To appear in 2015 IEEE International Conference on Software Quality, Reliability and Security (QRS 2015)

ASR: Abstraction Subspace Reduction for Exposing Atomicity Violation Bugs in Multithreaded Programs †

Shangru Wu
Department of Computer Science
City University of Hong Kong
Hong Kong
shangru.wu@my.cityu.edu.hk

Chunbai Yang
Department of Computer Science
City University of Hong Kong
Hong Kong
chunbyang2@gapps.cityu.edu.hk

W.K. Chan‡
Department of Computer Science
City University of Hong Kong
Hong Kong
wkchan@cityu.edu.hk

***Abstract*—Many two-phase based dynamic concurrency bug detectors predict suspicious instances of atomicity violation from one execution trace, and examine each such instance by scheduling a confirmation run. If the amount of suspicious instances predicted is large, confirming all these instances becomes a burden. In this paper, we present the *first* controlled experiment that evaluates the efficiency, effectiveness, and cost-effectiveness of reduction on suspicious instances in the detection of atomicity violations. A novel form of reduction technique named ASR is proposed. Our empirical analysis reveals many interesting findings: First, the reduced sets of instances produced by ASR significantly improve the efficiency of atomicity violation detection without significantly compromising the effectiveness. Second, ASR is significantly more cost-effective than random reduction and untreated reduction by 8.5 folds and 60.7 folds, respectively, in terms of mean normalized bug detection ratio. Third, six ASR techniques can be significantly more cost-effective than the technique modeled after a state-of-the-art detector.**

Keywords—atomicity violation, bug detection, reduction technique, vulnerability, failures, reduction ratio, detection ratio

I. INTRODUCTION

Multithreaded programs are more difficult to be developed correctly than sequential programs because of the additional thread interleaving dimension appearing in program behavior. Previous empirical studies (i.e., [13]) have shown that many real-world multithreaded programs contain concurrency bugs, which include deadlocks [2][3][4], data races [8][21], as well as atomicity and linearizability violations [7][11][12][17][23].

The manifestation of many atomicity violation bugs involves three memory accesses of two threads to the same memory location [12][18], which are referred to as *single-variable atomicity violation* [14]. For ease of presentation, we refer to the three involved memory access events of a single-variable atomicity violation as e_p, e_c, and e_r, where e_p, e_c, and e_r are called preceding-access event, current-access event, and remote-access event, respectively [18].

Suppose that in an execution trace of a faulty multithreaded program, with respect to a specific memory location, one thread consecutively executes events e_p and e_c, and another thread executes event e_r. Further suppose that the access sequence of these three events is either $\langle e_p, e_c, e_r \rangle$ or $\langle e_r, e_p, e_c \rangle$, and the execution trace does not reveal a failure. If there is another execution trace of the same program over the same input such that following the permuted access sequence $\langle e_p, e_r, e_c \rangle$ results in a program failure (e.g., a crash or an assertion violation in test oracles), then we say that there is a single-variable atomicity violation. More precisely, the atomicity formed by e_p and e_c is said to have been violated by e_r [14]. The interleaving sequence $\langle e_p, e_r, e_c \rangle$ that exposes an atomicity violation bug is denoted as a *non-serializable interleaving* [14].

Many existing dynamic concurrency bugs detectors [11][18][30] implement a two-phase strategy to detect atomicity violations. In the predictive phase of such a two-phase strategy, such a detector analyzes an execution trace to find *suspicious* instances according to the non-serializable interleaving patterns [12][18]. For example, in the above example, the access sequence $\langle e_p, e_c, e_r \rangle$ or $\langle e_r, e_p, e_c \rangle$ is observed by the detector, and a permuted access sequence $\langle e_p, e_r, e_c \rangle$ is ***predicted*** as a ***suspicious instance*** (e.g., according to whether the pair e_c and e_r and the pair e_p and e_r, respectively, can be executed concurrently based on the happened-before relation in the observed trace). In the confirmation phase, for each given suspicious instance $\langle e_p, e_r, e_c \rangle$, the detector schedules a *confirmation run* over the same input with an attempt to trigger the access sequence $\langle e_p, e_r, e_c \rangle$. If the execution crashes or violates an assertion in the test case after the access sequence $\langle e_p, e_r, e_c \rangle$ is exercised, then the detector is said to have exposed an atomicity violation bug, and the corresponding suspicious instance is referred to as a *bug instance* [28] because the instance pinpoints the location of a concurrency bug.

Such a two-phase strategy is sound by only reporting harmful bugs. But, for real-world and large-scale multithreaded programs, a large number of suspicious instances may be predicted from a single trace. The examination of each suspicious instance requires one confirmation run. Such a confirmation run may incur 3 to 100 folds of slowdown compared to the native run of the program [30]. Examining all suspicious instances would result in the consumption of overwhelming computation resources. To allow developers to debug more efficiently, the previous work [28] has studied how to effectively prioritize all these suspicious instances before the confirmation phase.

In the cloud computing era, cloud users have the option to pay for their actual usages of the cloud resources. Nonetheless, prioritization does not save any computations and resources consumed in the confirmation phase.

We thus ask an important question: How to select a highly effective and small subset of suspicious instances for

† This research is supported in part by the General Research Fund of Research Grants Council of Hong Kong (project nos. 111313, 123512, 125113, and 11201114).

‡ W.K. Chan is the contact author of this work.

examination in the confirmation phase? In this paper, we study this question through an empirical study.

Each suspicious instance is a composition of three events. Each event is associated with an execution context [3]. An exemplified kind of execution context of an event is the call stack of the event plus the instruction (i.e., machine code) that generates the corresponding event in the execution trace.

Our previous study [28] models from the test case selection perspective that two suspicious instances can be regarded as equivalent if they are the same at a specified *abstraction level* (e.g., same statement) after projecting onto a specified *subspace* (e.g., only considering the execution context of the event e_p). That is, given a pair of abstraction level and subspace, the set of all suspicious instances can be partitioned into a set of equivalence classes. The instances in each equivalence class are considered as equivalent. The previous study further reveals that after partitioning, an equivalence class containing a bug instance (referred to as a *buggy equivalence class*) has a high probability to only contain buggy instances [28]. A practical implication of the result is that even though we do not know which equivalence class is buggy, we may infer that if a buggy equivalence class is picked, then a bug instance in the buggy equivalence class is likely to be examined.

Therefore, we refine the above research question further to ask the following: after partitioning, is it effective to expose atomicity violation bugs if we only examine one suspicious instance per equivalence class? Moreover, to what extent such reduction can be cost-effective?

In this paper, we report a large-scale controlled experiment to study the influence of the above two design factors: the abstraction level of execution context and the subspace after dimensional projection, on the reduction of suspicious instances. Following the previous study [28], we examined five abstraction levels and seven subspaces. For each combination of abstraction level and subspace, an ***A***bstraction-***S***ubspace-***R***eduction (***ASR*** for short) technique is synthesized. ASR firstly partitions a set of suspicious instances into equivalence classes. Then for each equivalence class, ASR randomly[1] selects one suspicious instance from the class for examination. We measure the efficiency, effectiveness, and cost-effectiveness of ASR on the detection of atomicity violation bugs through the instance reduction ratio, the bug detection ratio, and the normalized detection ratio, respectively, using a set of 10 multithreaded programs with linearizability-related bugs.

The findings show that ASR techniques are significantly more cost-effective than both random reduction by 8.5 folds and untreated reduction by 60.7 folds in terms of mean normalized detection ratio. The findings also show that (1) using one-dimensional subspaces (Y5–Y7) achieves a significantly higher reduction ratio than using two-dimensional subspaces (Y2–Y4), which in turn achieves a significantly higher reduction ratio than the original three-dimensional space (Y1). (2) Using a coarse-grained abstraction level (X4–X5) results in significantly higher reduction ratio than using a fine-grained abstraction level (X1–X3). (3) Interestingly, a higher reduction ratio does not necessarily lead to a lower detection ratio for atomicity violation detection. The overall finding shows that ASR can make the detection of atomicity violation bugs efficient and maintain the effectiveness of bug detection.

[1] Other selection strategies rather than random can be used, we leave the study of other strategies in the future work.

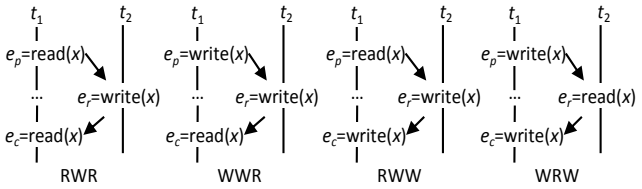

Fig. 1. Patterns of non-serializable interleaving.

The main contribution of this paper is threefold. First, it is the *first* work to study the efficiency, effectiveness, and cost-effectiveness of reduction techniques for the detection of atomicity violation bugs in a systematic manner. Second, it proposes a novel form of reduction technique: *ASR*. Third, it presents the evidence that ASR techniques on the intersection of coarse-grained abstraction levels (X4 and X5) and one-dimensional subspace (Y5–Y7) are promising candidates to generate a systematic, highly effective and small subset of suspicious instances for follow-up confirmation. These techniques offer a good performance-effectiveness tradeoff in assisting developers to locate concurrency bugs.

The rest of the paper is organized as follows. Section II reviews the background of this work. Section III introduces the reduction strategy. Section IV presents the controlled experiment and the results. Section V reviews the closely related work, followed by the conclusion in Section VI.

II. BACKGROUND

A. Non-serializable Interleaving

A non-serializable interleaving that exposes a single-variable atomicity violation involves three memory access events, labelled as e_p (preceding event), e_c (current event), and e_r (remote event) [12][18]. Specifically, e_p and e_c issued by one thread consecutively visit a memory location, say x, and e_r, issued by another thread, visits x in between e_p and e_c.

In previous work [12], four non-serializable memory accesses patterns are identified to capture non-serializable interleaving. The four patterns are illustrated in Fig. 1: *RWR* means that, in between two consecutive reads on a memory location x by thread t_1, thread t_2 writes to x. The other patterns can be interpreted similarly. (Note that *WWW* is not a non-serializable memory accesses pattern [12].)

B. Suspicious Instances and Bug Instances

A typical prediction-confirmation strategy is adopted by many dynamic atomicity violation detectors [11][18][30]. In the prediction phase, an execution trace is analyzed by such a detector to predict whether there is any access sequence which may manifest into one of the access patterns presented in Fig. 1. In this paper, we refer to such a predicted access sequence as a **suspicious instance**, or an **instance** for short.

Algorithm 1: Reduction Strategy of ASR

Input: $\Omega := \{\omega_1, ..., \omega_k, ..., \omega_m\}$: a set of suspicious instances
Input: X: an abstraction level
Input: Y: a subspace
Output: R : a reduced set of suspicious instances, initially empty
01 Partition Ω into a set Ψ of equivalence classes
  $\Psi := \pi(\Omega, X, Y) = \{C(\omega_1)..., C(\omega_i)..., C(\omega_t)\}$
// reduction
02 **for each** equivalence class $C(\omega)$ **in** the set Ψ **do**
03 | $\omega' :=$ randomly select an instance from $C(\omega)$
04 | $R := R \cup \{\omega'\}$ // add the instance to the reduced set R
05 **end for**

The set of all suspicious instances detected in the prediction phase is denoted by $\boldsymbol{\Omega} = \{\omega_1, ..., \omega_k, ..., \omega_m\}$, where each instance $\boldsymbol{\omega_k} = \langle e_{pk}, e_{rk}, e_{ck} \rangle$ conforms to one of the four above-mentioned non-serializable memory access patterns.

Each suspicious instance ω may or may not manifest in a feasible execution trace. Thus, in the confirmation phase, the detector schedules a program execution to try to confirm the existence of the detected suspicious instance in the real execution trace of the program. For a suspicious instance ω, if ω manifests in one confirmation execution trace and also incurs either a crash or an assertion violation, we refer to ω as a ***bug instance***.

C. Equivalence Class Partitioning

Following [28], we also consider two design factors in partitioning, namely ***abstraction level*** and ***subspace***.

Given an abstraction level X, the corresponding abstraction of an event e can be retrieved, which is denoted as $abs(e, X)$. For instance, the abstraction $abs(e, I)$ of an event e under the instruction level I refers to the program instruction (i.e., machine code) generating e. To partition the set of suspicious instances Ω, each instance is abstracted into a certain abstraction level. Different instances which are regarded as equivalent at the same abstraction level are placed in the same equivalence class.

Each suspicious instance $\omega = \langle e_p, e_r, e_c \rangle$ can be formulated as a point in a 3-dimensional space $\{p, r, c\}$. This 3-dimensional space can be projected onto 1-dimensional subspaces and 2-dimensional subspaces. 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 ω to its e_c component.

Two instances $\omega_1 = \langle e_{p1}, e_{r1}, e_{c1} \rangle$ and $\omega_2 = \langle e_{p2}, e_{r2}, e_{c2} \rangle$, are said to be ***equivalent*** under an abstraction level X and a subspace Y (denoted as $[\omega_1 \approx \omega_2]_{X,Y}$) if ω_1 and ω_2 share the same abstraction at the abstraction level X for each event specified by the subspace Y. For instance, suppose X = I and Y = $\{p, r\}$, ω_1 and ω_2 are equivalent iff $abs(e_{p1}, I) = abs(e_{p2}, I)$ and $abs(e_{r1}, I) = abs(e_{r2}, I)$, then, denoted as $[\omega_1 \approx \omega_2]_{I,\{p, r\}}$.

Under a given abstraction level X and a subspace Y, the set Ω is partitioned into a set of *disjoint* equivalence classes $\Psi = \pi(\Omega, X, Y) = \{C(\omega_1)..., C(\omega_i)..., C(\omega_t)\}$ such that all suspicious instances in each equivalence class are equivalent, i.e., for each equivalence class $C(\omega) \in \Psi$, $\forall \omega_m, \omega_n \in C(\omega)$, $[\omega_m \approx \omega_n]_{X,Y}$.

III. *A*BSTRACTION-*S*UBSPACE-*R*EDUCTION

A. Reduction-Confirmation Strategy

The basic idea of our reduction-confirmation (ASR) strategy is as follows. For each given pair of an abstraction level and a subspace, we firstly partition a set Ω into a set of equivalence classes. We then select one suspicious instance per equivalence class, and examine each selected instance in the confirmation run.

Algorithm 1 implements the above strategy. The algorithm accepts the following inputs: a set Ω, an abstraction level X, and a subspace Y. At the beginning, the algorithm partitions Ω into a set Ψ of equivalence classes (line 01). After the construction of equivalence classes, the algorithm goes into the process of reduction. It goes through these equivalence classes one by one (lines 02–05). For each visited equivalence class C(ω) in Ψ, the algorithm randomly selects an suspicious instance ω′ from C(ω), and adds ω′ to the reduced set R (lines 03–04). After all equivalence classes are visited, the algorithm produces a reduced set R of suspicious instances and will send this set R to the confirmation phase for examinations.

Note that the reduction process can be performed on the fly. That is, unlike prioritization over Ω, which should wait till the generation of a set of suspicious instances, reduction can be generally performed on the fly once an instance is predicted. For instance, adapting Algorithm 1 with a hash table could make the online reduction available: Each time Algorithm 1 receives a predicted instance ω, it calculates the hash value of ω based on the given abstraction level X and subspace Y. If the hash value of ω matches some value in the hash table of Algorithm 1, then this instance ω can be considered equivalent with some previous instance sent to the subsequent confirmation phase before. As such, Algorithm 1 performs the reduction on ω online, whereas, the process of prioritization can only be performed offline. As such, ASR can improve the performance of concurrency bug detection.

B. ASR Dimensions and Techniques

Following [28], we use the below five abstraction levels and seven subspaces to initialize Algorithm 1 to produce our ASR techniques.

1) Abstraction Level (X-Dimension)

The five abstraction levels studied are **o**bject frequency abstraction level (***O***) [3], ***k***-object sensitivity level (***K***) [11], **i**nstruction level (***I***) [14], **s**tatement level (***S***) [28], and **b**asic block level (***B***) [28].

These five levels are increasingly more abstract. That is, each basic block may contain multiple statements; each statement may include a set of instructions (i.e., machine code); each instruction may be associated with different call stack instances during execution; and call stack instances with same call stack structure can still be distinguished by their occurrence orders. We refer to the abstraction levels *O*, *K*, *I*, *S*, and *B* as **X1**, **X2**, **X3**, **X4**, and **X5**, respectively.

Moreover, one statement or basic block may include more than one instruction. Whereas, an instruction is actual

piece of code that performs an event, and the other two levels are built on top of the instruction level. We refer to the statement and basic block levels as *coarse-grained* abstraction levels and the other three as *fine-grained* ones [28].

2) Subspace (Y-Dimension)

The seven subspaces studied include all the non-empty subsets of the component set {*p*, *r*, *c*} used to project the original three-dimensional space. We refer to the subspaces {*p*, *r*, *c*}, {*p*, *r*}, {*r*, *c*}, {*p*, *c*}, {*p*}, {*r*}, and {*c*} as **Y1**, **Y2**, **Y3**, **Y4**, **Y5**, **Y6**, and **Y7**, respectively. Specifically, subspaces Y5–Y7, Y2–Y4, and Y1 contain one element, two elements, and three elements, respectively. Thus, we refer to Y5–Y7, Y2–Y4, and Y1 as *one-dimensional subspaces*, *two-dimensional subspaces*, and the *original three-dimensional space*, respectively.

3) ASR Techniques

Using each combination of the above abstraction levels and subspaces to initialize Algorithm 1, we generate 35 ASR techniques in total. Each technique takes a set Ω of suspicious instances as input and produces a reduced set *R* for confirmation. Each ASR technique corresponds to one unique coordinate on a 2-dimension Euclidean plane with X (i.e., abstraction level) and Y (i.e., subspace) being the two axes. Thus, we simply use ASR(X_i, Y_j) to denote a ASR technique which adopts the combination of abstraction level X_i and subspace Y_j.

In our experiment, we also include two controlled techniques. The first one is the reduction by the ***random*** strategy. That is, given the number *N* of suspicious instances requested in a reduced set *R*, *Random* reduction randomly selects *N* instances from the set Ω of all suspicious instances to construct set *R*. The second one is the reduction by the ***untreated*** strategy. The *Untreated* reduction only selects the first *N* suspicious instances by their occurrence orders in the prediction phase of the detector to construct set *R*.

IV. Controlled Experiment

A. Research Questions

We study the following three new research questions:

- **RQ1**: To what extent do *abstraction level* and *subspace* affect the efficiency of ASR techniques on the examination of suspicious instances?
- **RQ2**: To what extent do *abstraction level* and *subspace* affect the effectiveness of ASR techniques on the detection of atomicity violations?
- **RQ3**: Compared to the controlled techniques, to what extent can ASR techniques additionally offer on the detection of atomicity violations?

B. Independent Variables

Our controlled experiment has two independent variables, which are the two design factors of ASR techniques: *abstraction level* and *subspace*. Five abstraction levels and seven subspaces are studied in the experiment. Each pair of abstraction level X_i and subspace Y_j generates an ASR technique from Algorithm 1, and we refer to each technique as ASR(X_i, Y_j).

C. Dependent Variables and Measures

To measure the *efficiency* of a reduction technique, we use the metric *reduction ratio* [10][20], which evaluates the extent of the reduction on the set *R* against Ω. A higher reduction ratio indicates a higher efficiency on the examinations of suspicious instances by using the set *R*. The formula of reduction ratio is defined as follows [10][20]:

$$Reduction\ Ratio = 1 - \frac{|R|}{|\Omega|}$$

In the formula, |Ω| means the total number of suspicious instances in the set Ω, and |*R*| means the number of suspicious instances (to be examined) in the reduced set *R* after reduction.

To measure the *effectiveness* of a reduction technique, we compute the bug detection ratio of the reduced set *R* produced by the technique. The detection ratio is the ratio of the number of bug instances in *R* to the total number of bug instances in Ω. A higher detection ratio indicates higher effectiveness on the detection of bugs by using the reduced set *R*. The formula of detection ratio is defined as follows:

$$Detection\ Ratio = \frac{d_R}{d}$$

In the formula, *d* is the total number of bug instances in the given set Ω, and d_R is the number of bug instances included by the reduced set *R*.

To consider the efficiency and effectiveness of a reduction technique together, we further measure the *cost-effectiveness* of a reduction technique. We evaluate the detection ratio per suspicious instance in the reduced set *R*, and denote this metric as normalized detection ratio (NDR in short). The formula of NDR is defined as follows:

$$NDR = \frac{d_R}{d} \Big/ \frac{|R|}{|\Omega|} = \frac{Detection\ Ratio}{1 - Reduction\ Ratio}$$

The baseline value of NDR is 1, which means that if a reduced set *R* only contains *k* percentage of all suspicious instances (i.e., |*R*| / |Ω| = *k*), then the baseline ability of this set *R* should detect *k* percentage of bugs (i.e., d_R / *d* = *k*). A higher NDR means higher cost-effectiveness of a reduced set on the detection of bugs.

D. Benchmarks

We performed the experiment on a set of 10 benchmarks, which are widely used in the research studies on the concurrency bug detection [14][30][32]. TABLE 1 shows the descriptive statistics of these benchmarks. PBZIP2 and Aget are utility programs. LU, FFT, and Barnes are scientific programs from Splash2 [27]. Memcached, Apache, and MySQL are real-world server programs.

The version or the bug report identifier of each benchmark is shown in the third column of TABLE 1, if any (where "−" means that no official bug report identifier is available). The fourth column lists the benchmark sizes in SLOC [22]. The fifth column shows the number of threads in the executions of the benchmarks. Note that we obtained the test inputs of Memcached, Apache, and MySQL from

TABLE 1. DESCRIPTIVE STATISTICS OF BENCHMARKS

Benchmark	Type	Version / Bug ID	Size in SLOC	# of Thds	# of R/W Events*	# of Bug Instances
PBZIP2	Utility	0.94	1,500	4	400	2
LU	Scientific	–	1,700	2	4,300	3
FFT	Scientific	–	1,700	2	1,500	5
Aget	Utility	0.4.1	1,900	4	12,000	1
Barnes	Scientific	–	4,500	2	42,000	4
Memcached	Server	127	24,000	4	8,700	1
Apache#1	Server	25520	240,000	28	3,100,000	1
Apache#2	Server	21287	270,000	28	4,300,000	1
MySQL#1	Server	791	620,000	14	3,000,000	1
MySQL#2	Server	3596	650,000	15	10,000,000	2

* The read / write memory access events for counting only contain non-stack memory operations (e.g., shared memory access events).

previous work [18][30]. We reused the test inputs we constructed in [28] for the remaining utility programs and scientific programs. The sixth column reports the mean number of read/write memory access events (on shared memory locations) in the execution traces of the benchmarks. The last column is the number of bug instance exposed (see Section IV.E). Specifically, these bugs expose vulnerability issues or program failures in the corresponding benchmarks. All bug instances refer to different faults [28].

E. Experimental Setup

Our experiment was performed on the Ubuntu Linux 12.04 x86_64 configured with four 2.90GHz Xeon Cores, 14.3GB physical memory and GCC 4.4.3 compiler.

We implemented our test framework on top of the infrastructure of the Maple tool [30], which is a dynamic analysis framework built on top of Pin [15]. Our framework extended the Maple tool to compute the object frequency abstraction [3] of each memory access event in an execution trace. We used the default setting in [3] to collect the call stack fragment of each object frequency abstraction. We further collected the abstractions of each event at other studied abstraction levels along the execution trace.

In Section II, we have mentioned how two-phase techniques expose atomicity violations. In the predictive phase, our framework monitored all shared memory accesses, and wrapped the Pthread library functions to collect happen-before relations and locksets of memory access events. It reported a suspicious instance $\langle e_p, e_r, e_c\rangle$ if there exists three events e_p, e_c, and e_r such that the following four conditions are satisfied: (1) Their access sequences is $\langle e_p, e_c, e_r\rangle$ or $\langle e_r, e_p, e_c\rangle$. (2) The permuted sequence $\langle e_p, e_r, e_c\rangle$ matches a pattern shown in Fig. 1. (3) There is no synchronization order between e_r and e_p (for the case $\langle e_r, e_p, e_c\rangle$) or between e_c and e_r (for the case $\langle e_p, e_c, e_r\rangle$). And, (4) e_p, e_c and e_r are not mutually exclusive [14].

In each predictive run, our framework detected a set Ω of suspicious instances from the corresponding execution trace. We repeated the prediction procedure 30 times for each benchmark and predicted a total of 300 sets of Ω for all 10 benchmarks. Fig. 2 shows the number of suspicious instances reported in the predictive phase through boxplots. The *x*-axis lists the benchmarks and the *y*-axis lists the corresponding numbers of predicted suspicious instances. From Fig. 2, we observe that in large-scale programs (e.g., MySQL), the number of suspicious instances predicted may exceed 10000, which is large. We recall that each confirmation run usually slows down the native run by one to two orders of magnitude [30]. Reducing the number of suspicious instances for confirmation reduces the incurred computational costs, which is attractive.

For each set Ω of suspicious instances, we applied each ASR technique 100 times to generate 100 reduced sets of suspicious instances. For the two controlled techniques (i.e., random reduction and untreated reduction), we obtained the number of suspicious instances $|R|$ from each reduced set R produced by each ASR technique, and used $|R|$ to guide the reduction of these two controlled techniques. Specifically, the random reduction randomly selects suspicious instances from the whole set Ω until reaching the given number $|R|$. The untreated reduction selects suspicious instances one by one by their occurrence order until reaching the given number $|R|$. As a result, for each reduced set R produced by each ASR technique, there were two other reduced sets produced by the two controlled techniques, respectively. And each such reduced set contained the same number of suspicious instances as the set R.

To evaluate the detection ratio of each reduced set, we firstly ran the confirmation phase for all the suspicious instances in each set Ω. In the course of confirmation of each given instance $\langle e_p, e_r, e_c\rangle$, if a thread had firstly executed event e_p and was going to execute event e_c, then our test framework suspended the thread to wait till event e_r had executed, producing the non-serializable interleaving $\langle e_p, e_r, e_c\rangle$. Similarly, if a thread had firstly executed event e_r, then our test framework suspended that thread until some other thread both had executed event e_p and was about to execute event e_c. After that, the former thread was resumed to have executed e_r before the latter thread executed e_c. At each suspension point, following CTrigger [14], we set up a timeout threshold. The timeout threshold we used in the

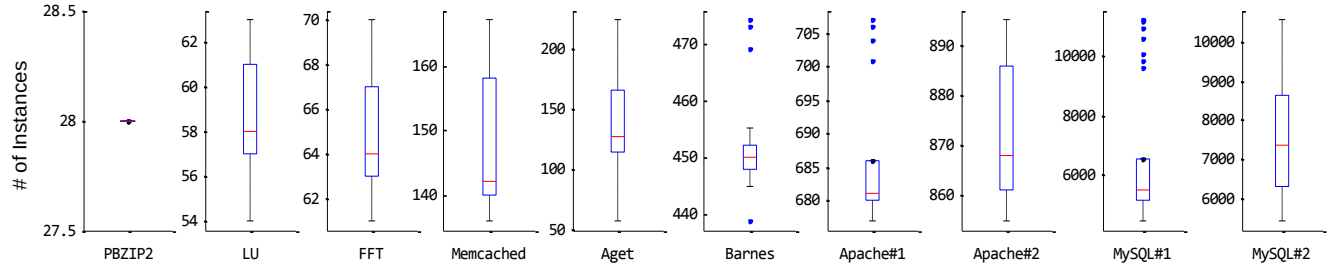

Fig. 2. The number of suspicious instances reported on each benchmark.

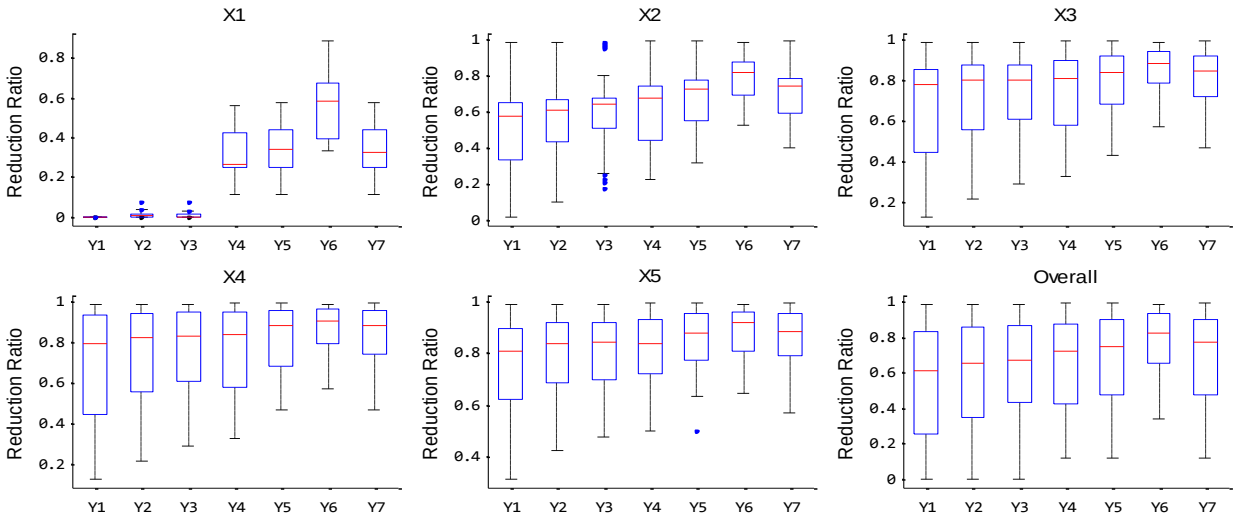

Fig. 3. The reduction ratios of the 35 ASR techniques as well as the overall results.

experiment was 1000 milliseconds, which consisted of 20 sleeps each with delay of 50 milliseconds [28].

If the above attempt to confirm a suspicious instance resulted in a non-serializable interleaving, and the execution crashed or violated an assertion in the test case after the interleaving was observed, we labeled the suspicious instance as a *bug instance*. With all suspicious instances labeled, we then simulated the confirmation phase by directly computing the detection ratio of each reduced set. As such, we collected 3000 detection ratios per technique on each benchmark (which exceeds the minimum threshold (1000) stated in a practical guideline for techniques involving random factor [1]).

F. Data Analysis

1) Answering RQ1

In this section, we analyze data along the dimension of abstraction level and the dimension of subspace to study the influence of these two design factors on the reduction ratios of ASR techniques.

We follow [28][31] to present the results at the aggregated level. The results of all the 35 techniques are shown in Fig. 3. In the figure, from left to right, there are six plots: one for each level of X as well as the overall results. Take the plot entitled X1 for example. There are seven bars in the plot. From left to right, they refer to the techniques ASR(1, 1), ASR(1, 2), ASR(1, 3), ASR(1, 4), ASR(1, 5), ASR(1, 6), and ASR(1, 7), respectively. The *y*-axis of the plot is the reduction ratio of the corresponding technique. Other plots in Fig. 3 can be interpreted similarly.

The bottom-right plot in Fig. 3 shows the overall results. That is, each bar represents the reduction ratios of all techniques sharing the same *y*-value irrespective to their *x*-values. From this plot, we observe that overall speaking, in terms of median value, techniques at Y6 achieve the highest reduction ratio, followed by techniques at Y7 then Y5. Their bars are located observably higher than techniques at Y2, Y3 and Y4 on the plot. Techniques at Y1 achieve the lowest reduction ratio. Similarly, in each of the remaining five plots (i.e., the plots for X1−X5), in terms of median value, Y6 also achieves the highest reduction ratio among all subspaces in that plot; Y5 and Y7 achieve observably higher reduction ratios than Y2, Y3 and Y4; and Y1 achieves the lowest reduction ratio. That is, in terms of median reduction ratio, one-dimensional subspaces are observably more efficient than two-dimensional subspaces, which in turn are observably more efficient than the original space. Among all subspaces, Y6 is the most efficient one.

We have further performed the Fisher's LSD test [26] to compare these techniques along the Y dimension at the 5% significance level using MatLab. TABLE 2 summarizes the results. In the table, if the test shows that a group of techniques (at the same X level with different *y* values) are not significantly different among each other, we assign the same grouping letter to them. TABLE 2 also shows the mean

TABLE 2. THE MEAN REDUCTION RATIO OF EACH TECHNIQUE AND THE FISHER'S LSD TEST RESULTS FOR COMPARING DIFFERENT Y LEVELS

	Y1 {*p, r, c*}	**Y2** {*p, r*}	**Y3** {*r, c*}	**Y4** {*p, c*}	**Y5** {*p*}	**Y6** {*r*}	**Y7** {*c*}
X1: *O*	0.00 (A)	0.01(A)	0.01 (A)	0.31 (B)	0.33 (C)	0.57 (**D**)	0.32 (BC)
X2: *K*	0.54 (A)	0.58 (B)	0.61 (BC)	0.63 (C)	0.68 (D)	0.78 (**E**)	0.70 (D)
X3: *I*	0.66 (A)	0.71 (B)	0.73 (B)	0.73 (B)	0.78 (C)	0.84 (**D**)	0.79 (C)
X4: *S*	0.70 (A)	0.74 (B)	0.76 (B)	0.76 (B)	0.81 (C)	0.85 (**D**)	0.82 (CD)
X5: *B*	0.75 (A)	0.78 (B)	0.80 (B)	0.80 (B)	0.83 (C)	0.87 (**D**)	0.85 (CD)
Mean	0.53 (A)	0.56 (B)	0.58 (B)	0.65 (C)	0.69(D)	0.78 (**E**)	0.70 (D)
Mean	0.53	0.60			0.72		

TABLE 3. THE FISHER'S LSD TEST RESULTS FOR COMPARING DIFFERENT X LEVELS IN TERMS OF REDUCTION RATIO

	X1: *O*	**X2: *K***	**X3: *I***	**X4: *S***	**X5: *B***
Y1: {*p, r, c*}	0.00 (A)	0.54 (B)	0.66 (C)	0.70 (D)	0.75 (**E**)
Y2: {*p, r*}	0.01 (A)	0.58 (B)	0.71 (C)	0.74 (D)	0.78 (**E**)
Y3: {*r, c*}	0.01 (A)	0.61 (B)	0.73 (C)	0.76 (D)	0.80 (**E**)
Y4: {*p, c*}	0.31 (A)	0.63 (B)	0.73 (C)	0.76 (C)	0.80 (**D**)
Y5: {*p*}	0.33 (A)	0.68 (B)	0.78 (C)	0.81 (D)	0.83 (**D**)
Y6: {*r*}	0.57 (A)	0.78 (B)	0.84 (C)	0.85 (CD)	0.87 (**D**)
Y7: {*c*}	0.32 (A)	0.70 (B)	0.79 (C)	0.82 (D)	0.85 (**E**)
Mean	0.22 (A)	0.65 (B)	0.75 (C)	0.78 (D)	0.81 (**E**)
Mean	0.54			0.79	

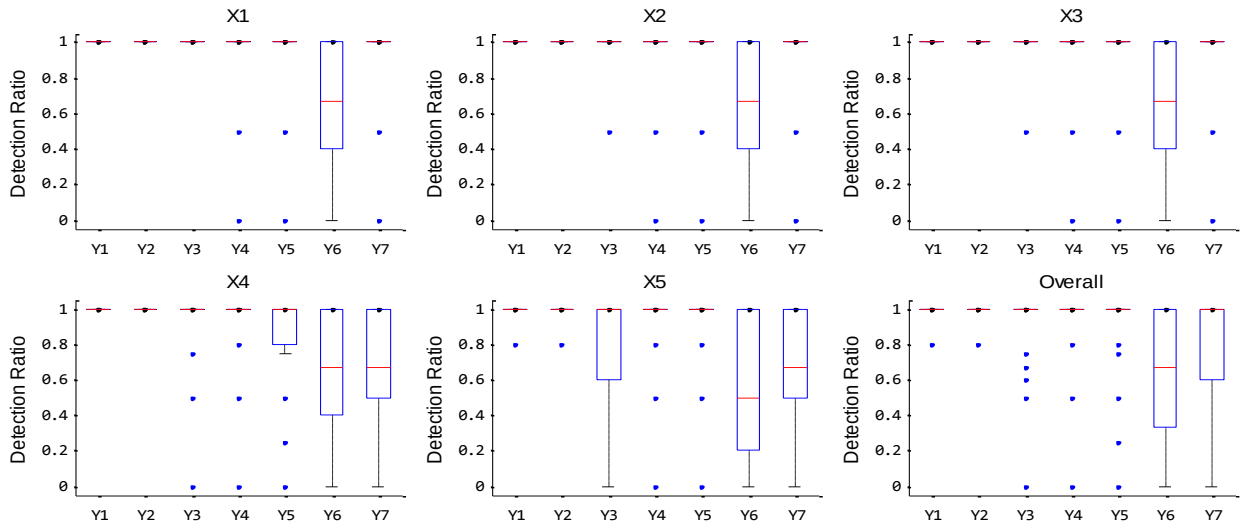

Fig. 4. The detection ratios of the 35 ASR techniques as well as the overall results.

reduction ratio of each technique, where if a technique has a higher mean reduction ratio, then we assign an alphabetically higher letter to it.

The results in TABLE 2 consolidate our observation stated above that one-dimensional subspaces (Y5–Y7) are significantly more efficient than two-dimensional subspaces (Y2–Y4), which in turn are significantly more efficient than the original space (Y1). Among all subspaces, Y6 achieves the highest reduction ratio with a mean value of 78%.

Similarly, for the analysis along the X dimension, we compare these techniques (at the same Y level with different *x* values) using Fisher's LSD test at 5% significance level. The results are shown in TABLE 3. This table can be interpreted similarly to TABLE 2. (We note that all 35 techniques have been shown in Fig. 3 and their mean reduction ratios are the same as the values in TABLE 2.)

From TABLE 3, we observe that overall speaking, X5 is the most efficient one among all abstraction levels; X4 is more efficient than X3; and X1 is the least efficient one followed by X2. The result shows that the use of coarse-grained abstraction levels (X4−X5) achieves significantly higher reduction ratios in a statistically meaningful way.

To answer RQ1, we find that the use of one-dimensional subspaces is more efficient than the use of other subspaces. Also, the use of the subspace {*r*} (i.e., Y6) achieves the highest reduction ratio among all subspaces. In terms of abstraction level, the use of a coarse-grained abstraction level is more efficient than the use of a fine-grained abstraction level. Moreover, the use of the basic block abstraction level (i.e., X5) achieves the best result with a mean reduction ratio of 81%.

2) Answering RQ2

We also apply the same data analysis methodology presented in the last section to study the influence of each design factor on the detection ratios of ASR techniques.

Fig. 4 shows the aggregated detection ratios of all the 35 techniques. From the overall results shown in the bottom-right plot of Fig. 4, we observe that, the median detection ratios of Y1–Y5 and Y7 are all 100%. The median detection ratio of Y6 is the lowest. In each of the remaining plots, the median detection ratios of Y1–Y5 are still 100%, irrespective to their abstraction levels. Such results indicate that in most of cases at Y1–Y5, a reduced set *R* is still able to contain all bug instances. That is, the effectiveness of ASR techniques for bug detection at Y1–Y5 is not compromised with an increase in efficiency, which is desirable for the purpose of fault detection. For Y7, the median detection ratio can attain 100% only at X1–X3. Although Y6 achieves the highest reduction ratio, the detection ratio of Y6 at any abstraction level is the lowest.

The LSD test results along the Y dimension at the 5% significance level are shown in TABLE 4. Similar to TABLE 2, we assign the same grouping letter to the techniques (at

TABLE 4. THE MEAN DETECTION RATIO OF EACH TECHNIQUE AND THE FISHER'S LSD TEST RESULTS FOR COMPARING DIFFERENT Y LEVELS

	Y1 {*p*, *r*, *c*}	**Y2** {*p*, *r*}	**Y3** {*r*, *c*}	**Y4** {*p*, *c*}	**Y5** {*p*}	**Y6** {*r*}	**Y7** {*c*}
X1: *O*	1.00 (C)	1.00 (C)	1.00 (C)	0.92 (B)	0.92 (B)	0.66 (A)	0.92 (B)
X2: *K*	1.00 (E)	1.00 (E)	0.95 (D)	0.92 (C)	0.92 (C)	0.65 (A)	0.84 (B)
X3: *I*	1.00 (E)	1.00 (E)	0.95 (D)	0.92 (C)	0.92 (C)	0.65 (A)	0.84 (B)
X4: *S*	1.00 (F)	1.00 (F)	0.87 (D)	0.90 (E)	0.86 (C)	0.64 (A)	0.69 (B)
X5: *B*	0.98 (F)	0.98 (F)	0.81 (C)	0.90 (E)	0.87 (D)	0.55 (A)	0.68 (B)
Mean	1.00 (F)	1.00 (F)	0.92 (E)	0.91 (D)	0.90 (C)	0.63 (A)	0.80 (B)
Mean	1.00	0.94			0.78		

TABLE 5. THE FISHER'S LSD TEST RESULTS FOR COMPARING DIFFERENT X LEVELS IN TERMS OF DETECTION RATIO

	X1: *O*	**X2: *K***	**X3: *I***	**X4: *S***	**X5: *B***
Y1: {*p*, *r*, *c*}	1.00 (B)	1.00 (B)	1.00 (B)	1.00 (B)	0.98 (A)
Y2: {*p*, *r*}	1.00 (B)	1.00 (B)	1.00 (B)	1.00 (B)	0.98 (A)
Y3: {*r*, *c*}	1.00 (D)	0.95 (C)	0.95 (C)	0.87 (B)	0.81 (A)
Y4: {*p*, *c*}	0.92 (B)	0.92 (B)	0.92 (B)	0.90 (A)	0.90 (A)
Y5: {*p*}	0.92 (C)	0.92 (C)	0.92 (C)	0.86 (A)	0.87 (B)
Y6: {*r*}	0.66 (D)	0.65 (C)	0.65 (C)	0.64 (B)	0.55 (A)
Y7: {*c*}	0.92 (D)	0.84 (C)	0.84 (C)	0.69 (B)	0.68 (A)
Mean	0.92 (D)	0.90 (C)	0.90 (C)	0.85 (B)	0.82 (A)
Mean	0.90			0.84	

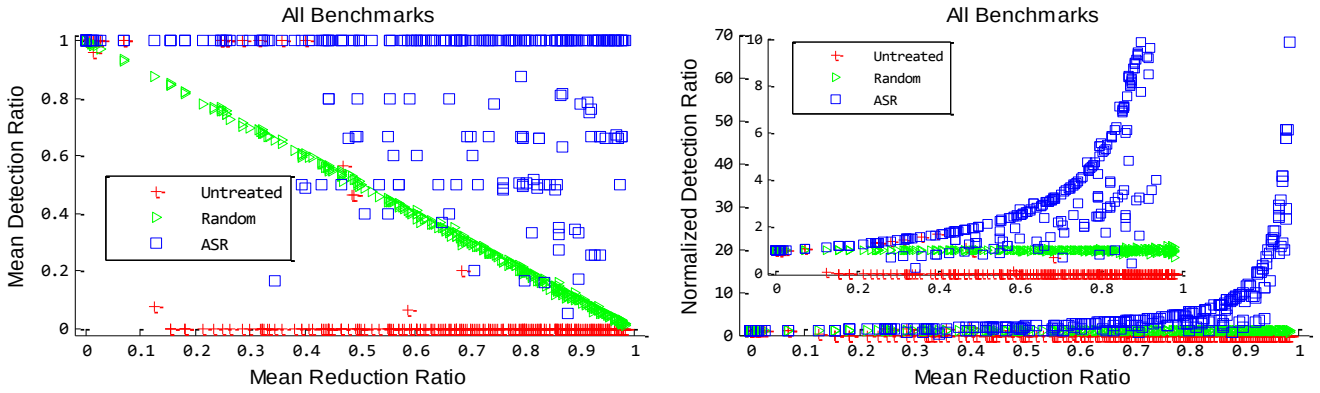

Fig. 5. The mean detection ratio and the mean normalized detection ratio (NDR) of each technique on each benchmark.

the same X level with different *y* values) which do not show any significant differences. Moreover, if a technique has a higher mean detection ratio, we assign a higher letter to it. From TABLE 4, we observe that Y1 and Y2 are the most effective among all subspaces, and they are not significantly different from one another. Y6 is least effective followed by Y7. Y3 is more effective than Y4 except at X4 and X5, and Y3 is more effective than Y5 except at X5. Y4 outperforms Y5 at X4 and X5, but is as effective as Y5 at X1−X3.

We find that one-dimensional subspaces (Y5−Y7) are less effective than two-dimensional subspaces (Y2−Y4) and original space (Y1) in terms of mean detection ratio. Interestingly, such result does not indicate that a higher reduction ratio will lead to a lower detection ratio. For instance, Y1 is as effective as Y2, and yet Y2 is more efficient than Y1. Similarly, the differences in effectiveness among Y3–Y5 are mixed, and yet Y5 is most efficient among Y3–Y5. Encouragingly, Y5 achieves relatively high efficiency with mean detection ratio of 90% and median detection ratio of 100%.

TABLE 5 follows a similar arrangement as TABLE 3. TABLE 5 shows the Fisher's LSD test results along the X dimension in terms of detection ratio. From the results, we observe that coarse-grained abstraction levels (X4–X5) are less effective than fine-grained abstraction levels (X1–X3). Nonetheless, similar to the Y dimension, a higher reduction ratio does not necessarily lead to a lower detection ratio. For instance, X2 is as effective as X3 but X3 is more efficient than X2.

To answer RQ2, we find that the use of one-dimensional subspaces is less effective than the use of other subspaces, and the use of coarse-grained abstraction levels is less effective than the use of fine-grained abstraction levels. However, such results do not indicate that a higher reduction ratio will lead to a lower detection ratio. We find that some ASR techniques (e.g., techniques at Y5) can still achieve a mean detection ratio of 90% and a median detection ratio of 100% with relatively high reduction ratios.

3) Answering RQ3

In this section, we compare each ASR technique to the two controlled techniques: random reduction and untreated reduction. The plot on the left in Fig. 5 shows the mean reduction ratio against the corresponding mean detection ratio of each technique on each benchmark. From the plot, as expected, the detection ratio of the random reduction linearly decreases when the reduction ratio increases. The detection ratio of the untreated reduction is most often close to zero. That is, when the reduction ratio increases, both controlled techniques become *highly ineffective*. With regard to ASR techniques, the detection ratio of ASR is most often located significantly higher than random reduction. In some cases, the mean detection ratio of ASR is (close to) 100% with the reduction ratio higher than 90%.

The mean normalized detection ratio (NDR) of ASR, random reduction, and untreated reduction across all benchmarks are shown in Fig. 5 as well. From the right plot in Fig. 5, we observe that the NDR value of random reduction is always close to 1. The cost-effectiveness of such a technique never changes no matter how many suspicious instances have been reduced. For untreated reduction, the NDR value is mostly close to zero. Such a technique is undesirable. For our ASR techniques, the plot shows the overall cost-effectiveness of ASR techniques increases exponentially when the reduction ratio increases. We note that the regression line for ASR is: $y = 1.076 / (1-x) - 0.917$, with $R^2 = 0.9596$. That is, based on the formula of NDR (i.e., $(d_R / d) / (|R| / |\Omega|)$ = detection ratio / (1 − reduction ratio)), such result indicates that the reduced sets produced by the ASR techniques do retain their effectiveness when the reduction ratios increase. In other words, in the experiment, the studied ASR techniques can make the detection of atomicity violations both efficient and effective.

TABLE 6 further shows the mean NDR of each ASR technique. The table shows that ASR (with mean NDR of 8.50) is more cost-effective than random reduction (with mean NDR of 1.00) by 8.5 folds and untreated reduction (with mean NDR of 0.14) by 60.7 folds. The differences are significant.

We recall that CTrigger [14] is a state-of-the-art dynamic detector that uses abstraction level I and subspace $\{c\}$ [14][28]. Therefore, we further compare each ASR technique to the technique ASR(3, 7), which is modeled after CTrigger. The LSD test results are shown in TABLE 6. In the table, if the test shows that a technique is more cost-effective than or

TABLE 6. THE MEAN NDR+ OF EACH ASR TECHNIQUE AND THE FISHER'S LSD TEST FOR COMPARING EACH TECHNIQUE WITH ASR(3, 7)*

	Y1 {p, r, c}	Y2 {p, r}	Y3 {r, c}	Y4 {p, c}	Y5 {p}	Y6 {r}	Y7 {c}	Mean
X1: *O*	1.00	1.01	1.01	1.35	1.40	2.02	1.38	1.31
X2: *K*	5.67	5.86	6.77	7.16	7.51	8.02	9.45	7.21
X3: *I*	7.43	8.01	8.96	9.28	10.40	11.48	12.18 *	9.68
X4: *S*	10.10	11.63	12.18 (=)	11.66	14.16 (>)	15.83 (>)	15.09 (>)	12.95
X5: *B*	8.70	9.84	10.23	10.66	12.99 (>)	12.87 (>)	14.22 (>)	11.36
Mean	6.58	7.27	7.83	8.02	9.29	10.04	10.46	8.50

Random Reduction: 1.00 **Untreated Reduction**: 0.14

+ NDR = (d_R / d) / (|R| / |Ω|) = *detection ratio* / (1− *reduction ratio*)
* ASR(3, 7) is the technique modeled after a state-of-the-art detector CTrigger

as cost-effective as the technique ASR(3, 7), then we mark the corresponding cell with > or =, respectively. Techniques less cost-effective than ASR(3, 7) are not marked.

From TABLE 6, we find that six techniques ASR(4, 5), ASR(4, 6), ASR(4, 7), ASR(5, 5), ASR(5, 6), and ASR(5, 7) are more cost-effective than ASR(3, 7); and one technique ASR(4, 3) is as cost-effective as ASR(3, 7). The results indicate that seven techniques are at least as cost-effective as ASR(3, 7). Four of these seven techniques adopt the abstraction level X4. We also observe from TABLE 6 that comparing the two rows X4 and X5, X4 is always more cost-effective than X5 in each column. Such result is different from the finding of [28] that the abstraction level X5 is the most effective one among all abstraction levels.

Because the NDR values of ASR techniques increase exponentially as the reduction ratios increase, the mean values shown in TABLE 6 may be relatively higher. Therefore, we further show the median values in TABLE 7 for reference. The median NDR values at Y6 are observably smaller than the mean NDR values at Y6. Such result indicates the variance for Y6 is large, which corresponds to our finding in RQ2 that the detection ratios of Y6 have the largest variance among all subspaces.

To answer RQ3, we find that our ASR techniques are more cost-effective than random reduction by 8.5 folds and untreated reduction by 60.7 folds in terms of mean normalized detection ratio. We also find that ASR techniques can significantly improve the efficiency of atomicity violation bugs detection without compromising the effectiveness of detection. Compared to the technique ASR(3, 7), which is modeled after a state-of-the-art detector, we find that six techniques can be significantly more cost-effective than ASR(3, 7). These techniques all concentrate on the intersection of coarse-grained abstraction levels (X4 and X5) and one-dimensional subspaces (Y5-Y7), which corresponds to our answer to RQ1.

G. Threats to Validity

Our controlled experiment suffers from a number of threats to validity. One threat to validity is the benchmark suite we used. We only used the benchmarks from some previous work [14][30][32], which contain bugs resulting in program failures or bugs exposing vulnerability issues. These benchmarks are not necessarily representative of other programs. The interpretation of the results beyond these benchmarks should be careful. The test cases of these benchmarks were also a threat to validity. To reduce this threat, we had further verified the used test cases of real world server programs against original bug reports. We have not manually localized the exposed bugs to assess the quality of the bug instances selected by ASR.

TABLE 7. THE MEDIAN NDR OF EACH ASR TECHNIQUE

	Y1: {p, r, c}	Y2: {p, r}	Y3: {r, c}	Y4: {p, c}	Y5: {p}	Y6: {r}	Y7: {c}
X1: *O*	1.00	1.01	1.00	1.33	1.35	1.57	1.35
X2: *K*	2.35	2.53	2.78	3.02	3.26	2.27	3.21
X3: *I*	4.56	5.05	5.17	4.82	5.70	3.39	4.90
X4: *S*	4.82	5.87	4.46	6.07	8.31	3.39	5.63
X5: *B*	5.27	6.28	4.90	6.18	7.00	2.40	5.68

V. RELATED WORK

We review closely related work in this section.

In parallel to the present work, Wu et al. [28] proposed the notion of Abstraction Subspace Partitioning and a prioritization algorithm to prioritize the set of suspicious instances generated by the predictive phase of a two-phase detector for its subsequent confirmation phase. Similar to the relationship between test case prioritization [19] and test case reduction [20], the present work and Wu et al. [28] study two closely related but independent aspects in regression testing.

To dynamically detect atomicity violations, Atomizer [6] reports atomicity violations on each execution trace without any confirmation phase. AtomFuzzer [17] complements Atomizer by adding a confirmation phase in the same execution trace. Other dynamic detectors [11][14][23] further employ a two-phase strategy to predict suspicious instances and then examine these instances through other traces. Such a two-phase strategy has also been used to detect other types of concurrency bugs, including data races [21] and deadlocks [2][4]. However, the number of predicted instances in the predictive phase remains large. Our present work interestedly shows that the reduction of suspicious instances has the potential of not significantly compromising the effectiveness of detection of atomicity violations. Our results significantly complement their results to make the detection process more cost-effective.

There are also dynamic techniques [5][7][16][25] not using such a two-phase strategy in detecting concurrency bugs. These scheduling techniques [5][16][25] aim to explore different thread schedules for detection of concurrency bugs. ConTest [5] randomly inserts time delay to perturb thread interleaving schedules. Chess [16] systematically enumerates thread interleaving subsequences by setting a preemption bound. Our present work studies on top of the native scheduler. The use of other schedulers in the predictive phase to generate suspicious instances may lead to different reduction sets of suspicious instances by our algorithm. However, we tend to believe that the results presented in this paper are likely to be valid under such schedules for two reasons: Wu et al. [28] has shown that the result on prioritization is still valid under different thread schedulers of the predictive phase. Moreover, Thomson et al. [24] reported an empirical study on systematic concurrency testing techniques. Their results show that techniques using

native (randomized) schedulers can perform as good as thread-bounding techniques.

Lu et al. [13] reported a survey on a set of concurrency bugs in a suite of multithreaded programs. Their results summarize many insightful heuristics about the types of concurrency bugs in programs and how these bugs can be triggered. It is interesting to integrate their insights into systematic prioritization and reduction.

Similar to existing studies on test suite reduction [10][20], we use the reduction ratio to measure the cost saving of techniques in our experiment. Most of the test suite reduction techniques [10][20] use code-based coverage as the test adequacy criterion. The coverage domains of most coverage criteria proposed in concurrency bug detection [9] are unable to be precisely calculated, however. Almost all existing test suite reduction studies focus on semantic bugs rather than concurrency bugs. Our work also contributes to a novel concurrency-bug-oriented approach to reduce suspicious instances in the detection of concurrency bugs.

VI. CONCLUSION

This paper has reported the first controlled experiment that systemically examines the efficiency, effectiveness, and cost-effectiveness of reduction on suspicious instance set in the two-phase detection of atomicity violations. The empirical results have shown many interesting findings. For instance, our ASR techniques achieve mean reduction ratios of 79% and 72%, at coarse-grained abstraction levels and one-dimensional subspaces, respectively. Also, a higher reduction ratio does not necessarily lead to a lower detection ratio. These results indicate that the reduced sets produced by our ASR techniques have the potential of significantly improving the efficiency of atomicity violation detection without significantly compromising the effectiveness. Moreover, ASR techniques are significantly more cost-effective than random reduction and untreated reduction by 8.5 folds and 60.7 folds, respectively, in terms of mean normalized detection ratio. Such a difference is significant, showing the value of our proposed approach. We have also found that six ASR techniques can be significantly more cost-effective than the technique modeled after a state-of-the-art detector, and all of them concentrate on the intersection of coarse-grained abstraction levels (X4 and X5) and one-dimensional subspaces (Y5−Y7), which are interesting and yet the underlying reasons to be uncovered.

REFERENCES

[1] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ICSE'11*, 2011.

[2] Y. Cai and W. K. Chan. MagicFuzzer: scalable deadlock detection for large-scale applications. In *ICSE '12*, 606–616, 2012.

[3] Y. Cai, K. Zhai, S. Wu, and W.K. Chan. TeamWork: Synchronizing Threads Globally to Detect Real Deadlocks for Multithreaded Programs. In *PPoPP '13*, 311–312, 2013.

[4] Y. Cai, S. Wu, and W.K. Chan. ConLock: a constraint-based approach to dynamic checking on deadlocks in multithreaded programs. In *ICSE '14*, 491–502, 2014.

[5] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java Program Test Generation. In *IBM Systems Journal*, 41(1), pages 111–125, 2002.

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

[7] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In *PLDI '08*, 293–303, 2008.

[8] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *PLDI '09*, 121–133, 2009.

[9] S. Hong, M. Staats, J. Ahn, M. Kim, and G. Rothermel. The impact of concurrent coverage metrics on testing effectiveness. In *ICST' 13*, 232–241, 2013.

[10] D. Jeffrey and R. Gupta. Improving fault detection capability by selectively retaining test cases during test suite reduction. In *TSE*, 33(2), 108-123, 2007.

[11] Z. Lai, S. C. Cheung, and W. K. Chan. Detecting Atomic-Set Serializability Violations in Multithreaded Programs through Active Randomized Testing. In *ICSE '10*, 235–244, 2010.

[12] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *ASPLOS '06*, pages 37–48, 2006.

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

[14] S. Lu, S. Park, and Y. Zhou. Finding atomicity-violation bugs through unserializable interleaving testing. In *TSE*, 38(4), 844–860, 2012.

[15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI '05*, 191–200, 2005.

[16] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *OSDI '08*, 267–280, 2008.

[17] C.-S. Park and K. Sen. Randomized Active Atomicity Violation Detection in Concurrent Programs. In *FSE '08*, 135–145, 2008.

[18] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *ASPLOS '09*, 25–36, 2009.

[19] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *TSE*, 27(10), 929–948, 2001.

[20] G. Rothermel, M. J. Harrold, J. Von Ronne, and C. Hong. Empirical studies of test-suite reduction. In *Software Testing, Verification and Reliability*, 12(4), 219-249, 2002.

[21] K. Sen. Race Directed Random Testing of Concurrent Programs. In *PLDI '08*, 11–21, 2008.

[22] SLOCCount 2.26. http://www.dwheeler.com/sloccount

[23] F. Sorrentino, A. Farzan, and P. Madhusudan. PENELOPE: Weaving Threads to Expose Atomicity Violations. In *FSE '10*, 37–46, 2010.

[24] P. Thomson, A. F. Donaldson, and A. Betts. Concurrency testing using schedule bounding: an empirical study. In *PPoPP '14*, pages 15–28, 2014.

[25] C. Wang, M. Said, and A. Gupta. Coverage guided systematic concurrency testing. In *ICSE '11*, 221-230, 2011.

[26] L. J. Williams and H. Abdi, Fisher's least significance difference (LSD) test. In *Encyclopedia of Research Design*, 491–494, 2010.

[27] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In. *ISCA '95*, 24–36, 1995.

[28] S. Wu, C. Yang, C. Jia, and W.K. Chan. ASP: Abstraction Subspace Partitioning for Detection of Atomicity Violations with an Empirical Study. *IEEE Transactions on Parallel and Distributed Systems* (*TPDS*), 2015. *http://dx.doi.org/10.1109/TPDS.2015.2412544*

[29] J. Yu and S. Narayanasamy. A Case for an Interleaving Constrained Shared-Memory Multi-Processor. In *ISCA'09*, 325–336, 2009.

[30] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: A Coverage-Driven Testing Tool for Multithreaded Programs. In *OOPSLA '12*, 485–502, 2012.

[31] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *ICSE '13*, 192–201, 2013.

[32] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps. ConSeq: Detecting Concurrency Bugs through Sequential Errors. In *ASPLOS '11*, 251–264, 2011.