serializable executions. For instance, the resumption of `t₁` may cause `t₁` to execute $R_{2,4}(x.s)$ before `t₂` executes $W_5(x.c)$. As a result, an execution refining the problematic access pattern is missed. Missing such executions reduces the effectiveness of testing techniques to detect atomic-set serializability violations.

We address the thrashing problem by conducting a lockset analysis. If the hybrid analysis reports a potential violation that involves events e_i and e_j belonging to unit of work *u*, and events e_k and e_l belonging to unit of work *u'* (c.f., execution *X* in Figure 7), a *vulnerable window lockset* is computed as follows.

$$VWLS = \{lk \mid lk \in LS \text{ and } \exists e_x = \mathrm{REL}(lk,t) \wedge (i < x < j) \wedge t = Tid(u), \\ \text{where } LS = L_{Tid(u)}(e_i) \cap L_{Tid(u)}(e_j) \cap L_{Tid(u')}(e_k) \cap L_{Tid(u')}(e_l)\}$$

The vulnerable window lockset *VWLS* contains locks that are released between e_i and e_j and these locks belong to the common lockset *LS* held by threads when these four events occur. Intuitively, when thread *Tid*(*u*) releases locks in *VWLS*, it creates a window for events e_k and e_l to occur between e_i and e_j. Such interleaving causes an atomic-set serializability violation. We discuss how to utilize the vulnerable window lockset to choose proper context switching points for alleviating the thrashing problem in the next section.

4.2.3 Description of Thread Manipulation Algorithm

In this section, we informally describe the thread manipulation algorithm for detecting atomic-set serializability violations. Details of this algorithm can be found in our technical report [14]. Given a violation-free execution X in Figure 7, if the hybrid analysis infers that events e_i and e_j belonging to unit of work u, and events e_k and e_l belonging to unit of work u' can be reordered to match the problematic access pattern "$e_i\ e_k\ e_l\ e_j$", the thread manipulation algorithm explores a concrete execution (c.f., execution X' in Figure 7) which matches the pattern. The algorithm accepts two inputs from the hybrid analysis: a sequence of statement label and codified calling context pairs that generate events e_i, e_j, e_k, and e_l, and a vulnerable window lockset *VWLS*.

The algorithm runs iteratively. In each iteration, it randomly selects an enabled thread to execute. If the next event does not match any of the events e_i, e_j, e_k, and e_l, the algorithm simply executes the event. The algorithm identifies events e_i, e_j, e_k, and e_l by their statement labels and codified calling contexts from the hybrid analysis. In order to find an execution that matches the pattern "$e_i\ e_k\ e_l\ e_j$", the algorithm manipulates a randomized schedule as follows. It suspends thread *Tid*(u) and resumes thread *Tid*(u') after executing event e_i. Similarly, it suspends thread *Tid*(u') and resumes thread *Tid*(u) after executing event e_l.

However, the points for thread suspension and resumption must be chosen judiciously to avoid creating thrashing. As discussed in the preceding section, if *Tid*(u) holds locks in *VWLS* after executing e_i, thrashing can occur if the algorithm switches thread execution from *Tid*(u) to *Tid*(u') at that point. To avoid thrashing in this case, the algorithm chooses a switching point according to *VWLS*. If *VWLS* is empty, the algorithm suspends *Tid*(u) immediately after executing e_i. This operation is safe because *Tid*(u') does not require locks held by *Tid*(u). If *VWLS* is non-empty, the algorithm performs the thread suspension and resumption after *Tid*(u) releases locks in *VWLS*. This mechanism prevents thrashing because *Tid*(u) has released the common locks that *Tid*(u') needs to acquire before event e_k can occur. Thrashing may also occur when the algorithm needs to switch thread execution from *Tid*(u') to *Tid*(u) after executing event e_l. Based on *VWLS*, a similar thread manipulation mechanism is employed in that case. In the other cases, when the algorithm encounters thrashing, it relies on the default mechanism of RACEFUZZER [27] to resolve the problem: At any point of the execution, if the program gets into a deadlock due to thread manipulation, the algorithm randomly selects a suspended thread to break the tie.

When the algorithm encounters event e_j, it actually detects a real atomic-set serializability violation. At that point, it executes the event, reports the violation, and resumes *Tid*(u') if it is suspended. After that, the algorithm runs the program using a randomized schedule, expecting to catch some program failures, such as uncaught exceptions, due to atomic-set serializability violations. Note that if event e_k occurs before event e_i, the algorithm can suspend thread *Tid*(u') instead of executing e_k to create the correct event order for exploring violating executions.

5. EVALUATION

We implemented ASSETFUZZER on top of the testing framework CALFUZZER [10]. We instrumented Java bytecode to monitor events and control thread schedules. The instrumentation adds additional methods to support the hybrid analysis and the thread manipulation algorithm. We used the heuristics similar to [9] to infer atomic sets and units of work. Specifically, we assume that all non-`final` instance fields of a class and those of its super classes form a per-instance atomic set. All instance methods of that class and those of its super classes are considered the initial units of work declared on these per-instance atomic sets. All non-`final static` fields of a class form a per-class atomic set. All methods of that class are considered the initial units of work for this atomic set. We instrumented method entry and exit points to keep track of dynamic call graphs, which are used to determine what unit of work each access event belongs to. A dynamic call graph is essentially the stack traces of the methods visited by a thread. An access event in an atomic set belongs to the outermost unit of work declared on that atomic set.

We conducted experiments to study the effectiveness of ASSETFUZZER in detecting atomic-set serializability violations. The experiments also studied ASSETFUZZER's ability in revealing failures and in discovering concurrency bugs.

5.1 Benchmarking Subjects

We evaluated ASSETFUZZER using 13 Java multithreaded subjects, which have been recently used to benchmark concurrency defect analysis techniques [6, 22, 27, 31]. The first six subjects are open libraries from Sun's JDK 1.4.2 and the last seven subjects are closed programs. These subjects include `jigsaw 2.2.6`, which is W3C's leading-edge web server platform with 381,348 lines of code.

5.2 Experimental Setup

We compare ASSETFUZZER (*AsF*) with several other testing strategies. The first strategy (*RM*) combines a runtime monitoring technique [9] with a randomized scheduler [28] to detect atomic-set serializability violations. The second strategy RACEFUZZER (*RF*) uses a biased randomized scheduler to find real data races [27]. The third strategy ATOMFUZZER (*AtF*) controls a randomized thread scheduler to detect an atomicity-violating locking pattern [22]. We used the implementation of RACEFUZZER available from the CALFUZZER repository [10]. We implemented ATOMFUZZER, the runtime monitoring technique, and the randomized scheduler [28] because they are not publicly available. For each subject, we use 10 profiling runs under Phase I of *AsF* to build an initial set of potential violations [23]. We chose 10 profiling runs because we observed that for most subjects the potential violations inferred from 10 profiling runs are almost the same as those inferred from 5 profiling runs. For each inferred violation in the set, we ran Phase II of *AsF* 100 times to estimate its effectiveness following [27]. To compare the effectiveness of *AsF* with that of *RM*, we ran *RM* the same total number of times as *AsF* for each subject. To study bug detection ability, we ran *RF* in the same manner as *AsF*. Since *AtF* has only one phase, we ran *AtF* the maximum number of times used by *RF* and *AsF* for each subject.

5.3 Experimental Results

Table 2 summarizes the results of the experiments. Column 2 reports the total lines of code for each subject. The column headed "Average Runtime" reports the average runtime of normal executions without employing any testing strategies, as well as the average runtime of executions employing each of the four testing strategies *RM*, *RF*, *AtF*, and *AsF*, respectively. For *RF* and *AsF*, the time of the profiling runs in Phase I is also included to

Table 2. Experimental results.

Programs		LOC	Average Runtime (sec)					# of Violations (real)		$r_{violation}$		# of Exceptions			# of Bugs		
			Normal	*RM*	*RF*	*AtF*	*AsF*	*RM*	*AsF*	*RM*	*AsF*	*RF*	*AtF*	*AsF*	*RF*	*AtF*	*AsF*
1	StringBuffer	1,320	0.332	0.883	-	0.285	0.797	1	1	0.200	0.980	0	1	1	0	1	1
2	ArrayList	5,866	0.326	1.166	0.558	0.335	0.856	3	17	0.060	0.350	2	1	5	6	2	5
3	LinkedList	5,979	0.325	1.088	0.545	0.335	0.861	5	27	0.200	0.534	2	1	4	6	2	9
4	HashSet	7,086	0.362	1.012	0.584	0.364	0.978	11	83	0.290	0.347	3	2	4	6	3	7
5	TreeSet	7,532	0.329	1.219	0.545	0.342	0.995	0	16	0	0.150	3	2	5	3	5	7
6	LinkedHashSet	12,926	0.332	1.294	0.518	0.330	0.876	3	4	0.130	0.110	4	3	4	5	4	3
7	moldyn	1,352	0.345	13.474	13.576	0.887	1.797	3	25	1.000	1.000	0	0	0	0	0	0
8	raytracer	1,924	0.166	24.826	1.555	2.206	0.830	9	28	0.580	0.998	0	0	0	1	0	1
9	montecarlo	3,619	0.219	4.606	0.944	0.718	2.541	4	80	1.000	0.990	0	0	0	0	0	0
10	cache4j	3,897	1.382	3.413	4.918	2.304	2.013	1	56	0.100	0.933	0	0	0	2	0	2
11	hedc	29,949	1.030	1.830	3.691	1.060	1.796	0	5	0	0.976	0	0	0	1	0	3
12	weblech	35,175	1.236	8.729	3.556	8.694	5.574	4	16	0.088	0.261	1	1	1	3	0	4
13	jigsaw	381,348	5.933	> 3600	12.485	93.277	16.746	-	116	-	0.853	0	0	2	8	0	12

compute the average. *RM* has the highest overhead. For the `jigsaw` subject, *RM* could not finish within one hour for each test run and was terminated to let the experiments complete in a reasonable time. For the other 12 subjects, its slowdown factors range from 1.77x-149.55x. The slowdown factor is given by the ratio of the average runtime incurred by *RM* over that of normal executions. The overhead of *RM* is high because the runtime monitoring algorithm [9] needs to check each pair of units of work, each pair of memory locations, and each pattern for every execution. The overhead of *AsF* is much lighter than *RM*, with slowdown factors ranging from 1.45x-11.60x. This is because *AsF* needs to perform the checking only for the profiling runs in Phase I. In Phase II, it just monitors memory access events and suspends threads upon an occurrence of the events relevant to a potential violation. The slowdown factors of *RF* and *AtF* range from 1.56x-39.35x and 0.86x-15.72x, respectively. Generally, *AtF* has low overhead because it just monitors lock acquisition and release events. The slowdown factor of *AtF* for the `StringBuffer` subject is less than one because *AtF* causes the subject to raise uncaught exceptions and the subject can terminate earlier than normal executions. The overheads of *AtF* for the `raytracer` and `jigsaw` subjects are quite high because *AtF* encounters thrashing problems. Additional time is required to resolve these thrashings.

The column headed "# of Violations" reports the number of real atomic-set serializability violations detected by *RM* and *AsF*. In the experiments, *AsF* is able to detect ten times more real violations than *RM*. *AsF* finds 116 real violations in the `jigsaw` subject while *RM* cannot finish and report any violations within one hour. *RM* cannot detect any violations in the subjects `TreeSet` and `hedc` because violating executions rarely occur in these two subjects under randomized schedules. We measured the effectiveness of *AsF* (*RM*) in detecting atomic-set serializability violations by *detection rate*, which is given by the ratio of the number of test runs that *AsF* (*RM*) detects any violation over the total number of test runs. This metrics is similar to fault detection rate [7], which is used to measure the effectiveness of test selection strategies. The column headed "$r_{violation}$" reports the detection rates of *RM* and *AsF*. The average detection rates of *RM* and *AsF* are 0.30 and 0.65, respectively. *RM* has high detection rates in the subjects `moldyn` and `montecarlo` because these two subjects contain customized synchronization primitives that exhibit intentional races [29]. The violations in these two subjects are therefore considered benign. If we exclude the results of these two subjects, the average detection rate of *RM* is about 0.16. *AsF* increases the detection rate to 0.59, excluding the two subjects. These results show that *AsF* is more effective in detecting atomic-set serializability violations than randomized testing.

The column headed "# of Exceptions" reports the number of distinct uncaught exceptions detected by *RF*, *AtF*, and *AsF*. On average, *AsF* detects the largest number of uncaught exceptions. For each subject, the number of uncaught exceptions detected by *AsF* is greater than or equal to those of *RF* and *AtF*. In the `jigsaw` subject, *AsF* detects two previously unknown uncaught exceptions of type `NullPointerException` in the `httpd` class. These exceptions are missed by *RF* and *AtF*. We made an aggregation of all the uncaught exceptions detected by *RF*, *AtF*, and *AsF* for each subject. We observed that *AsF* does not miss any uncaught exceptions with respect to these aggregations.

We categorized the testing reports from *RF*, *AtF*, and *AsF* according to the problematic memory locations identified. We classified all access to shared memory locations without proper synchronizations as a malign bug [9]. For violations that are caused by the same bug, we only report the problem once. We aggregated all the distinct bugs discovered by *RF*, *AtF*, and *AsF* for each subject. The column headed "# of Bugs" reports the number of bugs detected by *RF*, *AtF*, and *AsF* in these aggregations. Overall, *AsF* detects the largest number of bugs except for the subjects `ArrayList` and `LinkedHashSet`. It does not miss any bugs in the closed subjects with respect to the aggregations. *AtF* detects the least number of bugs and cannot detect any bugs in the closed subjects. The number of bugs detected by *RF* is ranged between those of *AtF* and *AsF*.

5.4 Discussion

ATOMFUZZER misses the largest number of bugs. One major reason is that many concurrency bugs exhibit as asymmetric races [25], which occur when some well-behaved threads consistently access a shared memory location with a lock while some ill-behaved threads improperly access the memory location due to synchronization errors, such as forgetting to use a lock. These bugs do not have the atomicity-violating locking pattern that ATOMFUZZER looks for. ATOMFUZZER misses such bugs in subjects `raytracer`, `weblech`, etc. ASSETFUZZER detects these bugs because they exhibit improper access sequences that can be captured by the problematic access patterns in Table 1.

AssetFuzzer misses some bugs in the open libraries. We used the test drivers from the repository [10] and they randomly invoke methods of these subjects and may not exercise all buggy code in the profiling runs. The hybrid analysis thus cannot infer potential violations in Phase I. Owning to this, AssetFuzzer misses some bugs in the subjects `ArrayList`, `HashSet`, `TreeSet`, and `LinkedHashSet`. For the same reason, RaceFuzzer misses several bugs in all the open subjects. Being dynamic in nature, AssetFuzzer cannot detect all atomic-set serializability violations in a multithreaded program. It detects a real violation if the violation can be produced with the given test harness for some thread schedules. This can be alleviated by combining AssetFuzzer with techniques like stateless model checking [30] to explore more thread interleavings.

Programs that are free from low-level data races can contain high-level data races [1], which do not guarantee two correlated memory locations to be accessed atomically (e.g., the bug in Section 2). RaceFuzzer fails to detect such bugs in the subjects `StringBuffer`, `hedc` and `weblech`, but these bugs can be detected by AssetFuzzer because atomic-set serializability violations subsume high-level data races [9]. AtomFuzzer also misses these bugs in the subjects `hedc` and `weblech` because it cannot find atomicity-violating locking patterns in these subjects.

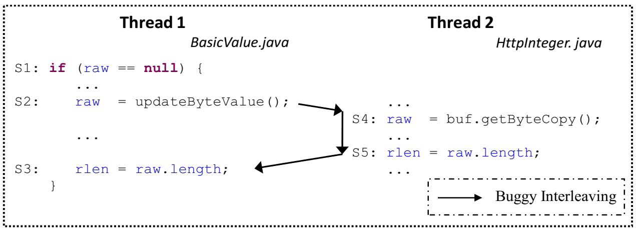

Figure 10. A high-level data race in the `jigsaw` subject.

Even though some high-level data races are caused by low-level data races, AssetFuzzer provides useful information to diagnose the bugs. For instance, consider a high-level data race in the `jigsaw` subject (Figure 10). The variable `raw` is a shared buffer and the variable `rlen` indicates the length of the buffer. These two variables are correlated, but the subject fails to let them be updated atomically. The bug manifests itself when a thread writes to `raw`, and then another thread overwrites `raw` and updates the variable `rlen`, yet the first thread sets `rlen` according to the depleted contents of the original buffer. In this scenario, the correlated variables `raw` and `rlen` are not consistent. Although RaceFuzzer reports that `S2` and `S4`, as well as `S3` and `S5`, are in race conditions, separately reporting them may not be useful for developers to effectively diagnose the bug. On the contrary, AssetFuzzer directly reports that the program does not guarantee these two variables to be accessed atomically.

6. RELATED WORK

Researchers have proposed various techniques to detect data races [21, 24, 26]. However, a large fraction of data races may not reflect real concurrency bugs [29]. Recently, lots of research efforts have focused on atomicity/serializability violations [5, 6, 31]. However, the underlying criteria of these techniques do not respect correlations between shared variables and thus can cause these techniques to miss detecting bugs or reporting false warnings. To address this problem, Vaziri et al. [32] proposed a new criterion, called atomic-set serializability. This criterion characterizes a wider range of concurrency bugs. Not only do data races [26], high-level data races [1], and standard serializability violations [31] fall into atomic-set serializability violations, but violations of access invariants [16, 18] mined from machine learning algorithms can also be captured by the problematic access patterns (Table 1). Meanwhile, this criterion is accurate in discerning real concurrency bugs [9]. AssetFuzzer is based on atomic-set serializability and is thus more effective and accurate in detecting concurrent bugs.

Kidd et al. [13] proposed to verify atomic-set serializability by model checking techniques. Although this approach is able to explore all executions, it is not scalable even for medium-sized programs [12]. Hammer et al. [9] proposed a runtime monitoring technique to detect atomic-set serializability violations. However, this approach may not be able to effectively detect violations without cautiously controlling thread schedules. AssetFuzzer increases the rate of detecting atomic-set serializability violations by directing a thread scheduler using information collected from a hybrid analysis.

Randomized testing techniques [4, 28] have been proposed for multithreaded programs. These techniques randomly seed `sleep()`, `yield()`, and `priority()` primitives in Java programs. However, these primitives can only advise a scheduler to make a thread switch but cannot accurately force a thread switch. Our results show that AssetFuzzer is more effective in detecting atomic-set serializability violations than a randomized testing strategy. Active randomized testing has recently been proposed as a promising technique to detect concurrency defects. RaceFuzzer [27] controls a thread scheduler to create real data races based on potential race conditions derived by an imprecise race detection algorithm [21]. RaceFuzzer cannot detect some atomicity violations that are not caused by data races [31]. AtomFuzzer [22] is another active testing system that detects a special class of causal atomicity [5] violations, characterized by an atomicity-violating locking pattern. However, some atomicity violations, such as the one in Section 2, do not match the locking pattern. On the contrary, the underlying atomic-set serializability criteria of AssetFuzzer enable it to detect a wider range of concurrency bugs including malign data races and atomicity violations.

CHESS [20] utilizes a context-bounded search strategy for systematic testing of multithreaded programs. To apply CHESS to detecting atomic-set serializability violations, CHESS needs to combine with a dynamic detector [9] which can have high runtime overhead. CTrigger [23] is a two-phase testing tool for detecting atomicity violations that involve one shared variable. AssetFuzzer does not have such a restriction. CTrigger uses heuristics to estimate how long to delay a thread, whereas AssetFuzzer directly identifies proper switching points for thread manipulation.

7. CONCLUSION

We have proposed AssetFuzzer, a two-phase testing technique to detect atomic-set serializability violations. AssetFuzzer uses a hybrid analysis technique to infer potential violations from a concrete execution and controls a randomized thread scheduler to detect real violations. If atomic sets and units of work are correctly specified, AssetFuzzer gives no false positives because it actually brings out executions that match one of the problematic access patterns in Table 1. The experimental results on a number of Java subjects show that AssetFuzzer effectively detects more atomic-set serializability violations than a runtime monitoring technique running on top of a randomized scheduler. The results also show that ASSETFUZZER detects more distinct failures and concurrency bugs than two recent active testing tools RACEFUZZER and ATOMFUZZER. In the future, we plan to study how to synergize ASSETFUZZER and stateless model checker (e.g., [30]) to increase ASSETFUZZER's ability to detect atomic-set serializability violations.

8. ACKNOWLEDGMENT

We thank the ICSE reviewers for their insightful and constructive comments. We thank Koushik Sen and Chang-Seo Park for their help on setting up the `jigsaw` subject. We thank Chang Xu for proofreading a previous version of this paper.

9. REFERENCES

[1] Artho, C., Havelund, K., and Biere, A. 2003. High-level data races. *Softw. Test. Verif. Reliab.* 13, 4 (Nov. 2003), 207–227.

[2] Ben-Asher, Y., Eytani, Y., Farchi, E., and Ur, S. 2006. Noise makers need to know where to be silent - producing schedules that find bugs. In *Proc. ISOLA* '06, 458-465.

[3] Boonstoppel, P., Cadar, C., and Engler, D. 2008. RWset: Attacking path explosion in constraint-based test generation. In *Proc. TACAS* '08, 351-366.

[4] Edelstein, O., Farchi, E., Nir, Y., Ratsaby, G., Ur, S. 2002. Multithreaded Java program test generation. *IBM Syst J.* 41, 1, 111-125.

[5] Farzan, A. and Madhusudan, P. 2006. Causal atomicity. In *Proc. CAV* '06, 315-328.

[6] Flanagan, C. and Freund, S. N. 2004. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proc. POPL* '04, 256-267.

[7] Frankl, P. G. and Weiss, S. N. 1993. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Trans. Softw. Eng.* 19, 8 (Aug. 1993), 774-787.

[8] Godefroid, P. and Nagappan, N. 2008. Concurrency at Microsoft: An exploratory survey. *Technical Report* MSR-TR-2008-75, May 2008.

[9] Hammer, C., Dolby, J., Vaziri, M., and Tip, F. 2008. Dynamic detection of atomic-set-serializability violations. In *Proc. ICSE* '08, 231-240.

[10] Joshi, P., Naik, M., Park, C., and Sen, K. 2009. CALFUZZER: An extensible active testing framework for concurrent programs. In *Proc. CAV* '09, 675-681.

[11] Joshi, P., Park, C., Sen, K., and Naik, M. 2009. A randomized dynamic program analysis technique for detecting real deadlocks. In *Proc. PLDI* '09, 110-120.

[12] Kidd, N., Reps, T., Dolby, J., and Vaziri, M. 2007. Static detection of atomic-set-serializability violations. *Technical Report* #1623, University of Wisconsin-Madison, Oct. 2007.

[13] Kidd, N., Reps, T., Dolby, J., and Vaziri, M. 2009. Finding concurrency-related bugs using random isolation. In *Proc. VMCAI* '09, 198-213.

[14] Lai, Z., Cheung, S.C., and Chan, W.K. 2009. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. *Technical Report* HKUST-CS09-07, September 2009.

[15] Lamport, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (Jul. 1978), 558-565.

[16] Lu, S., Park, S., Hu, C., Ma, X., Jiang, W., Li, Z., Popa, R. A., and Zhou, Y. 2007. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proc. SOSP* '07, 103-116.

[17] Lu, S., Park, S., Seo, E., and Zhou, Y. 2008. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proc. ASPLOS* '08, 329-339.

[18] Lu, S., Tucek, J., Qin, F., and Zhou, Y. 2006. AVIO: Detecting atomicity violations via access interleaving invariants. In *Proc. ASPLOS* '06, 37-48.

[19] Mazurkiewicz, A. 1987. Trace theory. In *Advances in Petri Nets*, 279-324.

[20] Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P., and Neamtiu, I. 2008. Finding and reproducing Heisenbugs in concurrent programs. In *Proc. OSDI* '08, 267-280.

[21] O'Callahan, R. and Choi, J. 2003. Hybrid dynamic data race detection. In *Proc. PPoPP* '03, 167-178.

[22] Park, C. and Sen, K. 2008. Randomized active atomicity violation detection in concurrent programs. In *Proc. SIGSOFT* '08/*FSE*-16, 135-145.

[23] Park, S., Lu, S., and Zhou. Y. CTrigger: Exposing atomicity violation bugs from their hiding places. In *Proc. ASPLOS* '09, 25-36.

[24] Perkovic, D. and Keleher, P. J. 1996. Online data-race detection via coherency guarantees. In *Proc. OSDI* '96, 47-57.

[25] Ratanaworabhan, P., Burtscher, M., Kirovski, D., Zorn, B., Nagpal, R., and Pattabiraman, K. 2009. Detecting and tolerating asymmetric races. In *Proc. PPoPP* '09, 173-184.

[26] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.* 15, 4 (Nov. 1997), 391-411.

[27] Sen, K. 2008. Race directed random testing of concurrent programs. In *Proc. PLDI* '08, 11-21.

[28] Stoller, S. D. 2002. Testing concurrent Java programs using randomized scheduling. In *Proc. RV* '02, 142-157.

[29] Tian, C., Nagarajan, V., Gupta, R., and Tallam, S. 2008. Dynamic recognition of synchronization operations for improved data race detection. In *Proc. ISSTA* '08, 143-154.

[30] Wang, C., Chaudhuri, S., Gupta, A., and Yang, Y. 2009. Symbolic pruning of concurrent program executions. In *Proc. ESEC/FSE* '09, 23-32.

[31] Wang, L. and Stoller, S. D. 2006. Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Softw. Eng.* 32, 2 (Feb. 2006), 93-110.

[32] Vaziri, M., Tip, F., and Dolby, J. 2006. Associating synchronization constraints with data in an object-oriented language. In *Proc. POPL* '06, 334-345.