SDA-CLOUD: A Multi-VM Architecture for Adaptive Dynamic Data Race Detection

Changjiang Jia, Chunbai Yang, W.K. Chan, and Yuen Tak Yu

Abstract—A concrete service consists of a number of program components, each of which is integrated to the service at either design time or runtime. In testing a concrete service, testers should validate the correctness of each of its components under diverse service consumption scenarios. Analyzing the program executions of these components under different configurations allows developers to compare and pinpoint issues therein. There is surprisingly little work in bridging this gap. In this paper, to the best of our knowledge, we propose the first work in designing dynamic analysis-as-a-service using a multi-virtual machine (multi-VM) approach to dynamic data race detection. Almost all existing work on dynamic data race detection focuses on improving detection precision, efficiency, or coverage of thread interleaving scenarios on the same but single compiled concurrent program component. Our model continually selects VM instances, each hosting a different compiled version of the same program component and running a state-of-the-art detector to detect data races. As such, our model innovatively takes existing race detectors as building blocks and operates at a higher level of abstraction. We have evaluated our proposal through an experiment. The experiment reveals that the multi-VM approach is feasible in monitoring multiple compiled versions and can detect different races both in amount and in detection probability. Under a limited execution budget constraint, the multi-VM approach is also significantly more effective in detecting races than approaches that use single compiled versions only. Some races hidden deeply in one compiled version have been found to be significantly more detectable in some other compiled versions of the same service component.

Index Terms—Services engineering, dynamic analysis, services testing, cloud-based usage model, data race detection, selection strategy, virtual machine, PaaS, hidden data race

1 INTRODUCTION

A concrete and atomic service hosting in a cloud is a runtime system that consists of a set of interacting processes. Each such process is a dynamic instance of its corresponding program code, which is assembled to constitute the static counterpart of the service. At the service interface level, a concrete service accepts a service request (e.g., a test input from the testing viewpoint) and returns a corresponding response (e.g., a test output). During such a process, a request session is created, where the underlying processes of the service consume the input data and compute a result. In the real world, an execution of such a program sometimes exhibits undesirable dynamic program behavior, which, if occurred, may affect the dependability (e.g., correctness) or performance of the service in general and the corresponding process in particular. To support a service, such a process is required to handle diverse requests. At the same time, with the advent of multicore processors, more and more of these processes are multithreaded to improve the response time of the offering services. For ease of reference in the paper, we refer to such a program as a service component.

Specifically, a typical execution trace of a service component often involves the execution of multiple threads. Each thread may create a new thread or join to an existing one. A thread execution is a sequence of method invocations, each executing a sequence of program statements. Some of these statements are for thread coordination (e.g., acquisition and release of lock objects), and some others are for sharing resources among threads (e.g., accesses to shared memory locations). The exact execution orders of the statements among these threads in the same trace are determined by factors like the underlying PaaS (Platform as a Service) and the synchronization orders imposed by the developers in the service component.

This dimension of thread interleaving nonetheless incurs the issue of concurrency bugs [16] (e.g., deadlocks among a set of threads [5] or data races on the same memory locations between a pair of threads [11]). Such a concurrency bug may occur only in some but not all thread interleaving even for the same input. Thus, in testing such a service, testers often apply a dynamic analyzer (e.g., a race detector [4], [11]) to execute the service component over the same test case multiple times (e.g., 100 times) so as to cover an adequate subset of the thread interleaving space (with respect to the input) to capture such concurrency bugs cost-effectively. To do so, testers compile the service component followed by executing the dynamic analyzer to monitor the traces of the compiled code over that test case. Typical dynamic analyzer frameworks can automatically run a compiled service component multiple times and report the final results.

The same service component offered as different compiled code versions (see Section 2.3) may be available to different services for smooth integration. Applying the above traditional usage model, developers would need to analyze the service component in each compiled code version multiple times.
This paper extends our preliminary work [20] and presents a novel and refined model, which is referred to as SDA-Cloud, standing for Selective Dynamic Analyzer on the Cloud. Specifically, for each compiled service component (under one compiler configuration setting to be required to test), testers first configure a virtual machine (VM) image running on a cloud platform to host the compiled component and its test cases subject to dynamic analysis. Then SDA-Cloud uses a two-phase approach to detect data races among these compiled components. In Phase 1, SDA-Cloud initializes an instance for each VM image. It associates the process of each compiled component with its dynamic race detector using dynamic binary instrumentation. In other words, SDA-Cloud places the dynamic race detector and the compiled service component in the same memory space to keep the overheads of the involved profiling and analysis in data transfer practical. In Phase 2, unlike the traditional usage model stated above, to analyze the service component, SDA-Cloud invokes an analysis-aware selection strategy to choose among the initialized VM instances iteratively. The selected VM instance (and thus a configuration setting) then invokes one round of race detection and passes back the detection result to SDA-Cloud. For each configuration setting, SDA-Cloud maintains the statistical data that represent the estimated capacity of using the VM instance in detecting data races, and the analysis-aware selection strategy may make use of these statistics of race detection in making its selection decision in subsequent rounds of VM selection.

SDA-Cloud offers data race detection as a service. Users need to provide a VM image installed with the testing subjects with test suites and configure SDA-Cloud accordingly. SDA-Cloud is then able to adaptively distribute the testing workload to perform efficient and cost-effective data race detection.

To evaluate SDA-Cloud, we have built a newer version of our prototype system, conducted a completely new experiment to detect data races in real-world service components with real data races using different analysis-aware strategies, and compared their effectiveness with the usage of race detector in the traditional usage model (i.e., detecting data races on one compiled service component for the same total and predefined number of times of execution). The empirical findings show that on benchmarks having more than 2 races, SDA-Cloud can detect 30 to 100 percent more races using the same overall and limited budget (in terms of 100 test runs) than using the traditional usage model. Together with the qualitative evaluation, SDA-Cloud is found to be significantly more cost-effective compared with the traditional usage model.

The main contribution of this work is two-fold. First, our work is the first one that proposes a novel multi-VM design to a cloud-based service for adaptive dynamic analysis in general and race detection in particular. Second, it is also the first work to report the empirical results on race detection in the context of service (pay-per-use) usage model.

The rest of the paper is organized as follows: Section 2 reviews the preliminaries related to dynamic analysis. Section 3 presents our SDA-Cloud model followed by an evaluation to be reported in Section 4. Section 5 reviews related work. Section 6 concludes the paper.

2 PRELIMINARIES

Our model works at the VM level, and yet its underlying core is a dynamic analyzer in general and data race detector in particular in each VM image. In this section, we revisit the background of dynamic data race detection to enable our work to be accessible to a broad range of readership.

2.1 Dynamic Data Race Detection

Data race refers to two concurrent accesses to the same shared resources and at least one of them is an update (called write) but the order of the two accesses is not protected by some synchronization mechanism. Data races are widely present in multithreaded programs, and they are fundamental (and low-level) symptoms of concurrency issues. In this section, we review Djit+ [33], a classic happened-before [22] relation based race detector.

A trace \( \sigma \) is a global sequence of events produced by multiple threads when running such a program over a test case. An event \( e_i \) represents an operation that the race detector monitors. These events are for thread management (fork/join), synchronization (lock acquisition/release), and memory access (read/write).

The happened-before relation [22], denoted by \( e_1 \rightarrow e_2 \), is a partial order relation defined between a pair of events \( e_1 \) and \( e_2 \) over the trace. Specifically, the relation is defined by three rules. First, if \( e_1 \) precedes \( e_2 \) in the trace and they are performed by the same thread, we have the relation \( e_1 \rightarrow e_2 \). Second, if \( e_1 \) is a lock release event and \( e_2 \) is a subsequent acquire event of the same lock and these two events are performed by different threads, we have the relation \( e_1 \rightarrow e_2 \). Third, the happened-before relation is transitive over all the events of the trace. That is, if we have both \( e_1 \rightarrow e_2 \) and \( e_2 \rightarrow e_3 \), we have the relation \( e_1 \rightarrow e_3 \).

Two events \( e_1 \) and \( e_2 \) are said to be in data race if and only if neither \( e_1 \rightarrow e_2 \) nor \( e_2 \rightarrow e_1 \), both of them access the same shared memory location, and at least one of them is a write event.

A popular type of dynamic data race detectors [2], [11], [33] uses happened-before relation to precisely detect data races on execution traces without reporting false warnings. In this paper, we denote this type of data race detector as precise data race detectors for simplicity.

Fig. 1 illustrates Djit+ [33] on an exemplified trace containing six events (i.e., \( e_1, e_2, e_3, e_4, e_5, \) and \( e_6 \)) performed by two threads (i.e., \( t_1, \) and \( t_2 \)) acting on one lock (\( m \) and one...
memory location \(x\). We will use this example to illustrate race detection as follows.

A **timestamp** is a nonnegative integer. A vector clock \(VC\) is a vector of timestamps, in which the entry \(VC[t]\) keeps a timestamp for thread \(t\). We also denote the current timestamp of the thread \(t\) as \(tstamp(t)\).

Initially, Djit+ creates a vector clock \(VC(t_1)\) for \(t_1\), a vector clock \(VC(t_2)\) for \(t_2\), a vector clock \(VC(m)\) for the lock \(m\), a vector clock \(VC_W(x)\) for read accesses on the memory location \(x\), and another vector clock \(VC_W(x)\) for write accesses to \(x\). Djit+ updates these \(VC\)s according to the sequence of events in the trace. The size of a vector clock is the same as the number of threads in the trace.

When a thread \(t\) acquires the lock \(m\), Djit+ updates \(VC(t)\) to \(\max(VC(t)[i], VC(m)[i])\) for each entry \(i\). That is, Djit+ sets each entry in the vector clock \(VC(t)\) as the maximum of the two timestamps \(VC(t)[i]\) and \(VC(m)[i]\) of the two vector clocks \(VC(t)\) and \(VC(m)\) involved in the lock acquisition event. For instance, in the example shown in Fig. 1, when \(e_4\) occurs, \(VC(t_2) = \max(VC(t_2), VC(m)) = \max(0, 1, (1, 0)) = (1, 1)\). A thread \(t\) releases the lock \(m\), Djit+ assigns \(VC(t)\) to \(VC(m)\), and then increment \(tstamp(t)\) by 1. For instance, when \(e_5\) occurs, \(VC(t_1)\) is updated from \((1, 0)\) to \((2, 0)\).

When a thread \(t\) reads from the memory location \(x\), Djit+ sets \(VC_W(x)[t]\) to \(tstamp(t)\) and checks whether there is a write-read race between \(e_2\) and \(e_5\). Specifically, it evaluates the condition \(VC_W(x)[i] \leq VC(t)[i]\) for all \(i\). If the condition is not satisfied, Djit+ reports a data race.

When a thread \(t\) writes to the memory location \(x\), Djit+ sets \(VC_W(x)[t]\) to \(tstamp(t)\). Then, it checks whether there is a write-write race between \(e_2\) and \(e_5\) and a read-write race between \(e_2\) and \(e_5\). Specifically, it evaluates the condition \(VC_W(x)[i] \leq VC(t)[i]\) for all \(i\) and \(VC_W(x)[i] \leq VC(t)[i]\) for all \(i\), respectively. In either case, if the condition is not satisfied, Djit+ reports a data race (e.g., between \(e_2\) and \(e_5\) in Fig. 1).

FastTrack [11] and Pacer [2] advance Djit+ as follows. They use the notion of epoch instead of a vector to represent the vector clock of each write access to each shared memory location, and dynamically swap between the epoch and vector representations to track the read accesses to the same shared memory location. LOFT [4] is built on top of FastTrack, which further identifies and eliminates a class of vector clock updates and assignments irrelevant in tracking inter-thread happened-before relations for race detection. We denote the precise data race detector used in SDA-Cloud as \(D\), which could be, for example, Djit+, FastTrack, Pacer, or LOFT.

Races may be injected by developers intentionally for performance gains or by mistake. Some data races may be introduced by compilers while performing optimizations or code transformation, resulting in races that may only appear under certain compiler optimization options.

### 2.2 Dynamic Binary Instrumentation

Dynamic binary instrumentation frameworks such as Pin [25] are frameworks that attach instrumentation code dynamically to the runtime of a compiled service component. Tool developers can use the provided function calls to insert their code to analyze the results by their own tools. As such, the entire process of the service component (in the user space) can be monitored and analyzed. When a race on two memory access events in a trace is detected, the race detector \(D\) reports the pair of program instructions in the code that generates these two events, which is called a **racy pair** [4]. In this paper, the set of racy pairs in a given trace is referred to as a **race set**.

### 2.3 Compiler Optimization Configuration Setting

Modern compilers are shipped with control flags to facilitate developers to choose how the source code of their service component is to be translated into lower level representations. For instance, the GNU gcc compiler [12] may be enabled with different optimization options: O0 (default), O1 (level 1 bundle), O2 (level 2 bundle), O3 (level 3 bundle), Os (bundle for size optimization), Ofast (bundle for speed optimization) [18]. Some subsets of control flags, such as O0 and g (for debugging), are compatible and may be turned on at the same time. We refer to such a valid combination of flags as a configuration setting. Each bundle option stated above includes a set of flags and also represents a configuration setting. We note that JVMs (for Java programs) also have their own configuration settings.

### 3 SDA-Cloud

In this section, we present the SDA-Cloud model.

#### 3.1 Motivations and Observations

We have two observations to motivate SDA-Cloud.

First, in a typical service usage model, a tester should only pay for what the tester uses. Ideally, the tester should be able to stop or suspend the dynamic analyzer service on a particular compiled code version and pay to use the same analyzer on another compiled code version flexibly. It is different from the traditional usage model in which a tester has to commit the cost of executing the dynamic analyzer for a predefined number of times on each compiled code version. In the extreme case, to analyze the component offered in every compiled code version, a naive approach is to analyze all these versions independently, each for some predefined number of times. This would incur the highest cost. Although the predefined number of times can be viewed as a parameter of what a tester wants to pay for testing a particular compiled code version, the basic question is whether the tester can achieve more or less the same result by paying a less amount; and if it is feasible, to what extent the job can be done cost-effectively. To the best of our knowledge, we present the first work to answer this question.

Second, a dynamic analysis detector (e.g., [3], [16], [24], [35]) typically has to monitor a huge amount of interesting events along an execution trace of a service component so as to precisely detect concurrency bugs therein, and yet the loosely-coupled message passing approach promoted in service-oriented computing (e.g., via JSON or XML messages) could be too heavy. For instance, a real-world program used in our experimentation was mysql [30], and its execution generated billions of memory access events and synchronization events for precise dynamic data race detection. A typical precise data race detector [2], [4], [11], [16],
3.2 Overview

Fig. 2 depicts the blueprint of SDA-Cloud.

SDA-Cloud consists of a selection component and a set of static VM images administrated by SDA-Cloud.

A service component \( P \) is modeled as a multithreaded program that accepts a test case and computes an output. Each compiled version of \( P \), together with its test cases, all reside in the same VM image. The runtime of \( P \) is executed in the user space of the system running on that VM. The detector \( D \) shares the same memory space with the service component under profiling, thus providing an efficient within-process data transfer for the required dynamic analysis (through function invocation and pointers to shared memory locations).

The compiler configuration setting \( c \) for each VM image is a unique symbol registered in SDA-Cloud. It models the optimization flags that are turned on when compiling the service component \( P \) into the compiled version of \( P \) in that VM image.

The service usage model of SDA-Cloud involves an initialization phase followed by an analysis phase. In the initialization phase, SDA-Cloud prepares an infrastructure for the analysis to be carried out in the subsequent phase. In essence, the first phase initializes the instances of the above-mentioned VM images (i.e., VM\(_i\) in Fig. 2). Service computing advocates the pay-per-use model [37]. In the analysis phase, to allocate the paid budget for testing the service component \( P \) in different configuration settings, SDA-Cloud invokes an analysis-aware strategy. In the design of SDA-Cloud, this strategy adopts a probabilistic approach to select VM instances that are predicted to be most suitable to detect races from the subsequent analysis on execution traces. SDA-Cloud also maintains the necessary data structures to facilitate this analysis-aware selection strategy and the subsequent (race detection) dynamic analysis. In the sequel, we will present the architectural design of SDA-Cloud in these two phases.

3.3 Phase 1: Initialization

This phase is to initialize an infrastructure for subsequent analysis. SDA-Cloud models an analysis VM image as a quadruple \((F, C, D, H)\). In the quadruple, \( F \) represents a compiled version of the service component \( P \), \( C \) is a unique symbol representing the configuration setting enabled when compiling \( P \) into the version \( F \), \( D \) represents the race detector configured by SDA-Cloud, and \( H \) is the report history produced by \( D \). A report history \( H \) is a sequence of race sets, which is initially empty (i.e., \( H = \emptyset \)).

We refer to an instance of an analysis VM image simply as a VM instance. Specifically, a VM instance of an analysis VM image \((F, C, D, H)\) is a 6-tuple \((F, C, D, H, N, R)\). In the tuple, \( N \) and \( R \) are derived attributes. \( N \) denotes the number of times that the VM instance has been used to analyze the execution traces of \( F \). \( N \) is computed as the length of \( H \). The race sets detected from different execution traces of \( F \) may be different. \( R \) represents the set of unique data races kept in \( H \), and is computed by taking a set union of all the race sets kept in \( H \). Initially, \( N \) is 0 and \( R \) is an empty set.

SDA-Cloud initializes a set of VM instances based on the set of analysis VM images, one for each configuration setting. It models all such instances as the children of the service selection component of SDA-Cloud to streamline the service discovery process. The above initialization is performed when SDA-Cloud receives an analysis preparation request, which includes an overall budget \( B \). This value \( B \) is the total number of analysis execution traces to be monitored in the analysis phase across all VM instances. For instance, in the literature of dynamic analysis for race detection [4], [11], \( B \) ranges from 100 to 1,000 per combination of compiled version and a test case. For ease of presentation, we simply refer to each execution trace under the budget \( B \) as a \( B \) run.

SDA-Cloud also maintains the total number of invocations that have been performed, which is denoted by \( S \) and is initially 0. After initialization, SDA-Cloud transits into its analysis phase.

3.4 Phase 2: Analysis

In this phase, SDA-Cloud periodically monitors the progress of race detection under each configuration setting and applies an analysis-aware strategy \( \theta \) to select the compiled version \( F \) in the corresponding VM instance for race detection purpose. SDA-Cloud is designed with the following family of strategies as instances of \( \theta \).

The first strategy is called the single VM option selection strategy (SO). As we have stated in the introduction of this paper, this is the strategy popularly used in existing race detection experimentation. For each of the \( B \) runs, SO assigns it with the same VM instance. SO models the traditional usage model that only considers the service component \( P \) (observable to developers at the design time) for dynamic analysis.
The second strategy is the random VM selection strategy (RS), aka random testing (RT) [8]. It randomly and independently assigns each of the budget \( B \) with one VM instance initialized in the Phase 1.

Before presenting other strategies, we need to revisit the proportional sampling strategy (PSS) [7], [8], which is a partition testing strategy that has been proved to have a higher probability of detecting at least one failure than RT except under certain conditions when their probabilities are equal. A partition testing strategy first divides the input domain of a service component under test into disjoint subdomains and then selects test cases from each subdomain. Formally, suppose that \( I \) is the input domain of a faulty service component \( P \). Let \( I_j \) for \( j = 1 \ldots n \) be the disjoint subdomains of \( I \) such that their union is \( I \). Let \( |f_j| \) denote the number of inputs in the subdomain \( I_j \). Suppose that \( B \) inputs are selected in total from \( I \) in which \( b_j \) is the number of inputs selected from the subdomains \( I_j \) for \( j = 1 \ldots n \), respectively, and the sum of \( b_j \) for \( j = 1 \ldots n \) is \( B \). Then PSS chooses the values of \( b_j \) such that \( \frac{n}{|I_j|} = \ldots = \frac{b_j}{|I_j|} \) which, as proved in [8], guarantees that the probability of detecting at least one failure will be higher than or equal to that achieved by randomly selecting the same total number of inputs \( B \) from \( I \). In the special case when all subdomains are equal in size, PSS evenly allocates the same number of test cases to each subdomain.

Motivated by the theory of partition testing [7], [8], the third strategy we adopt for data race detection budget allocation in this experiment is the systematic multi-VM selection strategy (SS), which evenly allocates all \( B \) runs among all the VM instances. Applying SS once means to select the VM instances in a round-robin manner [27], [39] to perform race detection until the budget \( B \) is exhausted.

SDA-Cloud also supports an analysis-aware strategy to make use of the derived attributes (e.g., \( R \)) of the VM instances. Suppose that some of these \( B \) runs have been analyzed, thereby SDA-Cloud is maintaining the attribute \( R_i \) for each VM instance \( V_{Mi} \).

The next two strategies are variants of SS and are race-aware. They are referred to as the probability proportional-to-race VM selection strategy (PPR) and the probability inverse proportional-to-race VM selection strategy (IPR). Specifically, for the first selection interval, PPR and IPR are the same as SS, which selects each VM instance with the probability of 1. By doing so, each VM instance is selected once in the first interval. Starting from the second selection interval, before doing the selection, PPR assigns the probability of selecting the VM instance \( V_{Mi} \) as \( PPR_i = \frac{R_i}{\sum R_j} \) and that for IPR is \( IPR_i = 1 - \frac{R_i}{\sum R_j} \).

The design philosophy behind PPR strategy is that if a VM instance has reported more races than others in the past, the instance is considered more capable of detecting data races and thus it is allocated a higher proportion of the budget. The design philosophy behind the IPR strategy is different. If a VM instance has reported fewer races than others, it is deemed as not being adequately tested and thus requires more testing resources.

Applying PPR (IPR, respectively) once means that each test run is selected for race detection with the above adaptive probabilities until the budget \( B \) is exhausted.

In each round, after determining which VM instance to be selected, SDA-Cloud invokes the detector \( D \) in \( VM_i \). The detector \( D \) then executes the compiled version \( F_i \) in that VM instance, monitors the execution traces, and reports a race set \( r \), which is appended to the report history \( H_i \) of \( VM_i \). In case of a non-responsive VM execution, the VM instance should be shut down, and a new instance initialized from the same VM image using the same configuration setting can be used to continue the testing.

4 Evaluation

In this section, we report the evaluation of our model.

4.1 Implementation

We implemented a prototype of SDA-Cloud on a Dell server to evaluate SDA-Cloud. The server was configured with 32 Intel Xeon 2.90 GHz cores and 128 GB physical memory. All the VM images and instances were managed in the Microsoft Hyper-V Server 2012 [29]. Hyper-V server supported the creation of snapshot for running VM instances. As such, a new VM instance was created by the replication of such a snapshot in our prototype.

The VM images used in this experiment were configured to run the 32-bit desktop version of Ubuntu Linux 12.04.1 operating system with 3.16 GHz Duo2 processor and 3.8 GB physical memory. We chose the VM in this configuration (at the time of our experimentation) to simulate the setting we observed in the literature on race detection. This configuration had been shown to provide genuine thread control of the execution of each program benchmark where existing work [4], [5], [18] had demonstrated that this platform was able to repeat the third-party experimental results [11] on dynamic race detection. In the experiment, we used gcc 4.6.3 [12], which was shipped with Ubuntu Linux 12.04.1. We used Pin 2.11 [25] to monitor the events of service components developed with the Pthread multithreading model. The whole prototype system, including all the strategies presented in last section, was implemented in C++.

In our experiments, for each benchmark, we manually prepared the VM image of each configuration setting. The VM image preparation tasks included the preparation of test suites, compilation scripts and execution scripts, the compilation from source code to binary executables and the installation of the Pin framework. We simulated the selection component in SDA-Cloud by developing a program in our server to automatically initialize VM instances from the corresponding VM images. This program also took charge of instructing the VM instances to perform data race detection and report the results.

Compared to our previous prototype presented in [20], we have re-implemented the system (with some upgrade and downgrad of library versions). In this way, the system was able to execute the benchmark x264, which was not possible in the earlier prototype.

The prototype system was able to receive commands from users, create and select VM instances, execute a detector in each VM instance, and collect race detection results. We re-implemented the C++ version of LOFT [4] on the Pin framework [25] installed in the VM images. It extends the
TABLE 1
Descriptive Statistics of the Benchmark

<table>
<thead>
<tr>
<th>Service component (Name and version)</th>
<th>Description</th>
<th>SLOC</th>
<th>Size (in KB) of compiled service component</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>O0</td>
</tr>
<tr>
<td>blackscholes 2.2</td>
<td>Stock option pricing</td>
<td>914</td>
<td>37</td>
</tr>
<tr>
<td>swaptions 2.1</td>
<td>Pricing of stock swap options</td>
<td>1,119</td>
<td>74</td>
</tr>
<tr>
<td>canneal 2.2</td>
<td>Simulated cache-aware annealing</td>
<td>2,825</td>
<td>435</td>
</tr>
<tr>
<td>vips 7.22.1</td>
<td>Image Processing</td>
<td>138,536</td>
<td>113,087</td>
</tr>
<tr>
<td>x264 r1047.1</td>
<td>H.264 video encoding</td>
<td>41,933</td>
<td>2,786</td>
</tr>
<tr>
<td>raytrace 1.2</td>
<td>Real-time raytracing</td>
<td>13,764</td>
<td>33,666</td>
</tr>
<tr>
<td>bodtrack 2.0</td>
<td>Body tracking</td>
<td>16,423</td>
<td>2,511</td>
</tr>
<tr>
<td>dedup 3.0</td>
<td>Compression with data deduplication</td>
<td>3,374</td>
<td>242</td>
</tr>
<tr>
<td>streamcluster 2.2</td>
<td>Online clustering of an input stream</td>
<td>1,769</td>
<td>74</td>
</tr>
<tr>
<td>Real-world</td>
<td></td>
<td></td>
<td>996,920</td>
</tr>
<tr>
<td>mysql 5.0.92</td>
<td>Database</td>
<td></td>
<td>197,698</td>
</tr>
</tbody>
</table>

LOFT tool by adding more profiling functions to record the program execution trace information (e.g., PIN_IMG and PIN_RTN functions to return the image and routine of the memory accesses) and supports more Pthread primitives (e.g., pthread_mutex_lock/unlock) instead of merely basic ones. We verified the correctness of our implementation on some small programs and sent it to the authors of LOFT for review.

All the statistics of each VM instance were kept in our prototype instead of bundling them as attributes of corresponding VM instances. The prototype identified each VM instance by its configuration setting.

4.2 Benchmark and Their Compiled Versions

The benchmark we used consists of a set of widely-used subjects including the PARSEC 3.0 benchmark, the popular database system mysql [30] and the web server httpd [15]. They represent service components that were used as part of application logics, data management and web-based communications. To the best of our knowledge, these subjects had also been widely used in existing concurrency bug detection work (e.g., [3], [4], [5], [24]). The PARSEC 3.0 benchmark consists of 13 subjects in total: blackscholes, bodtrack, canneal, dedup, streamcluster, swaptions, vips, x264, raytrace, freqmine, facesim, ferret, and fluidanimate. The last four subjects (i.e., freqmine, facesim, ferret, and fluidanimate) were excluded from our experiment due to the following reasons: (1) our implementation of the race detector was built on top of the Pthread library but freqmine did not use the Pthread library, and (2) the other three subjects crashed when executed with the race detector under the Pin environment on our platform. Thus, we included the remaining 9 subjects in our experiment and executed them with the simsmall test suite and eight worker threads as what the existing work (e.g., [4], [43]) did. This test suite had also been widely used as a whole to execute each PARSEC subject in previous studies (e.g., [4], [16], [35]). For each subject, this test suite consists of a set of correlated test cases. We were unaware of any document to specify how to divide the test suite into unrelated subsets or individual test cases. Thus, to ensure correct usage of the test suite, we executed it as a whole against each PARSEC subject. By doing so, in our experiment, a trace of such a subject referred to an execution of the whole test suite instead of its subset or an individual test case.

For the two real-world subjects, mysql database and the httpd web server, we modified the test harness used in the existing work [4] to invoke the concurrent behavior of the subject (e.g., by sending requests at a higher rate to increase the concurrent connections to the httpd server).

For each subject, we used SLOCCount [38] to count the physical lines of source code (SLOC). We compiled each subject using each of the six configuration settings stated in Section 2.3 to generate one compiled service component. In total, we generated 66 compiled versions of these subjects. The sizes of these compiled service components, measured by du [9], are shown in Table 1. For ease of reference, we identify a particular VM instance by its configuration setting. For instance, the VM instance hosting the compiled component using the configuration setting -Ofast is simply referred to as Ofast. Other VMs are named similarly.

In total, 10 strategies were used in the experiment. They were SO for each of the six configuration settings (denoted as O0, O1, O2, O3, Ofast, and Os) and the four multi-VM strategies (RS, SS, PPR, and IPR).

The large variation in the size of the compiled components for the same subject suggests that different compiled versions may execute different traces in computing the results for the same test case.

4.3 Experimental Procedure

We performed two experiments to validate SDA-Cloud.

The purpose of the first experiment was to validate to what extent SDA-Cloud worked under the traditional usage model. Moreover, it provided a baseline of the data race detection capacity of each VM instance.

In the first experiment, for each subject, we assigned 1,000 to the budget B and requested SDA-Cloud to set up one VM instance from one VM image. Then, SDA-Cloud was executed B times. After having executed all B runs, we collected the race sets. To gain statistical power, we repeated the B executions of each subject 1,000 times.

We performed the procedure stated in the previous paragraph for each configuration setting. We used 11 subjects. Therefore, overall we executed SDA-Cloud for 11
(subjects) × 6 (VM instances) × 1,000 (budgeted runs) × 1,000 (repeated trials) = 66 million times in total. We noted that using Pin for dynamic instrumentation, race detection incurred significant slowdowns ranging from 10 fold to 100 fold [18].

There was only one VM instance in each case. Thus, all the above-mentioned selection strategies degraded into the same strategy, SO. Therefore, we only applied the SO strategy in the first experiment. We automated the entire procedure to remove any need of human intervention to collect the whole set of data from the experiment.

Following [34], we represented a data race as a pair of statements in the source code of a subject. In a test run, the race detector may report a set of data races. For each race, reporting it once is sufficient for developers to identify the source code locations in the corresponding subject. Therefore, in a test run, we only counted the same data race once irrespective of the number of times it was reported by the race detector.

To measure race detection effectiveness, we collected the sets of races reported on each VM instance and computed the number of distinct data races (DR), which was the cardinality of the set union of the race sets collected in one round of SDA-Cloud execution for its both phases. For each distinct data race, we further computed the average number of test runs that detect the data race, and then divided it by \( B \) to compute the detection probability (DP) for the race.

The purpose of the second experiment is to compare the effectiveness of the traditional single-VM strategies (SO) with that of the multi-VM strategies (RS, SS, PPR, and IPR) in SDA-Cloud.

In the second experiment, we assigned 100 to the budget \( B \). In SDA-Cloud, we set up six VM instances from the six VM images, one for each configuration setting. Then, SDA-Cloud was executed \( B \) times. After having executed all \( B \) runs, we collected the race sets. To gain statistical power, we repeated the above procedure 1,000 times. We then repeated the procedure for each analysis-aware strategy from amongst the 6 single-VM strategies and 4 multi-VM strategies. So, in the second experiment, we executed the test suites in SDA-Cloud for 11 (subjects) × 100 (budgeted runs) × 1,000 (repeated trials) × 10 (strategies) = 11 million times. We also measured both DR and DP.

### 4.4 Results and Data Analysis

#### 4.4.1 SDA-Cloud on Traditional Usage Model

Table 2 shows the mean number of distinct data races detected from each of the six VM instances at the end of the first experiment as well as the aggregated number of distinct data races (the column entitled “Overall total”). For instance, for \( \text{vips} \), each VM instance can detect one distinct data race, but all these VM instances together can (only) detect two distinct data races. Other rows can be interpreted similarly.

Along the rows, Table 2 shows that for \( \text{vips}, \text{raytrace}, \text{bodytrack}, \text{streamcluster}, \text{mysql}, \) and \( \text{httpd} \), no single VM instance can detect all data races even using 1,000 runs. The difference between the Total and the value in each other column is relatively larger for real-world subjects (\( \text{mysql} \) and \( \text{httpd} \)). The difference for data mining kind of subjects (represented by \( \text{streamcluster} \)) also seems large.

The results indicate that either the use of multiple VM instances having different configuration settings were necessary, or some races were very difficult to be detected using a single VM instance. Moreover, as far as these real-world subjects and the data-mining subject can represent, the variations in the amounts of races detected seem severe and do not show a consistent trend.

The result indicates that even though SDA-Cloud can support the (baseline) traditional usage model (which analyzes a service component on a single VM image) initially, it should be extensible to allow analyses with different configuration settings if testers are dissatisfied with the results obtained from the current VM image.

The baseline traditional usage model can be extended to support the use of multiple configuration settings. On closer look, however, we experienced that subjects like \( \text{mysql} \) required the setup of some shared libraries at specific locations, and these shared libraries and file contents were required to be compatible to the compiled version of \( \text{mysql} \) and located at some specific locations in the file system. Thus, when switching to a different configuration setting,
the previous setting for dynamic analysis has to be undone first. Alternatively, developers may resolve to use multiple machines (VM images), each configured for the testing of one compiled version of a subject. However, doing so essentially adopts a simplified version of SDA-Cloud, where SDA-Cloud has the feature of analysis-aware strategies and ability to execute the same dynamic detector multiple times with result collection and summarization. Yet another alternative is to add VM instances incrementally to SDA-Cloud. The architecture of SDA-Cloud attaches each VM instance as a child of the selection component. By registering additional VM images to SDA-Cloud, its Phase 1 can initialize the additional VM instances incrementally. To apply Phase 2 to the new VM instance only (by following the traditional usage model), the old VM instances and images (and thus the corresponding configuration settings) can be removed from the children list of the selection component accordingly. The architecture of SDA-Cloud can readily accommodate the application of any of the above alternatives.

Table 2 also shows that different VM instances detected rather different numbers of distinct data races. Moreover, with only one exception, each VM instance only detected less than half of all distinct data races in the experiment. There was also no general pattern of which particular VM instance is always superior (or inferior) to the others. The results indicate that, by adopting the traditional usage model, developers may not be able to utilize their experience in choosing a cost-effective machine to detect races from the subject under test. SDA-Cloud makes use of the ability to initialize multiple VM instances from multiple VM images and switch to run (or suspend) VM instances for race detection to achieve a cost-effective approach to dynamic analysis-as-a-service. An evaluation of the cost-effectiveness of SDA-Cloud will be presented in Section 4.4.2.

In this experiment, we experienced that SDA-Cloud successfully completed the race analysis of each trace of each subject over each test suite without human intervention. We recalled from the results reported in the preliminary version [20] of this paper that some subjects could not run successfully. We found that the main difference was due to the use of an improved race detector implemented in each VM image, as presented in the penultimate paragraph of Section 4.1. This experience indicates that the architecture of SDA-Cloud can be configured to use a different dynamic analyzer, and its ability to be offered as an analysis-as-a-service depends on the implementation compatibility between the detector and the subject under analysis, which is beyond the consideration in designing SDA-Cloud.

We have presented above that different VM instances detected different numbers of distinct races on some subjects. To understand the extent of reduction of the VM image setup cost in SDA-Cloud, we have further analyzed to what extent different distinct races can be detected by different numbers of VM instances. Table 3 shows the summarized results. Along each row from left to right, the table shows the number of distinct races that can be detected by 1 VM only, 2 VMs only, 3 VMs only, 4 VMs only, 5 VMs only, and all 6 VMs, respectively.

It is interesting to observe the following. First, 41 out of 158 races (or 26 percent of all races) could only be detected by 1 VM, and yet the developers would not know which VM to use for detecting the race! It shows that the design of SDA-Cloud supports the use of different VM instances to detect races in a cost-effective manner, which is essential to offering dynamic analysis as a service.

Second, 40 races (or 25 percent of all races) can be detected by any one of the six VMs. That is, these races may be detected as long as an adequate number of test runs (i.e., the parameter B in SDA-Cloud) can be used. They form a basis to compare races in different configurations settings. Thus, a further question is whether these races can be more or less detectable in each VM. We therefore further analyzed these races to answer this question.

The results are plotted in Fig. 3, where the y-axis shows the maximum difference in detection probabilities among all VM instances for each of these 40 races, and the x-axis shows the ids of these 40 races in descending order of their y-values. The figure shows that in 25 percent, 50 percent, and 75 percent of these races, the maximum difference in detection probabilities is larger than or equal to 0.46, 0.25, and 0.08, respectively. The finding shows that different configuration settings indeed render SDA-Cloud to show significantly different capabilities in detecting the same race.
It is worthy of mentioning that there were races (at positions $x = 1$ to 3, for example) having the differences of more than 0.9 (or 90 percent differences in detection probabilities between VM instances). By alternately using multiple VMs, SDA-Cloud can detect these races effectively. On the other hand, in the traditional usage model, the most effective VM instance with respect to these races may not be given a high enough priority, and thus testing through the use of other VM instances should be conducted.

We also computed the detection probability of each race in each VM instance. We found that except for one particular data race for httpd, SDA-Cloud can achieve the detection probability of at least 1 percent if all the VMs can be used. (For that particular race for httpd, the detection probability is 0.9 percent, i.e., 9 out of 1,000 runs in the experiment.) On the other hand, if only one VM is used, some races cannot be detected by some VM instances at all, which are consistent with the findings shown in Table 2. It is therefore necessary to allow developers to increase the test budget when the additional races detected begin to be diminishing.

As expected, we have also observed that there were races (at positions $x = 1$ to 3, for example) having the differences of more than 0.9 (or 90 percent differences in detection probabilities between VM instances). By alternately using multiple VMs, SDA-Cloud can achieve the detection probability of at least 1 percent if all the VMs can be used. (For that particular race for httpd, the detection probability is 0.9 percent, i.e., 9 out of 1,000 runs in the experiment.) On the other hand, in the traditional usage model, the most effective VM instance with respect to these races may not be given a high enough priority, and thus testing through the use of other VM instances should be conducted.

We also computed the detection probability of each race in each VM instance. We found that except for one particular data race for httpd, SDA-Cloud can achieve the detection probability of at least 1 percent if all the VMs can be used. (For that particular race for httpd, the detection probability is 0.9 percent, i.e., 9 out of 1,000 runs in the experiment.) On the other hand, if only one VM is used, some races cannot be detected by some VM instances at all, which are consistent with the findings shown in Table 2. It is therefore necessary to allow developers to increase the test budget when the additional races detected begin to be diminishing.

As expected, we have also observed that the time needed to instruct VM instances and maintain the result statistics was negligible. It is partially because we implemented the whole experiment in the same server and the communication between VM instances and the centralized component for VM selection only generated a very small amount of data transfers per analysis request. On the other hand, each execution of a compiled service component generated hundreds of millions of events (consistent with the findings reported in [4], [11]), which significantly slowed down each round of analysis [25].

### 4.4.2 SDA-Cloud on Services Usage Model

This section reports our findings in the second experiment. We compared the race detection effectiveness of different strategies implemented in SDA-Cloud. Recall from Section 3 that there are in total 10 strategy instances. Out of the 11 subjects, three (blackscholes, swaptions, and dedup) of them did not exhibit any race from any strategy. Therefore, we only show the results of the remaining eight subjects, as summarized in Fig. 4. For each boxplot, the $x$-axis represents a concrete strategy (denoted as O0, O1, O2, O3, Ofast, Os, RS, SS, PPR, and IPR) and the $y$-axis represents the number of distinct races detected from a subject. The lines in the boxes indicate the lower quartile, median, and upper quartile values. A cross indicates a suspected outlier. Boxes whose notches do not overlap mean that the medians of the two groups differ at the 5 percent significance level. All boxplots were generated by MatLab.

For canneal, only one race was detected. Hence, the boxplots for all concrete strategies are the same. However, for all other subjects, the multi-VM strategies (RS, SS, PPR, and IPR) were all more effective than the six single-VM concrete strategies (O0, O1, O2, O3, Ofast, Os) by a large extent. For instances, the median numbers of races detected by the multi-VM strategies on the subjects vips, x264, raytrace, bodytrack, streamcluster, mysql and httpd, were around 2, 2, 10–11, 20, 12, 45–55 and 50–55, respectively, whereas those of the single-VM strategies on these benchmarks were around 1, 1–2, 3–8, 2–11, 5–6, 3–40, and 30–44, respectively. The corresponding differences are noticeable.

To triangulate the above observation, we conducted hypothesis testing using Analysis of Variance (ANOVA) on the dataset of each subject presented in the plots and confirmed a significant difference among the ten groups at the 5 percent significance level. We further conducted the multiple-mean comparison test using Matlab (with HSD [27], which is the default option for Matlab multiple comparison). Fig. 5 shows the multiple-mean comparison result of the ten strategies for each subject. For all subjects except canneal and x264, each of the four multi-VM strategies (SS, RS, PPR and IPR) were all significantly more effective than each of the single-VM concrete strategies. Among the six single-VM concrete strategies, none of them consistently and statistically outperformed the others for all subjects. This finding shows the effectiveness of SDA-Cloud as an architecture designed to support multiple VMs to be used flexibly for dynamic analysis.

### 4.4.3 Qualitative Analysis on SDA-Cloud Software Architecture

Our prototype can complete the two experiments with no human effort after the preparation of VM images by developers. This indicates that it is feasible to instruct the dynamic analysis detectors in VM instances to profile and analyze executions and then SDA-Cloud aggregates the results. Nonetheless, the two experiments added up to spend over 1.5 months for nonstop computation. Apparently, one may lower the number of test runs needed to speed up the total elapse time of the analysis.
We have examined the dataset, but found that for httpd, there was a race that could only be detected in 3 VM instances and no VM instance could detect it with a probability higher than 1 percent. Thus, there appears to be a lower threshold imposed on the test budget to detect races reliably. It further indicates that a real system of SDA-Cloud should support the usage of multiple VM instances for each configuration setting instead of one VM instance as what we did in the experiments. The current architecture of SDA-Cloud uses the configuration setting to identify which VM image to be initialized and which VM instance to be executed based on the analysis-aware strategy. It can be trivially extended by creating a set of identical VM instances for each configuration setting.

Besides, our experience with the earlier prototype of SDA-Cloud presented in the preliminary version [20] of this paper was that a VM instance once became non-responsive (for reasons out of our control). In that case, the VM instance was terminated and re-initialized. In our model, for each VM instance, the compiled version of the service component $F$ and the detector $D$ were preloaded in the image of the VM instance, the data $H, N,$ and $R$ were kept in the prototype system of SDA-Cloud rather than in the VM instance, and our prototype system identified a VM instance by the configuration setting string. Therefore, our architecture can directly support the re-initialization of VM instances. Unlike the experiment reported in the preliminary version [20] of this paper, our current SDA-Cloud system did not experience any non-response VM instance.

A subject may become unable to proceed further (e.g., due to their poor software design). A typical dynamic detector usually incorporates a mechanism to timeout the process, terminates the current execution of the detector, and restarts it again. In our experiments, we had not encountered this scenario. SDA-Cloud assumes that each VM image has been successfully configured by developers to enable the subject to be run natively, which we believe is a reasonable assumption.

4.5 Threats to Validity
This study had not measured the effects on the detection of other types of concurrency bugs (e.g., deadlocks, atomicity violations). Although our definition and counting scheme of data races were representative, the use of other measures may yield different results from ours.
A trace in the experiment is the result of executing the test suite as a whole. If the test suite can be decomposed into subsets or individual test cases, although the number of distinct races detected in individual VM instances or via individual strategies might remain the same, the detection probability on individual races could change.

Ten strategies had been compared in the experiment. These strategies were either fundamental ones (e.g., random selection or systematic selection) or race-aware. Comparison between multi-VM and single-VM in terms of effectiveness can be further strengthened by extending the experiment with additional strategies.

We had assured our prototype system with small programs with and without races. The underlying detector was built on top of a third-party framework (Pin). Any race activated by Pin in the experiments posed a threat to our study, and yet we had not observed abnormal situations in the experiment nor when we analyzed the dataset. We had also reviewed the literature on race detections (including our previous work) to verify that our experimental procedure was in line with those presented in existing work.

Another threat is that our experiments only used one compiler. The use of other compilers can help to generalize the results. However, at the time of conducting the experiments, gcc was the standard compiler in the Linux system and to the best of our knowledge, most cloud providers offer Linux to solution service providers, which makes our experiments representative. The use of other subjects, other test cases, other configuration settings, and other numbers of test runs as well as other architectures to link the selection component and its children VM instances (see Fig. 1) may have impact on the results. Thus, like any other empirical study, results of the experiments should only be extrapolated with caution.

In our second experiment, we only studied the difference in effectiveness across different race detection strategies under the same budget, which is 100 test runs. The impact of test budget on the effectiveness of race detection strategies (i.e., SO, SS, RS, PPR and IPR) is an interesting question to be further studied. We consider that another large-scale systematic empirical study is necessary and we take it as a future work. Toward this goal, in the online appendix of this paper, we report an additional evaluation which
summarizes the slowdown incurred in race detection in each configuration setting and the amount of races reported by each strategy using 1 to 1,000 runs.

5 RELATED WORK

In this section, we review the literature related to our work.

Testing as a service (TaaS) on and for cloud computing is one of the closely related areas and has been increasingly studied by both the communities of cloud computing and software engineering. We have previously reported a comprehensive survey [19] on this topic. For brevity, we revisit some of the most representative ones in this section. Zhu and Zhang [44] propose to use existing tools as individual services and then test web services on cloud via a service orchestration approach. To handle the common challenge of TaaS, namely, the lack of control over testing executions, they utilize coordination components to support the collaboration of test service components. SDA-Cloud also relies on a selection component to coordinate race detection executions in different VM instances. However, rather than collaborating heterogeneous test services to meet the overall testing target, our work focuses on adaptive coordination of test executions to improve testing effectiveness. Yu et al. [41], [42] propose to organize different tasks by using different test scheduling and dispatching strategies so as to consider the environment factors in testing in the cloud. They demonstrate the scalability of test services and evaluate the efficiency of the scheduling and dispatching strategies. By contrast, we study and evaluate the effectiveness of data race detection within a user-specified testing budget across different selection strategies (defined in Section 3.4) in SDA-Cloud. Candea et al. [6] propose Cloud9, which applies parallel symbolic execution in a cloud setting to generate test cases and expose errors. Thus, Cloud9 can also be considered as a dynamic analysis-as-a-service on the cloud. Cloud9 focuses on enabling parallel symbolic execution to utilize the elastic resources in cloud with full automation. Different from Cloud9, our work focuses on adaptive selection of multiple VMs for data race detection and there is no public literature or document to indicate that Cloud9 supports such adaptive settings. Both Candea et al. [6] and Jenkins et al. [17] interestingly propose models that can be customized according to different usage scenarios. Tsai et al. [36] propose a new approach of service composition and its testing workflow in cloud. Rather than separating user application execution and testing execution as independent services composed together during testing, SDA-Cloud takes a holistic approach, for which the primary reason is due to the huge amount of events generated in monitoring memory access events in program executions. Our work treats each VM instance as a building block and uses these VM instances for dynamic race detection. Our work provides a framework such that the subject under analysis and the dynamic race detector are tightly cohesive in each VM instance. Yan et al. [40] also adopt such a holistic approach to develop a platform to support performance testing in the cloud.

Another aspect closely related to our work is the selection strategy for quality assurance or maintenance activities. Unlike our work which focuses on the selection of VM instances, to the best of our knowledge, all the existing work focuses on the selection of test cases (e.g., [23], [31]) or the generation of test inputs [32]. Hou et al. [14] also apply the notion of quota, which is assigned to web services as their upper limits of requests allowed to be sent by users. However, the quota (budget) in their work is static, whereas in our work a budget allocated to a VM instance may change dynamically. Zhai et al. [45] propose to test composite services with service selection that relies on a dynamic service discovery mechanism. In contrast, our work does not involve such a mechanism. Mei et al. [28] do not consider the notion of testing quota or budget. Rather, they adaptively and continuously test a system until no more changes can be detected. To the best of our knowledge, no existing work in dynamic analysis and testing considers the selection problem among VM instances in the service model when offering such quality assurance process as a service.

There are model-based approaches which model the cloud for the purpose of simulating the real environment to facilitate the testing of cloud applications [46]. Data races in shared memory systems are due to improper accesses of concurrently executing threads on shared memory locations. To the best of our knowledge, all the existing work (e.g., [4], [11], [16], [18] and the techniques they compared to) deploys the runtime of race detection and the program under detection in the same memory space of the same machine. Decoupling the race detection runtime and the subject program in different memory spaces inevitably incurs great communication overhead, which is thus not adopted in SDA-Cloud.

CoRD [21] provides a framework for distributed data race detection that utilizes the distributed and enormous computation resources of cloud platforms. CoRD first performs static data race detection to acquire a set of potential data races. Then, it resorts to crowdsourced executions to confirm whether the potential races are real races. The confirmation process of potential races is distributed to computational nodes, e.g., VM instances or compute nodes in a computer cluster. CoRD aims to share the loads among a set of homogeneous nodes, while SDA-Cloud pursues cost-effectiveness in race detection among heterogeneous nodes.

Cloud platform provides a great amount of resources to software testing in general. According to a recent survey on software testing in cloud [19], testing parallelization is a hot topic studied in the community. Our SDA-Cloud saves the transfer cost by putting the runtime of program under detection and race detection in the same memory space. In addition, SDA-Cloud partitions the testing process against different configuration settings and proposes an analysis-aware strategy to allocate the testing budget to different testing units dynamically. The analysis-aware selection strategy enables SDA-Cloud to not only allocate a larger portion of the budget to those VM instances that are estimated to be able to detect more distinct races, but also provide opportunities to those VM instances that currently detect fewer races.

6 CONCLUSION

To the best of our knowledge, this paper has presented the first work to offer trace-based dynamic analysis as a service using a novel multi-VM architecture approach in the cloud. We have casted our work to dynamic online data race
detection on service components. We called our approach the SDA-Cloud architecture. SDA-Cloud made a practical assumption that the VM image has been installed with the compiled service component under analysis, the test cases and the dynamic detector, and that they are all compatible. It respects the holistic approach required by dynamic binary instrumentation that underpins the detectors. Such a detector and the service component should share the same memory space to avoid incurring unacceptable slowdown required to transmit a huge amount of data access events between the profiling module and the analysis module. The architecture of SDA-Cloud includes a separate selection module, which chooses among VM instances for dynamic analysis. We have presented five VM selection strategies in this paper, four out of them being multi-VM strategies and one being single-VM. The single-VM strategy can be initialized based on each configuration setting.

We have also presented the first work to evaluate the impact of VM selections on dynamic race detection. We have evaluated SDA-Cloud by means of two experiments. The empirical result has shown that the use of multiple VMs to cover all configuration settings were consistently and significantly more effective than merely using the single-VM strategy for most subjects, and were at least as effective as the latter for the remaining subjects.

We would like to note that the model of SDA-Cloud is general and can be applied to handle programs of different languages, e.g., Java programs.

To the best of our knowledge, this paper has presented a new and interesting direction of research, which is to embody an individual dynamic analysis unit within a VM instance and allow sharing of analysis results to leverage the cost-effectiveness achieved by individual units. The uses of evolutionary algorithms, machine learning or Big Data approaches to select VMs should be studied in the future.

ACKNOWLEDGMENTS

This work was supported in part by the GRF and ECS of Research Grants Council of Hong Kong (project numbers 11200015, 11201114, 111313, 125113, and 125312). A preliminary version of this paper was presented at 2015 IEEE International Conference on Web Services (ICWS ’15) [20]. W.K. Chan is the contact author.

REFERENCES


Chungchang Jia received the PhD degree from the Department of Computer Science, City University of Hong Kong and the BEng and MEng degrees from National University of Defense Technology, China. His research interests include concurrency bug detection and the testing of service-based and cloud-based applications.

Chunbai Yang received the BEng degree in software engineering from the School of Software, Tianjin University, China. He is working toward the PhD degree in the Department of Computer Science, City University of Hong Kong. His current research interest includes dynamic program analysis for detection of concurrency bugs in multithreaded programs.

W. K. Chan is an associate professor with City University of Hong Kong. His current main research interests include program analysis and testing for concurrent software and systems. He is on the editorial board of the Journal of Systems and Software, the International Journal of Big Data Intelligence, and the International Journal of Creative Computing. He is a regular PC member of the ICWS and ICSOC series of conferences, and has served as the innovation show case chair of ICWS/SCC 2009–2010 and the ICWS 2015–2016 chair of the area on services composition.

Yuen Tak Yu is an associate professor with City University of Hong Kong. His research interests include software testing, e-commerce and computers in education. His publications have appeared in scholarly journals and leading international conferences, such as the ACM Transactions on Software Engineering and Methodology, the IEEE Transactions on Software Engineering, Information and Software Technology, Information Research, Computers and Education, ICSE, FSE, ISSRE, ICCE and others. He is a past chair of the IEEE Hong Kong Section Computer Society Chapter.