Cost-Effective Partitioning for Detection of Atomicity Violations: An Empirical Study
Shangru Wu, Chunbai Yang, Changjiang Jia†, and W.K. Chan

Abstract—Dynamic concurrency bug detectors predict and then examine suspicious instances of atomicity violations from executions of multithreaded programs. Only few predicted instances are real bugs. Prioritizing such instances can make the examinations cost-effective, but is there any design factor exhibiting significant influence? This work presents the first controlled experiment that studies two design factors, abstraction level and subspace, in partitioning such instances through 35 resultant partition-based techniques on 10 benchmarks with known vulnerability-related bugs. The empirical analysis reveals significant findings. First, partition-based prioritization can significantly improve the fault detection rate. Second, coarse-grained techniques are more effective than fine-grained ones, and using some 1-dimensional subspaces is more effective than using other dimensional subspaces. Third, eight previously unknown techniques can be more effective than the technique modeled after a state-of-the-art dynamic detector.

Index Terms— Atomicity violations, concurrency, debugging, abstraction, partitioning, multithreaded programs, vulnerability

1 INTRODUCTION

Many multithreaded programs contain concurrency bugs. A survey [16] reports that 49% of all studied concurrency bugs in large-scale multithreaded programs are atomicity violations [16]. Between single-variable and multi-variable atomicity violations, most existing techniques [17][25] study the single-variable ones, which are also the focus of this paper. An atomicity violation bug pattern on a single memory location models the scenario where two consecutive accesses, say, \((e_p, e_c)\), to a memory location generated by one thread \(t_1\) in an execution is interfered by an access, say, \(e_i\), to the same memory location generated by some other thread \(t_2\) in between \(e_p\) and \(e_c\) in the same execution.

Static detectors (e.g., [21]) can detect atomicity violation bugs. However, they are not scalable enough to precisely analyze large-scale programs (e.g., MySQL).

Many dynamic detectors [14][20][29] formulate a two-phase strategy to detect atomicity violations. Each detector observes an access sequence \((e_p, e_c)\) or \((e_r, e_p, e_c)\) from an execution trace, and then predicts whether the permutated sequence \((e_r, e_p, e_c)\) is suspicious based on the happens-before relations [17]. To examine such a suspicious instance, the detector schedules a follow-up execution with an attempt to trigger \((e_p, e_r, e_c)\) and then checks any failure produced. Such a two-phase strategy is sound by only reporting cases that produce program failures. But, a large number of suspicious instances may be predicted, which results in a long period of time to examine all these suspicious cases. For instance, in our experiment, 8000+ suspicious cases were identified on MySQL. The examinations on all these suspicious instances may incur significant slowdown (see TABLE 3). With prioritization on instances, the examinations can expose bugs quickly. As such, prioritization allows developers to debug earlier.

Each suspicious instance is a sequence of three access events. Each event is encoded with an execution context [4][5]. On examining such an instance, a detector needs the execution context of each event to actively expose a failure. Such an execution context may vary from a fine-grained abstraction (e.g., the call stack when generating the event) to a coarse-grained one (e.g., the belonging basic code block of the event). Also, a suspicious instance can be regarded as a point in the 3-dimensional space rendered by three axes of execution contexts, one for each of its three events \((e_p, e_r, e_c)\). As such, two instances different in the original 3-dimensional space may be equivalent in some projected 1- or 2-dimensional subspaces.

The experiment presented in [17] is inspiring: The execution context of each event is the instruction (machine code); and two suspicious instances are equivalent if they are same after projected onto the 1-dimensional subspace \(c\) determined by the execution context of \(e_c\). By randomly selecting one suspicious instance from each generated equivalence class for examination, their results show that their selected abstraction level and subspace are highly effective in exposing atomicity violation bugs.

This paper presents a comprehensive controlled experiment to study the influence of the above two factors: the abstraction level of execution context and the subspace after dimensional projection, on benchmarks and real-world bugs. Our experiment examined 5 abstraction levels and 7 subspaces, in which two abstraction levels and three subspaces are never reported in the literature. For each pair of an abstraction level and a subspace, a partition-based prioritization technique is synthesized which firstly partitions a set of suspicious instances into equivalence classes. The instances in each equivalence class are treated as equivalent at the specified level and subspace. Then, the technique randomly selects instances from all such classes iteratively to form a prioritized sequence of these instances. For each such prioritized sequence, we measure its cost-effectiveness of the detection of all atomicity viola-
tion bugs, which expose vulnerability issues of the programs or failures, in the confirmation runs.

The results show that partition-based prioritization significantly improves the cost effectiveness in exposing atomicity violation bugs against random ordering by 30% and untreated ordering by 59% in terms of average percentage faults detected (APFD) [22]. It also shows that (1) using a coarse-grained abstraction level is significantly more cost-effective than using a fine-grained abstraction level, and (2) projecting onto the subspace \( p \) is significantly more effective than projecting onto the other subspaces. Moreover, eight newly synthesized techniques can be significantly more effective than the technique modeled after the state-of-the-art dynamic atomicity violation detector [17][20].

The main contribution of this paper is threefold. (1) It is the first work that examines the effects of partitioning criteria in prioritizing suspicious instances for cost-effective detection of atomicity violation bugs. (2) It reveals a new and effective abstraction level (basic block) and a new and effective subspace \( (p) \). Eight techniques studied in our experiment are more effective than the technique modeled after a state-of-the-art detector and seven of them adopt either basic block or subspace \( p \) or both. (3) The data analysis result interestingly shows that an equivalence class containing a bug instance is less likely to contain non-buggy instances.

We organize the rest of the paper as follows: Section 2 reviews the preliminaries. Section 3 presents the partitioning criteria and prioritization strategy followed by a technique overview in Section 4. Section 5 presents the controlled experiment and results. Section 6 reviews the related work, followed by the conclusion in Section 7.

2 PRELIMINARIES

2.1 Execution Context

Each event in a trace can be modeled by an execution context such that events in traces can be identified. We use the object frequency abstraction (i.e., the algorithm in [4]) to model the execution context of each event in a trace [5]. This form of execution context \( (s, l, n) \) combines a call stack fragment \( s \) of a given event \( e \), the instruction \( l \) that generates the event \( e \), and the number of times \( n \) (known as the frequency) that the same combination \( (s, l) \) has been used to compute an execution context along the trace.

2.2 Suspicious Instances and Bug Instances

TABLE 1 lists the four sets of patterns of single-variable atomicity violation between two threads on a memory location [15][20]: RWR means that in between two consecutive reads to a location \( x \) by thread \( t_1 \), thread \( t_2 \) writes to \( x \). The other patterns can be interpreted similarly.

We denote the set of suspicious instances predicted by a dynamic detector [14][17][20] by \( \Omega = \{\omega_1, \ldots, \omega_k, \ldots, \omega_m\} \), where \( \omega_k = (e_{\text{r}_1}, e_{\text{r}_2}, e_{\text{c}_1}) \) is a suspicious instance that follows a pattern \( (e_{\text{r}_1}, e_{\text{c}_1}) \) in TABLE 1 for \( k = 1 \) to \( m \).

If an execution crashes or violates an assertion in some test oracles after \( \omega \) is observed, we refer to \( \omega \) as a bug instance. Two bug instances \( \omega_1 = (e_{\text{r}_1}, e_{\text{c}_1}, e_{\text{d}_1}) \) and \( \omega_2 = (e_{\text{r}_2}, e_{\text{d}_2}, e_{\text{c}_2}) \) are referred to as different faults if: (1) at least one code statement in \( \omega_1 \) is different to the code statements in \( \omega_2 \); and (2) the stack traces (reported by gdb’s bt) after crash are different or the violated assertions are different. Otherwise, they are referred to as a same fault.

3 PARTITIONING CRITERIA AND PRIORITIZATION

Abstraction level and subspace are two design factors of partitioning. We use \( \text{abs}(e, X) \) to denote the abstraction of an event \( e \) at a given abstraction level \( X \). For instance, the abstraction \( \text{abs}(e, I) \) of an event \( e \) at the instruction level \( I \) indicates the code instruction that generates the event \( e \). Also, each suspicious instance \( \omega = (e_{\text{r}}, e_{\text{c}}, e_{\text{d}}) \) can be formulated as a point in a 3-dimensional space. For ease of presentation, we use a non-empty subset \( Y \) of \( \{p, r, c\} \) to denote the corresponding projected subspace of the original 3-dimensional space. For instance, the subspace \( \{c\} \) projects each suspicious instance \( \omega \) onto its \( c \) component.

3.1 Equivalence Class (Cell)

Given two instances \( \omega_1 = (e_{\text{r}_1}, e_{\text{c}_1}, e_{\text{d}_1}) \) and \( \omega_2 = (e_{\text{r}_2}, e_{\text{c}_2}, e_{\text{d}_2}) \), an abstraction level \( X \), and a subspace \( Y \), we consider \( \omega_1 \) and \( \omega_2 \) equivalent (denoted as \( \omega_1 \equiv \omega_2 \)) if the execution contexts at the abstraction level \( X \) after projecting onto the subspace \( Y \) are the same. For instance, suppose that \( X = I \) and \( Y = \{p, r\} \). Then, \( \omega_1 \) and \( \omega_2 \) are equivalent if \( \text{abs}(e_{\text{r}_1}, I) = \text{abs}(e_{\text{r}_2}, I) \) and \( \text{abs}(e_{\text{c}_1}, I) = \text{abs}(e_{\text{c}_2}, I) \), written as \( \omega_1 \equiv \omega_2 \)\{p, r\}.

Based on a pair of abstraction level \( X \) and subspace \( Y \), the set \( \Omega \) is partitioned into a set of equivalence classes \( \Psi = \pi(\Omega, X, Y) = \{C(\omega_1), \ldots, C(\omega_n), \ldots, C(\omega_m)\} \) which satisfies three conditions: (1) For each equivalence class \( C(\omega) \) \( \in \Psi \), \( \forall \omega_{\text{r}}, \omega_{\text{c}}, \omega_{\text{d}} \in C(\omega), \omega_{\text{r}} \equiv \omega_{\text{r}} \} \). (2) The union of all these equivalence classes is the set \( \Omega \), that is, \( \bigcup_{k=1}^{n} C(\omega_k) = \Omega \). And, (3) no instance can be put into more than one equivalence class: \( \forall C(\omega_i), C(\omega_j) \in \Psi, \{C(\omega_i) \cap C(\omega_j) = \emptyset \}, \) for \( 1 \leq i, j \leq t \).

In mathematics, an equivalence class is called a cell.

3.2 Prioritization Strategy

Algorithm 1 shows our prioritization strategy. It accepts the set \( \Omega \), an abstraction level \( X \), and a subspace \( Y \) as inputs. It partitions \( \Omega \) into a set \( \Psi \) of equivalence classes (line 01). If \( \Psi \) is non-empty (line 02), Algorithm 1 randomly orders the equivalence classes in \( \Psi \) (line 03). It then visits these equivalence classes in turn (line 04). On each visit, it randomly selects an instance \( \omega' \) from \( C(\omega) \), adds \( \omega' \) to the ordered sequence \( W \), and removes \( \omega' \) from \( C(\omega) \) (lines 05-07). If \( C(\omega) \) becomes empty, the algorithm removes \( C(\omega) \) from \( \Psi \) (lines 08-10). It then iterates until \( \Psi \) is empty (lines 02-12). Thus, Algorithm 1 produces a prioritized sequence of instances.

**TABLE 1**

<p>| Patterns of Non-serializable Interleaving |</p>
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Thread Interleaving Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>RWR</td>
<td>( t_i : e_{\text{r}} = \text{read}(x) \ldots \quad e_{\text{c}} = \text{read}(x) \ldots )</td>
</tr>
<tr>
<td>WWR</td>
<td>( t_i : e_{\text{r}} = \text{write}(x)^{\prime} \ldots \quad e_{\text{c}} = \text{write}(x)^{\prime} \ldots )</td>
</tr>
<tr>
<td>RW</td>
<td>( t_i : e_{\text{r}} = \text{read}(x) \ldots \quad e_{\text{c}} = \text{write}(x)^{\prime} \ldots )</td>
</tr>
<tr>
<td>WR</td>
<td>( t_i : e_{\text{r}} = \text{write}(x)^{\prime} \ldots \quad e_{\text{c}} = \text{read}(x) \ldots )</td>
</tr>
</tbody>
</table>
4 OVERVIEW OF PRIORITIZATION TECHNIQUES

4.1 Abstraction Level (X-Dimension)

We have reviewed existing work, and find the following abstraction levels proposed in the literature applicable to our work. In each abstraction level below, we also cite representative detectors that each uses the corresponding abstraction level for ease of readers’ references.

MagicFuzzer [3] used the object frequency abstraction \(s, I, n\) to represent an event \(e\), which is denoted as \(O\).

AssetFuzzer [14] used the \(k\)-object sensitivity abstraction \(s, I\) which was the object frequency abstraction without the \(n\) component to represent an event \(e\), which is denoted as \(K\).

CTrigger [17][20], AtomFuzzer [18], and Maple [29] used the instruction that generated an event \(e\) to represent \(e\). We denote this instruction level by \(I\).

Each statement in the source code may be mapped to a set of instructions. Thus, we include the corresponding statement as another level to represent the same events. We use \(S\) to denote this statement level.

Each statement in the code is part of a basic block (single-entry-single-exit code block). For an event \(e\), we also study the basic block abstraction level, denoted by \(B\).

These five levels are increasingly more abstract. We refer to each of these five ways for abstracting events as an abstraction level \(X\). For the ease of presenting our controlled experiment and results, we simply refer to \(O, K, I, S,\) and \(B\) as \(X_1, X_2, X_3, X_4,\) and \(X_5\), respectively. The x-axis of the plot in Fig. 1 shows these five abstraction levels.

One statement or basic block may include more than one instruction to generate access events, whereas, the other three abstraction levels only denote each event by one instruction. Thus, we refer to the statement and basic block abstraction levels as coarse-grained abstraction levels and the other three as fine-grained ones. To the best of our knowledge, no existing work has explored how different granularities of abstraction levels affect the effectiveness of dynamic detection of atomicity violations.

4.2 Subspace (Y-Dimension)

Our controlled experiment enumerated all non-empty subsets of the component set \([p, r, c]\) to represent different subspaces of the original space. We refer to the subspaces \([p, r, c], [p, r], [r, c], [p, c], [p], [r],\) and \([c]\) as \(Y_1, Y_2, Y_3, Y_4, Y_5, Y_6,\) and \(Y_7\), respectively. The y-axis of the plot in Fig. 1 shows these 7 subspaces.

These two dimensions are orthogonal to one another. Fig. 1 shows some exemplified detectors which map to these two dimensions. For example, CTrigger adopted instruction level with subspace \([c]\) (i.e., \(c\), component), whereas AssetFuzzer used \(k\)-object sensitivity abstraction level with subspace \([p, r, c]\).

4.3 Prioritization Techniques

Using the above abstraction levels and subspaces, we produced 35 prioritization techniques from Algorithm 1 that each takes a set \(\Omega\) as input and outputs a sequence \(W\).

Each technique corresponds to one unique coordinate in Fig. 1. Thus, we use the coordinate \((X, Y)\) in this plot to denote a technique, and refer to the technique as \(T(X, Y)\). For instance, we refer to the technique at the coordinate \((3, 7)\) as \(T(3, 7)\), which is a technique modeled after CTrigger.

We included two controlled techniques random ordering (R) and untreated ordering (U) [22] in our experiment. Technique R randomly selects instances one by one. Technique U sequentially traverses the event sequence directly produced by the Maple tool [29] to generate suspicious instances one by one.

4.4 Illustration of Techniques

Fig. 2(a) shows a code excerpt simplified from the MySQL code listing (each line is annotated with a line number and a basic block number). When MySQL receives a connection request, thread \(t_1\) will process some tasks (lines 11-14). Specifically, \(t_1\) saves the data kept in the shared variable info to the local variable localInfo (line 12), processes the data, and then gets another data from input buffer (line 13). After all tasks are processed, \(t_1\) writes the data kept in info to a file if info is not NULL (lines 15-16). Another thread \(t_2\) cleans the data kept in info (line 17) before MySQL closes a connection. This code excerpt contains an atomicity violation bug, which is highlighted in Fig. 2(a): After \(t_1\) has executed l5, if \(t_2\) executes l7 followed by \(t_1\) executing l6, the execution will crash due to a null pointer exception occurred at l6.

Fig. 2(b) illustrates an execution trace for the code shown above. Based on TABLE 1, we predicted a set \(\Omega_s\) of 5 suspicious instances from the trace, shown as \(\Omega_1\) to \(\Omega_5\) in Fig. 2(c). After confirmation on these five instances (see Section 5.5), only \(\Omega_5\) is labeled as a bug instance.

Fig. 2(d) illustrates how the prioritization technique \(T(5, 5)\) partitions and prioritizes the set \(\Omega_s\). By inputting
(Ωb, X5, Y5), Algorithm 1 firstly produces a set of equivalence classes as shown in the upper part of Fig. 2(d). Two equivalence classes are produced and the bug instance ω5 is in the equivalence class C(ω5) that contains 1 element. Algorithm 1 uses a round-robin strategy to randomly select instances from these equivalence classes (as illustrated in the lower part of Fig. 2(d)). In this way, either the first or the second selected instance must be ω5. The average rank is thus 1.5 (i.e., E[ωi] = 1/2). In other words, on average, a detector can detect the atomicity violation bug by checking 50% (i.e., 1.5/3) of suspicious instances.

Fig. 2(e) and (f) illustrate two techniques T(3, 5) (i.e., instruction level with subspace [p]) and T(3, 7) (i.e., instruction level with subspace [c]), respectively. Based on the partitioning results, on average, T(3, 5) is expected to check 40% (i.e., 2/5) suspicious instances to trigger the bug. Whereas, on average, T(3, 7) is expected to check 50% (i.e., 2.5/5) suspicious instances to trigger the bug.

If we use random ordering to select instances, on average, 60% (i.e., 3/5) of suspicious instances should be examined to trigger the bug. Since ω5 is at the end of Ωb, using untreated ordering we should check all instances before triggering the bug.

5 Controlled Experiment

5.1 Research Questions

We study the following three research questions:

- **RQ1**: How much difference can partition-based prioritization offer on the detection of atomicity violations?
- **RQ2**: To what extent will the two design factors affect the effectiveness of suspicious instance prioritization?
- **RQ3**: CTrigger is a state-of-the-art detector that uses abstraction level l and subspace [c]. Is there any other combination of abstraction level and subspace outperforming the technique T(3, 7) modeled after CTrigger?

5.2 Independent Variables

Our experiment has two independent variables: *abstraction level* and *subspace*. By using each pair of abstraction level and subspace to initialize Algorithm 1, we generated 35 techniques, each of which is referred to as a technique T(X, Y) where (X, Y) is the coordinate in Fig. 1.

5.3 Dependent Variable and Measure

We measure the cost-effectiveness of a technique by the metric APFD [22][31] (average percentage faults detected). APFD evaluates the percentage of bugs detected versus the percentage of suspicious instances confirmed. The cumulative percentage of bugs detected is represented by a curve, and the area under the curve represents the weighted average percentage of bugs detected over the life of confirming suspicious instances [8][22]:

\[
APFD = 1 - \frac{\sum_{i=1}^{d} TFi}{d|Ω|} + \frac{1}{2|Ω|}
\]

In the formula, |Ω| means the number of suspicious instances in the set Ω, and d is the number of bug instances. TF denotes the number of suspicious instances needed to be confirmed to expose the bug instance i. A higher APFD value indicates higher effectiveness [22].

5.4 Benchmarks

TABLE 2 summarizes the 10 benchmarks used in our experiment. PBZIP2 and Aget are utility programs. LU, FFT, and Barnes are scientific programs from the Splash2 suite [27]. Memcached, Apache, and MySQL are server programs. We have used all the benchmarks from previous work [17][20] (expect Mozilla-extract, which is unavailable), and added three more benchmarks (i.e., Aget, Memcached, and MySQL#2) from [28][29]. We obtained the test inputs of the Memcached, Apache, and MySQL benchmarks from [20][29]. For the remaining utility programs and scientific programs, we followed [28][29] to construct test cases for these programs.

TABLE 2 shows the name, version, and the size in SLOC [24] of each benchmark, followed by the number of threads and number of accesses to non-stack memory in an execution. The next two columns show the time spent by native runs (in seconds) and the bug descriptions. Specifically, some of these bugs result in program failures, and some other bugs expose vulnerability issue in the corresponding benchmarks.

5.5 Experimental Setup

Our experiment was performed on the Ubuntu Linux 12.04 x86_64 configured with two 2.90GHz Xeon Cores, 4GB physical memory, and GCC 4.4.3 compiler.
We have developed a test framework, called ASP standing for Abstraction-Subspace-Partitioning, atop Maple [29]. ASP extended Maple to compute the object frequency abstraction with the default setting [4] to collect the call stack fragments of object frequency abstraction (i.e., the length of the call stack fragment is 8). It also collected the abstraction of each event at each studied level.

In the predictive phase, ASP executed each benchmark by monitoring all shared memory accesses and collecting happens-before relations and locksets of memory access events. It reported a suspicious instance \((e_p, e_v, e_c)\) at the object frequency abstraction level if these three events \(e_p\), \(e_v\), and \(e_c\) satisfy the following four conditions: (1) Their access sequences is \((e_p, e_v, e_c)\) or \((e_v, e_p, e_c)\). (2) The permuted sequence \((e_p, e_v, e_c)\) can match with a pattern in TABLE 1. (3) There is no ordered synchronization between \(e_c\) and \(e_c\) (for the case \((e_p, e_v, e_c)\) or between \(e_v\) and \(e_v\) (for the case \((e_v, e_p, e_c)\)). (4) \(e_p\), \(e_v\), and \(e_c\) are not in mutual exclusion [17]. ASP collected a set \(\Omega\) of suspicious instances after one predictive run. For each benchmark, ASP repeated the predictive phase 30 times, and collected in total 300 sets of \(\Omega\) for the 10 benchmarks. Then, for each \(\Omega\), ASP applied each prioritization technique 100 times to produce 100 prioritized sequences of suspicious instances.

The maximum time spent by any technique to prioritize any set of \(\Omega\) detected in the experiment never exceeded 0.6 seconds on each benchmark. From the third column in TABLE 3, we concluded that the time spent on prioritization is negligible. Hence we do not present the results of the time spent on prioritization in this paper.

ASP attempted to confirm each suspicious instance at the object frequency abstraction level by the algorithm presented in [17]. In the course of confirmation run for a given instance \((e_p, e_v, e_c)\), if a thread had executed event \(e_p\) and was going to execute event \(e_v\), ASP would suspend the thread till event \(e_v\) was executed and then produced \((e_p, e_v, e_c)\). Similarly, if a thread was going to execute event \(e_v\), ASP would suspend the thread until some other thread had executed \(e_c\) and was about to execute \(e_c\). After that, the former thread would be resumed to execute \(e_c\) before the latter thread executed \(e_c\). At each suspension point, following CTrigger [17], ASP set up a time-out threshold, which was 1000 milliseconds, consisting of 20 sleeps each with delay of 50 milliseconds.

If the above attempt of confirmation of a suspicious instance resulted in a non-serializable interleaving [17], and the execution crashed or violated an assertion in the test oracle after the interleaving was observed, ASP labeled the suspicious instance as a bug instance. With all suspicious instances labeled, ASP computed the APFD value of each prioritized sequence. ASP collected 3000 APFD values per technique on each benchmark (which exceeded the minimum threshold (1000) stated in a practical guideline for techniques involving random factor [1]). For random ordering, ASP repeated the process for 3000 times to compute 3000 APFD values per benchmark.

TABLE 3 shows the statistics on confirming each set \(\Omega\) of suspicious instances. It shows the mean number of suspicious instances, the mean time spent (in seconds) to confirm one set of \(\Omega\), the corresponding slowdown incurred by confirmation over the native run, and the number of bug instances \(d\) in each benchmark. All bug instances exposed refer to different faults (see Section 2.1).

A native run of each benchmark only took at most a few seconds to complete. We observed that the slowdown incurred in the confirmation phase was significant. For instance, on MySQL2, the native run only took 3.57 seconds, but the slowdown factor for atomicity violation

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Mean APFD (%)</th>
<th># of techniques in F more effective than R (out of 35)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>U</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>&gt;=20%</td>
<td>&gt;=10%</td>
</tr>
<tr>
<td>PBZIP2</td>
<td>10.71</td>
<td>50.39</td>
</tr>
<tr>
<td>LU</td>
<td>2.57</td>
<td>49.60</td>
</tr>
<tr>
<td>FFT</td>
<td>3.86</td>
<td>49.91</td>
</tr>
<tr>
<td>Aget</td>
<td>0.41</td>
<td>49.85</td>
</tr>
<tr>
<td>Barnes</td>
<td>2.54</td>
<td>49.73</td>
</tr>
<tr>
<td>Memcached</td>
<td>52.67</td>
<td>49.24</td>
</tr>
<tr>
<td>Apache1</td>
<td>42.05</td>
<td>49.11</td>
</tr>
<tr>
<td>Apache2</td>
<td>43.27</td>
<td>50.56</td>
</tr>
<tr>
<td>MySQL1</td>
<td>21.82</td>
<td>49.77</td>
</tr>
<tr>
<td>MySQL2</td>
<td>27.65</td>
<td>50.43</td>
</tr>
</tbody>
</table>

**Average** 20.76 49.86 79.62 24.5 29.6 34.2

**TABLE 4**

Fisher’s LSD Test for Comparing Random Ordering With Our Family in Terms of Mean APFD
Fig. 3. Overall comparisons among the untreated ordering (U), random ordering (R), and our family of techniques (F).

5.6 Data Analysis

5.6.1 Answering RQ1

In this section, following [31], we group all individual APFD values of all the 35 techniques into one dataset. We refer to the aggregated results as the results of our family (of techniques) denoted by F. We compare the APFD results among U (untreated ordering), R (random ordering), and F. Fig. 3 shows the results in box-and-whisker plots. In each plot, the x-axis lists prioritization techniques and the y-axis lists their APFD values for the benchmarks.

In Fig. 3, across all benchmarks, F is significantly more effective than U by a large extent. On 8 benchmarks, the boxes of R and F do not overlap; and for all benchmarks, F achieves significantly higher median APFD than R.

In Table 4, in terms of mean APFD, F is more effective than U by 58.9% and R by 29.8%. On Memcached, Apache#1, and Apache#2, U and R perform similarly. But on some other benchmarks, U is less effective than R by a large extent. The ANOVA test confirms that U, R and F are statistically different at the 5% significance level on each benchmark. Specifically, F is more effective than both U and R on all benchmarks, and R is more effective than U on all benchmarks except Memcached.

We have further conducted the Fisher’s LSD test [26] at the 5% significance level. The rightmost three columns of Table 4 show that 25 techniques in F are significantly more effective than R by 20% or more on all benchmarks except PBZIP2 and FFT, which only have 7 and 10 techniques outperforming R by 20%, respectively. On average, 29 techniques in F are significantly more effective than R by at least 10%. On 4 benchmarks (i.e., LU, Aget, Apache#1, and MYSQL#1), all 35 techniques in F are at least as effective as R, and on 5 benchmarks (i.e., PBZIP2, FFT, Barnes, Apache#2, and MYSQL#2), 34 techniques in F are at least as effective as R. On Memcached, R cannot outperform 32 techniques in F.

We can answer RQ1 that our partition-based prioritization can significantly improve the rate of fault detection in exposing atomicity violation bugs.

5.6.2 Answering RQ2

In this section, we analyze the data along the dimensions of abstraction level X and subspace Y to study the characteristics (impactful factors) of effective techniques.

All the 35 techniques in F are shown in Fig. 4. In the figure, from left to right, there are 8 plots: one for each level of Y as well as the overall results along the X dimension. In each plot, there are 5 levels of X. The technique T(3, 7), which is modeled after CTrigger, locates at X3. We follow the data analysis presented in [31] to compare the median values of the bars in each plot with the bar for X3. If the median of a bar is higher than the median of the bar for X3, we highlight the former bar (in red). Other plots in Fig. 4 can be interpreted similarly.

The rightmost plot in Fig. 4 shows that techniques at abstraction levels X4 and X5 are more effective than techniques at abstraction levels X1, X2, and X3. In each of the remaining 7 plots, which shows the result along the Y dimension, X5 is also most effective among all abstraction levels in that plot, X4 is more effective than X3 except on Y7, and X1 is the least effective one followed by X2.

We have performed the Fisher’s LSD test to compare these techniques along the X dimension at the 5% significance level. In Table 5, if the test shows that a technique T(i, j) is more effective than, as effective as, or less effective than the technique T(3, j) (1 ≤ i ≤ 5, 1 ≤ j ≤ 7), then we mark the corresponding cell as >, =, and <, respectively. The results consolidate our above observation that X3 cannot outperform X5 in all cases; and X3 cannot outperform X4 on all but one case, and X3 outperforms both X1 and X2 in all cases.

Fig. 4. Comparisons along the abstraction level dimension. X3 is the abstraction level that the controlled technique T(3, 7) is located. Each technique T(i, j) having a higher median APFD value than the technique T(3, j) is highlighted.
TABLE 5
FISHER’S LSD TEST FOR COMPARING DIFFERENT ABSTRACTION LEVELS WITH X3 IN TERMS OF MEAN APFD

<table>
<thead>
<tr>
<th>Y-Dimension</th>
<th>X1: O</th>
<th>X2: K</th>
<th>X3 (Control): J</th>
<th>X4: S</th>
<th>X5: B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y1: (p, r, c)</td>
<td>&lt;</td>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>Y2: (p, r)</td>
<td>&lt;</td>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>Y3: (p, c)</td>
<td>&lt;</td>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>Y4: (p, c)</td>
<td>&lt;</td>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>Y5: (p)</td>
<td>&lt;</td>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>Y6: (r)</td>
<td>&lt;</td>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>Y7: (c)</td>
<td>&lt;</td>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>Overall</td>
<td>&lt;</td>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
</tbody>
</table>

TABLE 6
FISHER’S LSD TEST FOR COMPARING DIFFERENT SUBSPACES WITH Y7 IN TERMS OF MEAN APFD

<table>
<thead>
<tr>
<th>X-Dimension</th>
<th>Y1 (p, r, c)</th>
<th>Y2 (p, r)</th>
<th>Y3 (r, c)</th>
<th>Y4 (p, c)</th>
<th>Y5 (p)</th>
<th>Y6 (r)</th>
<th>Y7 (Control): c</th>
</tr>
</thead>
<tbody>
<tr>
<td>X1: O</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&lt;</td>
<td>=</td>
</tr>
<tr>
<td>X2: K</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&lt;</td>
<td>=</td>
</tr>
<tr>
<td>X3: I</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&lt;</td>
<td>=</td>
</tr>
<tr>
<td>X4: S</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&lt;</td>
<td>=</td>
</tr>
<tr>
<td>X5: B</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&lt;</td>
<td>=</td>
</tr>
<tr>
<td>Overall</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&lt;</td>
<td>=</td>
</tr>
</tbody>
</table>

TABLE 7
FISHER’S LSD TEST FOR COMPARING INDIVIDUAL TECHNIQUE WITH TECHNIQUE T(3,7)

<table>
<thead>
<tr>
<th>Combination</th>
<th>Y1 (p, r, c)</th>
<th>Y2 (p, r)</th>
<th>Y3 (r, c)</th>
<th>Y4 (p, c)</th>
<th>Y5 (p)</th>
<th>Y6 (r)</th>
<th>Y7 (c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>X1: O</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>X2: K</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>X3: I</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>X4: S</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>X5: B</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
</tbody>
</table>

The levels X4 (statement) and X5 (basic block) are coarser in granularity than levels X1 (object frequent abstraction), X2 (k-object sensitivity), and X3 (instruction). The result seems indicating that in terms of both mean and median APFDs, the use of a coarse-grained abstraction level in a partition-based prioritization technique can be more effective than a fine-grained one.

Similarly, for the analysis along the Y dimension, we compare each technique T(i, j) with T(i, 7) using Fisher’s LSD test at the 5% significance level. TABLE 6 shows the results. This table can be interpreted similarly to TABLE 5. (We note that all 35 techniques have been shown in Fig. 4.)

From TABLE 6, we observe that overall speaking, techniques at subspaces Y5 and Y7 are more effective than techniques at Y1-Y4 and Y6. Moreover, Y5 is always significantly more effective than Y7; Y1 and Y3 never outperform Y7; and, Y6 can outperform Y7 only at the subspace X1, but is less effective at all other subspaces.

The results indicate that in terms of mean and median APFDs, Y5 is most effective among all studied subspaces. For the other studied subspaces, their results are mixed.

We further use the ANOVA tests to check whether the techniques in F are different significantly in terms of the abstraction level and in terms of subspace, and whether there is any significant interaction between these two dimensions. On each benchmark, the calculated p-values in all cases are less than 0.001, which reject the null hypothesis that there is no significant difference among them.

To answer RQ2, we find that both abstraction level and subspace are significant factors. The use of a coarse-grained abstraction level is more effective than the use of a fine-grained abstraction level. The use of the basic-block abstraction level (i.e., X5) achieves the most effective result among the studied levels. Among all subspaces, techniques at the subspace {p} (i.e., Y5) are most effective.

5.6.3 Answering RQ3

In this section, we compare each technique to the technique T(3,7). Similar to TABLE 5, in TABLE 7, if a technique in F is more effective than, as effective as, and less effective than the technique T(3,7), we mark the corresponding cell with >, =, and <, respectively.

From TABLE 7, we find that 8 techniques T(5, 1), T(5, 2), T(5, 4), T(5, 5), T(5, 7), T(4 2), T(4, 5), and T(3, 5) are more effective than T(3,7); and one technique T(4,4) is as effective as T(3,7). The results indicate that 9 techniques are at least as effective as T(3,7). Seven of these 9 techniques adopt either X5 or Y5 or both. This high concentration of effective techniques is interesting.

To answer RQ3, we find that 8 techniques outperform T(3,7), which is modeled after a state-of-the-art detector. These techniques mostly concentrate along abstraction level X5 and subspace Y5. To the best of our knowledge, they have not been reported in existing literature.

5.7 Discussion on Results

In this section, we study the underlying reason on why abstraction levels and subspaces would affect the effectiveness of prioritization techniques.

We first recall from Section 3.1 the following terminologies: Ω is a set of suspicious instances. The set Ω can be partitioned into a set Ψ of cells (i.e., equivalence classes) based on a given abstraction level X and a given subspace Y (i.e., Ψ = π(Ω, X, Y)). Moreover, we refer to a cell that contains a bug instance w as a buggy cell C(w).

The probability Pr(Ψ) of a suspicious instance selected from Ψ by Algorithm 1 being a bug instance is decided by two factors: whether a buggy cell is selected, and whether a bug instance in that cell is picked. We denote the proportion of all buggy cells in Ψ by PC(Ψ) = λ / |Ψ|, where λ is the number of buggy cells in Ψ. We also denote the mean proportion of bug instances among all buggy cells in Ψ by PL(Ψ) = (Σd1=1Σd2=1 / |C(wo)|) / λw, where d is the number of bug instances in the buggy cell C(wo) and Σd1=1Σd2=1 d is the probability Pr(Ψ) is defined as PC(Ψ) × PL(Ψ).

Intuitively, if the probability Pr(Ψ) of selecting bug instances from Ψ is higher, then prioritization on this set of Ψ will uncover bug instances more quickly, which results in a higher APFD. Our intuition is validated by the experimental results stated below.

As stated in Section 5.5, we have produced 300 sets of Ω for the 10 benchmarks, and applied 35 techniques to partition each set of Ω, resulting in 10500 sets of Ψ. We have prioritized each set of Ψ and measured their corresponding APFD values. We have computed the above probability Pr(Ψ) for each of these sets of Ψ.

Fig. 5 summarizes the APFD values against their corresponding mean Pr(Ψ) of each technique with a curve representing the best-fitting regression line of these points. The figure consistently shows that if a partitioning strategy results in a higher probability Pr(Ψ), then prioritization on its corresponding set Ψ achieves a higher APFD.
In the next two subsections, we present the analysis on each dimension of X and Y. We will show that for abstraction levels, \( Pr(\Psi) \) generally increases from X1 to X5, i.e., from fine-grained level to coarse-grained level; and for subspaces, Y5 generally has the highest \( Pr(\Psi) \).

For ease of references, we use \( Pr(X) \), \( PC(X) \), and \( PI(X) \) or \( Pr(Y) \) and \( PC(Y) \) respectively, to denote the results along the X or Y dimension, respectively. For instances, if we discuss a particular level X5, we will use \( Pr(X5) \), \( PC(X5) \), and \( PI(X5) \) to denote the results on all benchmarks and among all subspaces at X5.

### 5.7.1 Comparisons among Abstraction Levels

Fig. 6 (a)-(c) summarize how the values of \( PI(X) \), \( PC(X) \) and \( Pr(X) \) change among the 10500 sets of \( \Psi \) (along the X dimension). In each plot, we have split these 10500 sets, each represented by a point, into 5 abstraction levels. So there are 2100 points per abstraction level. For ease of presentation, we arranged each set of 2100 points in the same abstraction level from left to right in the ascending order of \( PI(X) \), \( PC(X) \) and \( Pr(X) \), respectively. Recall that \( PC(\Psi) = \frac{\lambda}{|\Psi|} \) and \( |\Psi| \) in these benchmarks are in different orders of magnitudes. To combine the results of all benchmarks, we normalized the maximum \( PC(\Psi) \) on each benchmark to 1. We also did the same normalization of \( Pr(\Psi) \) on each benchmark. (Normalization takes no effect on \( PI(\Psi) \) because the maximum \( PI(\Psi) \) on each benchmark is 1.) In Fig. 6, the mean values of \( PI(X1)-PI(X5) \), \( PC(X1)-PC(X5) \), and \( Pr(X1)-Pr(X5) \) after normalization are shown as the legends of the corresponding plots.

Fig. 6(a) shows that the mean values of \( PI(X1)-PI(X5) \) are within 0.90-0.94. Their differences are small, and their absolute values are close to 1. This indicates that for most of the buggy cells (irrespective to its abstraction level) their suspicious instances are all bug instances. This finding helps explain the effectiveness of the partition-based strategy used in [17], which selects one instance from each cell for confirmation until all cells are selected.

Bug instances are concentrated in some cells and the mean values of \( PI(X1)-PI(X5) \) are close to one another. Thus, if the proportion of buggy cells \( PC(\Psi) \) for an abstraction level is higher, then the corresponding probability \( Pr(\Psi) \) will also be higher.

We thus compare the proportion of buggy cells \( PC(\Psi) \) for each level in Fig. 6(b). In Fig. 6(b), the curves for the five levels are arranged from top to bottom as \( PC(X5) > PC(X4) > PC(X3) > PC(X2) > PC(X1) \). The curves for \( PC(X5) \) and \( PC(X4) \) are tangling when their values are low, but \( PC(X5) \) is higher when their values are high. The curves for \( PC(X2) \) and \( PC(X1) \) are lower than the others by a large extent. The curve for \( PC(X3) \) is located below the curves for \( PC(X5) \) and \( PC(X4) \).

The results show that by using a coarser abstraction level, the proportion \( PC(\Psi) \) of buggy cells is higher. We observed that the underlying reason can be multi-fold:

First, the total number of cells \( |\Psi| \) at a coarse-grained level \((X4-X5)\) is smaller than that at a fine-grained level \((X1-X3)\) as shown in the second row in TABLE 8. The ratio of number of cells \( |\Psi| \) to the total number of suspicious instances \( |\Omega| \) decreases from X1 to X5, which indicates that the number of cells \( |\Psi| \) decreases from X1 to X5. This is due to the relationships between any two abstraction levels studied in the experiment: Each executed basic block contains multiple statements; each executed statement is constructed by a set of code instructions; and each executed instruction is associated with one or more call stack instances when it is executed, and such call stack instances can be distinguished by their occurrence orders.

---

**Fig. 5.** APFD increases with the probability of selecting bug instances \( Pr(\Psi) \)

**Fig. 6.** Statistics of proportion of bug instances in buggy cells \( P(\Psi) \), proportion of buggy cells \( PC(\Psi) \), and probability of selecting bug instances \( Pr(\Psi) \) at each abstraction level and each subspace. \( PC(\Psi) \) and \( Pr(\Psi) \) are normalized.
Second, we found that the number of buggy cells $\lambda$ did not vary a lot among abstraction levels X1-X4. The third row in TABLE 8 shows the ratio of number of buggy cells $\lambda$ to the total number of bug instances $d$ slightly decreases across four abstraction levels. It is because two bug instances should be different by at least one statement; otherwise, they refer to the same bug in the source code listing. So, different bug instances should be distinguished at the statement level (X4). As such, if the total number of cells $|\Psi|$ decreases from X1 to X4, but the number of buggy cells $\lambda$ does not change a lot, then the proportion of buggy cells $PC(\Psi)$ increases. At the block level (X5), we found that some bug instances shared the same code block although they belonged to different statements. As such, they were grouped into the same buggy cells. As shown in Fig. 6(b), the curve for X5 is not always above that for X4. However, the decrease in the total number of cells $|\Psi|$ at level X5 is still more significant than that at level X4. Therefore, we still observe that the curve for X5 is largely located upper than that for X4 in the plot.

The results of $Pr(\Psi)$ shown in Fig. 6(c) reconcile our above discussions: First, if the proportion of buggy cells $PC(\Psi)$ is higher, then the probability $Pr(\Psi)$ will be higher. Second, the probability $PC(\Psi)$ increases from X1 to X5.

Lastly, TABLE 9 (columns two to six) shows the Fisher’s LSD test to compare these five abstraction levels at the 5% significance level. We use an alphabetically higher letter to represent a higher mean value. Two levels that do not vary a lot $(\Psi)$ or less number of buggy cells $\lambda$ ($X1: O; X2: K; X3: I; X4: S; X5: B$) $(Y1: [p, r, c]; Y2: [p, r]; Y3: [r, c]; Y4: [p, c]; Y5: [p]; Y6: [r]; Y7: [c])$.

### 5.7.2 Comparisons among Subspaces

Fig. 6 (d)-(l) and the rightmost 7 columns of TABLE 9 show the statistics and the LSD test results along the subspace dimension. Fig. 6(d) and TABLE 9 show that $PI(Y1)-PI(Y3)$ are large in values, $PI(Y4)-PI(Y5)$ are slightly smaller than $PI(Y1)-PI(Y3)$ but still larger than or equal to 0.90, and $PI(Y6)-PI(Y7)$ are the smallest. It indicates that for most of the buggy cells at subspaces Y1-Y5, the suspicious instances in these cells are all bug instances.

Subspaces Y5-Y7, Y2-Y4, and Y1 are the projections onto 1-dimensional subspaces, 2-dimensional subspaces, and the original space, respectively. Mathematically, subspaces Y5-Y7 result in fewer cells than subspaces Y2-Y4, which in turn result in fewer cells than the original space Y1. The second row in TABLE 8 under the Y1-Y7 columns shows this trend. For the row $\ell/d$ in the table, Y5 is 0.99 and Y7 is 0.93. This indicates that the subspace $[p]$ results in larger number of buggy cells $\lambda$ than the subspace $[c]$ after partitioning. Although the value of $\ell/d$ at Y6 is only 0.76, which indicates that subspace $[r]$ results in the smallest number of buggy cells $\lambda$, but Y6 also results in the least number of cells $|\Psi|$, which results in a high PC(\Psi) as a whole. Fig. 6(e) also confirms that PC(Y5)-PC(Y7) are located higher than PC(Y1)-PC(Y4) in the plot.

In summary, since Y5 has relatively high $PI(\Psi)$ and the highest $PC(\Psi)$, Fig. 6(f) reconciles that $Pr(Y5)$ is the highest, which indicates that subspace Y5 is more effective than any other subspaces in our experiment.

### 5.8 Threats to Validity

Our experiment incurred a number of threats to validity. AFPD did not reflect the fact that different suspicious instances may have different confirmation time. On each benchmark, we examined the execution time of all suspicious instances in the confirmation phase, and found the differences among their required time were marginal. Another threat was on comparing the results of the 35 techniques to T(3,7). Fig. 1 has indicated that AssetFuzzer can be classified as the technique T(2,1), Maple as the techniques T(3,1), T(3, 2), and T(3,3), and AtomFuzzer as the technique T(3,1). From TABLE 7, we find that these techniques are all less effective than T(3,7). This result further lowers the threats on using T(3,7) as a controlled technique in our study. We only used the benchmarks from some previous work [17][28][29], which contain bugs resulting in program failures or bugs exposing vulnerability issues of the corresponding benchmarks. The interpretation of the results beyond these benchmarks should be careful.

The prioritization results were limited by the set of suspicious instances detected in the predictive phase in the experiment. To lower the threats, we further used PCT [2] and RandDelay [7] thread schedule generation techniques to produce other sets of suspicious instances. The experimental results on these two thread schedules are presented in the appendix of this paper and are consistent with the conclusions presented above. Algorithm 1 uses a random strategy to select instances from equivalence classes. We further compared Algorithm 1 to the distance-based strategy proposed by CTrigger [17]. The comparisons are presented in the appendix of this paper.

### 6 Related Work

We review closely related work in this section.

**Dynamic detection of atomicity violations:** Atomizer [9] reports suspicious atomicity violations only, and AtomFuzzer [18] complements it by adding a confirmation phase for the same trace. CTrigger [17][20], PENEL-OPE [25], and AssetFuzzer [14] alternatively use a predictive phase to predict suspicious instances and verify these instances through follow-up traces. Among existing work, only CTrigger ranks suspicious instances based on the distance between $e_p$ and $e_c$. Such an approach is orthogonal to our partitioning work.
Two-phase strategy: This strategy has been used to detect many types of concurrency bugs, including data races [23] and deadlocks [3][5]. But, the number of predicted instances in the predictive phase remains large. Our work shows that arranging these suspicious instances using a partition-based prioritized technique can be cost-effective. We believe that our results and their results complement one another. It should also be noted that there are also other techniques [2][10] not using such two-phase strategy in detecting concurrency bugs.

Prioritization and code coverage: Most test case prioritization techniques [8][22][31] are coverage-based [6] and use basic code-based coverage such as function coverage [8] and branch coverage [22] as the test adequacy criterion. The effects of coarse and fine granularity on coverage criteria have been studied in [8]. Different from them, our notion of granularity is the abstraction level of suspicious instances, which is irrelevant to code coverage.Besides, the coverage domains of most coverage criteria proposed in concurrency bug detection [12] are unable to be precisely calculated. To the best of our knowledge, SimRT [30] is the first work that selects and prioritizes test cases for detecting concurrency bugs (data races specifically). Different from us, the main element SimRT considered is the modified code at statement level. SimRT has not studied different abstraction levels and subspaces.

Bug classification: Bug classification techniques such as Windows Error Reporting [11] and Portend [13] automatically classify bug reports or bugs and inform developers to fix the critical bugs first. Griffin [19] further clusters memory-access patterns in the failing executions, thereby constructing bug contexts from clustered accesses. These techniques are all built atop bugs that have already been exposed. Our work focuses on how to cost-effectively expose suspicious cases as real bugs.

7 Conclusion

This paper has reported the first controlled experiment that examines the effect of partition-based prioritization on the exposure of single-variable atomicity violation bugs by a two-phase strategy. The results have shown that arranging suspicious instances via untreated ordering or random ordering is significantly less effective than arranging them via our partition-based prioritization in exposing failures in program outputs or vulnerability of the programs. The use of a coarse-grained abstraction level is also found to be more effective than the use of a fine-grained abstraction level. Among all studied subspaces, the experiment has found that the use of the subspace $\pi$ (i.e., $Y_5$) can be more effective than the use of other subspaces. Moreover, eight techniques can be significantly more effective than the technique that is modeled after the state-of-the-art detector, and all of them are not reported before. Among these techniques, seven techniques concentrate along $X_5$ or $Y_5$, which are previously unstudied abstraction level and subspace in concurrency bug detection research.

References

Shangru Wu is a PhD student at Department of Computer Science, City University of Hong Kong. He received his BEng degree in Software Engineering from International School of Software, Wuhan University, China. His current research interest includes program analysis and cost reduction of dynamic techniques in detecting concurrency bugs.

Chunbai Yang is a PhD student at Department of Computer Science, City University of Hong Kong. He received his BEng degree in Software Engineering from School of Software, Tianjin University, China. His current research interest includes dynamic program analysis on detection of concurrency bugs in multithreaded programs.

Changjiang Jia is a PhD student at Department of Computer Science, City University of Hong Kong. He received his BEng and MEng degrees from National University of Defense Technology, China. His research interests include testing techniques and strategies for concurrent software and service-based software.

W.K. Chan is an associate professor at Department of Computer Science, City University of Hong Kong. He is the editor of Journal of Systems and Software, and guest co-editors of a few international software engineering journals, program co-chair of SETA of COMPSAC 2015 and the area chair of ICWS 2015. His research interest is software testing and analysis for large-scale applications.

FURTHER EVALUATION

In Section 5, we used the native schedule to perform the controlled experiment. In this section, we further evaluated our experiment by using two other kinds of thread schedule generation techniques: PCT [32] and the random time delay (RandDelay) [33].

In PCT, each thread is randomly assigned a priority and the priority would be changed during execution. Each time only the thread having the highest priority can be executed. RandDelay injects random time delays at random points during execution. PCT will bind the execution to one core. In contrast, RandDelay allows programs to run on multi-cores. Since the Maple tool [29] has already been shipped with the implementations of both PCT and RandDelay, therefore we directly used these two thread schedule generation techniques to conduct additional experiments. In the experiment, the depth value (d value) in PCT and the number of random points in RandDelay were both set to 3, which was the default setting of Maple.

The results of the additional experiments under PCT and RandDelay are shown in Fig. 7, Fig. 8, and TABLE 10 to TABLE 15. From the results, we observed that other thread schedules have similar conclusions with the conclusions presented in Section 5.

DISTANCE-BASED STRATEGY

Among existing work, only CTrigger [17] ranks suspicious instances for atomicity violation detection. The ranking metrics they used was the execution distance between \( v_p \) and \( v_e \) (named as local gap) and the execution distance between \( v_e \) and \( v_p \) or \( v_e \) and \( v_c \) (named as remote distance). For a suspicious instance \( \omega \), we denoted the

<table>
<thead>
<tr>
<th>X-Dimension</th>
<th>Y1 (p, r, c)</th>
<th>Y2 (p, r)</th>
<th>Y3 (r, c)</th>
<th>Y4 (p, c)</th>
<th>Y5 (p)</th>
<th>Y6 (r)</th>
<th>Y7 (c)</th>
<th>Y7 (Control)</th>
</tr>
</thead>
<tbody>
<tr>
<td>X1: O</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>X2: K</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>X3: I</td>
<td>=</td>
<td>&gt;</td>
<td>=</td>
<td>&gt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>=</td>
</tr>
<tr>
<td>X4: S</td>
<td>=</td>
<td>&gt;</td>
<td>=</td>
<td>&gt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>=</td>
</tr>
<tr>
<td>X5: B</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>=</td>
</tr>
<tr>
<td>Overall</td>
<td>&lt;</td>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>=</td>
</tr>
</tbody>
</table>

TABLE 12

<table>
<thead>
<tr>
<th>Combination</th>
<th>Y1 (p, r, c)</th>
<th>Y2 (p, r)</th>
<th>Y3 (r, c)</th>
<th>Y4 (p, c)</th>
<th>Y5 (p)</th>
<th>Y6 (r)</th>
<th>Y7 (c)</th>
<th>Y7 (Control)</th>
</tr>
</thead>
<tbody>
<tr>
<td>X1: O</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>X2: K</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>X3: I</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>X4: S</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>X5: B</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>Overall</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
</tbody>
</table>

Fig. 7. Overall comparisons among the untreated ordering (U), random ordering (R), and our family of techniques (F) under PCT.
local gap of \( \omega \) as \( \text{distPC}(\omega) \) and measured the distance by the number of events between \( e_p \) and \( e_c \). Similarly, we denoted the remote distance as \( \text{distR}(\omega) \) and measured the distance by the number of events between \( e_r \) and \( e_p \) or \( e_c \) (depending on the observed sequence).

CTrigger used the local gap as the primary ranking metric and only compared remote distances when two suspicious instances have the same local gap. If the local gap of one suspicious instance is smaller, then the suspicious instance is confirmed earlier. If two suspicious instances have the same local gap, then CTrigger will first attempt to confirm the suspicious instance which has the larger remote distance. We referred to this kind of strategy as \textit{distance-based strategy}.

Fig. 9 shows the result comparison between such distance-based strategy and our family of technique (produced by Algorithm 1). The result shows that, such distance-based strategy is less effective than our family of techniques on all benchmarks except Memcached.

We would like to mention that in Algorithm 1, we implemented a random strategy to select suspicious instances from a set \( \Psi \) of cells. However, this strategy can be replaced by other strategy, such as the above distance-based strategy. Algorithm 2 shows how we adapted Algorithm 1 to apply the distance-based strategy.

Similar to Algorithm 1, at the beginning, Algorithm 2 partitions \( \Omega \) into a set \( \Psi \) of cells based on given abstraction level \( X \) and subspace \( Y \) (line 01). It then starts to calculate the local gap (\( \text{sumDistPC} \)) and the remote distance (\( \text{sumDistR} \)) of each cell (lines 02-09). The local gap of each cell is the summation of all local gaps (\( \text{distPC} \)) of the suspicious instances in the cell (lines 05-06) [17]. Similarly, the remote distance of each cell is the summation of all remote distances (\( \text{distR} \)) of the suspicious instances in the cell (lines 05, 07). When going into the process of iteration, if the partition \( \Psi \) is non-empty (line 10), Algorithm 2 will order the cells in \( \Psi \) using distance-based strategy (line 11).

### Algorithm 2: Strategy to prioritize a set of instances based on the distance between events

Input: \( \Omega = \{\omega_1, \ldots, \omega_n\} \): a set of suspicious instances
Input: \( X \): an abstraction level
Input: \( Y \): a subspace
Output: \( W \): a sequence of suspicious instances, initially empty

01 Partition \( \Omega \) into a set \( \Psi \) of cells
\( \Psi := \pi(\Omega, X, Y) = \{C(\omega_1), \ldots, C(\omega_n)\} \)
// prioritization
02 for each cell \( C(\omega) \) in \( \Psi \) do
03 \quad \text{sumDistPC}_{\omega} := 0 // primary metric
04 \quad \text{sumDistR}_{\omega} := 0
05 \quad \text{for each instance } \omega' \text{ in } C(\omega) \text{ do}
06 \quad \quad \text{distPC}_{\omega, \omega'} := \text{sumDistPC}_{\omega} + \text{distPC}(\omega')
07 \quad \quad \text{distR}_{\omega, \omega'} := \text{sumDistR}_{\omega} + \text{distR}(\omega')
08 \quad end for
09 end for
10 while \( \Psi \neq \emptyset \) do
11 \quad sort the cells in \( \Psi \) based on \( \text{sumDistPC}_{\omega} \) by ascending order and based on \( \text{sumDistR}_{\omega} \) by descending order
12 \quad for each cell \( C(\omega) \) in the ordered \( \Psi \) do
13 \quad \quad \omega' := \text{select the instance which has minimum distPC with maximum distR in } C(\omega)
14 \quad \quad \text{add } \omega' \text{ to } W \text{ // add the instance to the sequence to be outputted}
15 \quad \quad C(\omega) := C(\omega) \setminus \{\omega'\}
16 \quad \quad \text{sumDistPC}_{\omega} := \text{sumDistPC}_{\omega} - \text{distPC}(\omega')
17 \quad \quad \text{sumDistR}_{\omega} := \text{sumDistR}_{\omega} - \text{distR}(\omega')
18 \quad \quad if \( C(\omega) = \emptyset \) then
19 \quad \quad \quad \text{\( \Psi := \Psi \setminus \{C(\omega)\} \)}
20 \quad \quad \quad \text{end if}
21 \quad \quad end for
22 \quad end while

#### Table 13
**Fisher’s LSD Test for Comparing Different Abstraction Levels with X3 under RandDelay**

<table>
<thead>
<tr>
<th>Y-Dimension</th>
<th>X1: O</th>
<th>X2: K</th>
<th>X3 (Control):</th>
<th>X4: S</th>
<th>X5: B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y1: {p, r, c}</td>
<td>&lt;</td>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>Y2: {p, r}</td>
<td>&lt;</td>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>Y3: {r, c}</td>
<td>&lt;</td>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>Y4: {p, c}</td>
<td>&lt;</td>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>Y5: {q}</td>
<td>&lt;</td>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>Y6: {r}</td>
<td>&lt;</td>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>Y7: {c}</td>
<td>&lt;</td>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>Overall</td>
<td>&lt;</td>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
</tbody>
</table>

#### Table 14
**Fisher’s LSD Test for Comparing Different Subspaces with Y7 under RandDelay**

<table>
<thead>
<tr>
<th>X-Dimension</th>
<th>Y1: O</th>
<th>Y2: K</th>
<th>Y3: S</th>
<th>Y4: S</th>
<th>Y5: B</th>
<th>Y6: K</th>
<th>Y7 (Control):</th>
</tr>
</thead>
<tbody>
<tr>
<td>X1: O</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>=</td>
</tr>
<tr>
<td>X2: K</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>=</td>
</tr>
<tr>
<td>X3: S</td>
<td>=</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>=</td>
</tr>
<tr>
<td>X4: S</td>
<td>=</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>=</td>
</tr>
<tr>
<td>X5: B</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>=</td>
</tr>
<tr>
<td>Overall</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>=</td>
</tr>
</tbody>
</table>

#### Table 15
**Fisher’s LSD Test for Comparing Individual Technique with Technique T(3, 7) under RandDelay**

<table>
<thead>
<tr>
<th>Combination</th>
<th>Y1: O</th>
<th>Y2: K</th>
<th>Y3: S</th>
<th>Y4: S</th>
<th>Y5: B</th>
<th>Y6: K</th>
<th>Y7 (Control):</th>
</tr>
</thead>
<tbody>
<tr>
<td>X1: O</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>= (Control)</td>
</tr>
<tr>
<td>X2: K</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>=</td>
</tr>
<tr>
<td>X3: S</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>=</td>
</tr>
<tr>
<td>X4: S</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>=</td>
</tr>
<tr>
<td>X5: B</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>=</td>
</tr>
<tr>
<td>Overall</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>=</td>
</tr>
</tbody>
</table>

Fig. 8. Overall comparisons among the untreated ordering (U), random ordering (R), and our family of techniques (F) under RandDelay.
rather than random strategy which is used in Algorithm 1. After that, Algorithm 2 visits these ordered cells one by one (line 12). For each cell $C(o)$ in $Ψ$, Algorithm 2 selects a suspicious instance $ω'$ from $C(o)$ by using distance-based strategy (line 13) (however Algorithm 1 will randomly select an instance $ω'$ from $C(o))$. The selected instance $ω'$ is added to the ordered sequence $W$ (line 14), and then is removed from $C(o)$ (line 15). The $\text{sumDistIPC}$ and $\text{sum-DistIR}$ of $C(o)$ are updated correspondingly (lines 16-17). If the cell $C(o)$ becomes empty, the algorithm removes $C(o)$ from $Ψ$ (lines 18-20). It then iterates until $Ψ$ is empty (lines 10-22).

Fig. 10 shows the comparison results of the two algorithms (Algorithm 1 in Section 3 and Algorithm 2 above). The overall result in the rightmost plot shows that Algorithm 1 performs a little better than Algorithm 2. We also conducted the ANOVA test on the overall result, which shows that Algorithm 1 and Algorithm 2 are statistically different at the 5% significance level. With regard to the results on each benchmark, we also find that on 6 benchmarks (i.e., LU, FFT, Aget, Barnes, Apache#1, and Apache#2), Algorithm 1 is more effective than Algorithm 2, and on 1 benchmark (MySQL#2), Algorithm 1 is as effective as Algorithm 2. For the rest three benchmarks (i.e., PBZIP2, Memcached, and MySQL#1), Algorithm 1 is less effective than Algorithm 2.

**Further Discussions**

In this section, we discuss other issues related to our work.

In Section 2.1, we presented four non-serializable patterns. In some other work such as PENELOPE [25], researchers also included the WWW pattern, which represents all three accesses writing to the same memory location. The WWW pattern was already discussed and shown in [15] that it is a serializable pattern rather than a non-serializable pattern. Specifically, the equivalent serial accesses of $\langle write_p, write_s, write_r \rangle$ is $\langle write_p, write_s, write_r \rangle$, where $write_p$ and $write_s$ are generated by the same thread and $write_r$ is generated by another thread. To verify our understanding further, we have also re-examined the work [25] presenting PENELOPE. We did not find any bugs claimed by that paper that can only be detected by the WWW pattern. Therefore, our work did not consider this pattern.

In Section 3, we have presented that Maple has three subspaces. In [29], the authors presented six idioms: three for single variable and three for multi-variable. Our work is a controlled experiment on single-variable atomicity violation. Thus, we extract three subspaces as shown in Fig. 1 based on the three idioms for single variable.

In Section 5, we have presented a comprehensive experiment. Some readers may want to know the results of adding the frequency $N$ to the instruction level $I$ in constructing a new form of abstraction level (denoted as $\langle I, N \rangle$). We take Fig. 6(b) for an indirect discussion. By adding $N$ to the level $K$ (i.e., level X2), we construct the object frequency abstraction $O$ (i.e., level X1). We have, $PC(X1) = 0.09$ versus $PC(X2) = 0.30$. The decrease is significant enough to bring down the curve for X1 in Fig. 6(c) and makes the curve for X1 much lower than the other curves in the same plot. We tend to believe that the level $\langle I, N \rangle$ results in a curve located much lower than that of $PC(X3)$. Moreover, as we have presented in Section 5.7, at different abstraction levels, the mean proportion $\Pi(Ψ)$ of bug instances in buggy cells (see Fig. 6(a)) are close to 1.0. We tend to consider that adding an intermediate level $\langle I, N \rangle$ has little chance to produce an exception as far as $\Pi$ is concerned. We thus tend to believe that the curve for $Pr(X3)$ is likely to locate higher than that for $Pr(\langle I, N \rangle)$ in Fig. 6(c).
In Section 5.5, we have presented that all bug instances exposed in TABLE 3 are referred to as different faults. The definition to classify different faults is described in Section 2.1. We had examined bug instances exposed in benchmarks LU, FFT, and Barnes. They were all caused by missing synchronous barriers in the programs. From this perspective, one may intuitively consider these bug instances as same faults in the programs. But the others may consider them as different bugs, even though a same fix can patch these bugs. Take the benchmark MySQL#2 as an example. The two bug instances of MySQL#2 reside in closely related code regions. One way to fix these bugs is to add a common lock to protect all these code regions. This strategy appears to be workable because it makes the code regions serializable. Nonetheless, the performance of the code will degrade significantly after applying such a fix. This fix can be considered as a bad/partial fix. Precisely defining and quantifying bug instances which refer to the same bug is another research topic out of our scope. Therefore we use the definition presented in Section 2.1 to classify whether multiple bug instances refer to the same fault or not.

In Section 5.5, the "Mean Time Spent" presented in TABLE 3 differs a lot from the time presented in the paper that proposed CTrigger (i.e., Park et al. [20], Table 4, column 6). The time presented in Park et al. [20] is the time spent to expose atomicity violation bugs. Their paper only showed the time spent (i.e., duration) until a targeted bug was exposed. On the other hands, the "Mean Time Spent" in our paper is the time spent to confirm all suspicious instances in one set of Ω. That is, we do not assume that after exposing a bug, there is no further bug in Ω. Thus, the scenarios described in Table 4 of Park et al. [20] and in TABLE 3 of our paper are different. Moreover, as mentioned in Section 1, CTrigger randomly selects one suspicious instance from each equivalence class for examination. Therefore the total number of suspicious instances confirmed by CTrigger is a subset of whole set Ω. Although the results of CTrigger in their paper interestingly showed that their technique was highly effective in exposing atomicity violation bugs, yet, as shown in Fig. 6, the proportion P(Φ) of bug instances in buggy cells is not always equal to 1. Such result indicates that the selection strategy used by CTrigger is likely to be efficient but it may miss some bugs. Since these bugs may crash programs in runtime, therefore our work focuses on confirming all suspicious instances in the set Ω, rather than only confirming a subset of all suspicious instances. So the time reported in Park et al. [20] must differ a lot from the time presented in TABLE 3 of our paper.

In Section 5.7, we have not discussed why a cell containing a buggy instance is less likely to contain non-buggy instances. In fact, it is an empirical observation, but we have not been able to iron out its reason, which warrants more research. Previous work such as LiteRace [34] has conjectured that real-world concurrency bugs often occur in the code regions that are seldom executed. We further recall that each basic block is a single-entry-single-exit program structure, which means that if one of the statements is seldom executed, the whole basic block is also seldom executed in the same execution. The above conjecture seems being in line with our empirical finding that for the set of events from instruction level to the basic block level, their involved pieces of code in an atomicity violation bug are seldom executed. It hints that the number of suspicious instances produced with respect to the same fault is likely small. We analyzed the data in our controlled experiment, and we found that the number of instances in a buggy cell was often 1, which agreed with this intuition.

Our work studies the detection of single-variable atomicity violation bugs by using a two-phase strategy. In fact, many other types of concurrency bugs including data races [13][23], deadlocks [5], atomicity-set serializability violations [14] can be detected through such two-phase strategy. In between the two phases, our work inserts a prioritization mechanism to arrange the suspicious instances. We believe that this approach can be further generalized. Moreover, each access event or locking event entails an execution context, and each suspicious instance for concurrency bug consists of at least two such events. Thus, the notions of abstraction levels and subspace projections can be applied in general, albeit that there may be other effective subspaces or abstraction levels to be found. For instance, multi-variable atomicity violations also involve three access events. Each event can be encoded with an execution context. We therefore believe that the set of such suspicious instances can still be partitioned based on our partitioning criteria and then prioritized. On the other hands, it is unclear to us whether the results observed along the abstraction level and subspace dimensions obtained in our controlled experiment can be generalized to cover the situations for multi-variable atomicity violations adequately. It is because in multi-variable atomicity violations, the events “e_{r}” and “e_{w}” are generated by different threads rather than by the same thread. Thus, the “hard” happens-before relation between events “e_{r}” and “e_{w}” in the single-variable atomicity violation scenarios is no longer maintained. This extra degree of freedom is likely to produce an additional degree of uncertainty on the characteristics observed on these two events and their relations with event “e_{r}”.

REFERENCES