Architecturing Dynamic Data Race Detection as a Cloud-based Service

Changjiang Jia
Department of Computer Science
City University of Hong Kong
Hong Kong
cjjia.cs@gmail.com

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*—A web-based service consists of layers of programs (components) in the technology stack. Analyzing program executions of these components separately allows service vendors to acquire insights into specific program behaviors or problems in these components, thereby pinpointing areas of improvement in their offering services. Many existing approaches for testing as a service take an orchestration approach that splits components under test and the analysis services into a set of distributed modules communicating through message-based approaches. In this paper, we present the *first* work in providing dynamic analysis as a service using a virtual machine (VM)-based approach on dynamic data race detection. Such a detection needs to track a huge number of events performed by each thread of a program execution of a service component, making such an analysis unsuitable to use message passing to transit huge numbers of events individually. In our model, we instruct VMs to perform holistic dynamic race detections on service components and only transfer the detection results to our service selection component. With such result data as the guidance, the service selection component accordingly selects VM instances to fulfill subsequent analysis requests. The experimental results show that our model is feasible.**

Keywords—service engineering, dynamic analysis, cloud-based usage model, data race detection, service selection strategy

I. Introduction

A *web-based service* is a kind of software-intensive runtime system. Its system architecture includes layers of programs in the technology stack [10]. For ease of our presentation, we refer to such a program as a ***service component***.

A web-based service takes a service request from a service consumer and returns a corresponding response. Such a response (in a session) is computed by executing a set of the service-involved components. Within such a response session, each service component may exhibit certain undesirable dynamic program behaviors, which significantly affect the dependability of the service or the performance of the response session in general and the service component in particular.

For instance, in computing a response, a particular service component may either lead the service to crash, making the whole service unable to serve all the other requests for a while, or run unexpectedly slow, lowering the service level (e.g., response time) for this response session perceived by the service consumer of the session.

Analyzing the program executions of a service component enables service providers to acquire insights into interesting dynamic program behaviors (e.g., "hot paths", which are frequently executed when the service is being consumed in a session [1]) and problems (e.g., faults or concurrency bugs [13]) in the service component. These insights thereby pinpoint areas of improvement (e.g., optimization with respect to these hot paths or bug fixing) in these services.

With the advent of multicore processors, many service components are implemented as multithreaded programs. Specifically, each execution run of such a multithreaded service component creates one or more threads. Each thread executes a sequence of program instructions. Some of these instructions are locking operations on some lock objects, which are designed to protect certain shared resources (e.g., memory locations) from concurrent accesses. To access such shared resources, threads should interleave among themselves to acquire these lock objects in turn to harvest the parallel processing potential of the underlying multicore platforms, and yet the order of their operations is not solely determined by the program code listing but also the interleaving among threads.

Hence, for a multithreaded service component, due to the thread interleaving, multiple execution runs with the same input data usually produce different sequences of program instructions. Some interleaving-specific issues (e.g., deadlocks among a set of threads [4] or data races on same memory locations between a pair of threads [8]) may only appear in some but not all execution runs. Thus, in a dynamic analysis session, to collect the facts of interests (e.g., data race occurrences) from the observed executions of a service component, the dynamic analyzer (e.g., a data race detector) needs to collect and analyze ***multiple*** **(**denoted as *n* for each of reference**)** execution runs of the multithreaded service component for the same input data.

This setting is a popular design in current dynamic concurrency bug detector framework. In the usage of such a dynamic analyzer, developers should firstly compile the source code of a service component, followed by executing the compiled code with the dynamic analyzer, which will automatically run the compiled service component for *n* times and report the results.

A service component may be placed into stacks of different services with different configuration settings (see Section II.C) for smooth integration. To check such a usage scenario, the developers would need to analyze the service component against these compiler configuration settings one by one, each for *n* times.

There are obvious limitations when migrating the above traditional usage model to a service usage model, where service consumers only pay for what they use. For instance, a developer in the traditional usage model has to pay the cost of analyzing one compiled component under one configuration setting for *n* times in full before selecting to analyze the same component under a different configuration setting. To analyze the component under all configuration settings, a naive approach is to concurrently analyze all compiled versions of the same service component. This would incur the highest cost.

Moreover, some service components may generate a great number of events required by the dynamic analysis techniques (e.g., [2][13][19][26]) in each execution run. For instance, the subject (mysql [22]) in our experiment generated billions of memory access and locking events for dynamic data race detection. If the design of a concurrency bug detector (e.g., [2][8][13][19][26][32]) follows the popular service composition discipline, the profiling module and the analysis module would be decomposed into separate services, communicating through loosely coupled messages. Such an event passing mechanism results in huge overhead in inter-service communications.

In this paper, to the best of our knowledge, we propose the *first* work to address challenges in offering a dynamic analyzer that exposes concurrency issues (data race precisely) in service components using a multi-VM approach. Our approach is called **SDA-Cloud**, meaning **S**elective **D**ynamic **A**nalyzer on the **Cloud**. Before running SDA-Cloud, we assume that developers have configured a virtual machine (VM) image running on a cloud platform[1] that has hosted the service component with a set of test inputs subject to dynamic analysis[2]. SDA-Cloud uses a two-phase approach to detect data races in service components under multiple compiler configuration settings. In Phase 1, SDA-Cloud initializes one instance from the VM image for each *unique* compiler configuration setting and compiles the service component in that VM instance with the setting. Each VM instance attaches the compiled service component with a dynamic analyzer using dynamic instrumentation support[3]. In other words, SDA-Cloud takes a holistic approach to place the dynamic concurrency bug detector and the subject under analysis 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, upon receiving a service request from a service consumer to analyze the service component, SDA-Cloud uses a new *analysis-aware selection* strategy (which is also a kind of adaptive strategy [21]) to select VM instances to analyze the component under detection. Each configuration setting has a weight that represents the estimated capacities of detecting data races. SDA-Cloud selects VM instances of each configuration setting with the probability proportional to the corresponding weight. The selected VM instance conducts race detection once and then returns results to SDA-Cloud. Based on the historic results, SDA-Cloud periodically adjusts the weight of each configuration setting to guide the follow-up analysis periods. After each analysis period, the service consumer may then decide to terminate the analysis or continue the dynamic analysis by sending a request to SDA-Cloud. To evaluate whether our model is feasible, we built a prototype and used it to detect data races from a set of real-world service components with real-world concurrency bugs. We evaluated our model both qualitatively and quantitatively. The result shows that our approach is feasible.

The main contribution of this work is two-fold. First, it is the first work that proposes a dynamic analyzer as a cloud-based service using a multi-VM approach. Second, it reports an evaluation on the core component of SDA-Cloud, which demonstrates that the architecture of SDA-Cloud is feasible.

The rest of the paper is organized as follows: Section II reviews the preliminaries related to dynamic analysis. Section III presents our SDA-Cloud model followed by an evaluation to be reported in Section IV. Section V reviews related work. Section VI concludes the paper.

II. Preliminaries

Our work spreads over the domain of dynamic program analysis and service selection. In this section, we revisit the background of dynamic data race detection to make our work self-contained.

A. Dynamic Online Data Race Detection

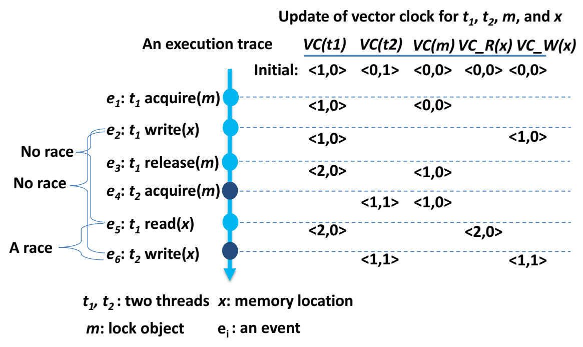
Data races are a fundamental kind of concurrency bugs appearing in multithreaded programs. In this section, we overview the algorithm of Djit+ [25], a classic happened-before [17] relation based race detector, to understand how the races are dynamically detected from multithreaded programs.

Executing the multithreaded program once generates a *trace* σ, which is a global sequence of events from multiple threads. A critical *event* [8] is an operation of interests for race detection, such as lock acquisition/release events, memory read/write access events, and thread creation/join events. The *happened-before relation* [17], denoted by $e_1 \rightarrow e_2$, is a partial order defined between two events e_1 and e_2 over σ by the following three rules: (1) Program order: if e_1 and e_2 are two events performed by the same thread, and e_1 precedes e_2, then we say $e_1 \rightarrow e_2$. (2) Release and acquire: if e_1 is a release event of a lock object and e_2 is a subsequent acquire event of the same lock object performed by a thread different from the one performing e_1, then we say $e_1 \rightarrow e_2$. (3) Transitivity: if $e_1 \rightarrow e_2$ and $e_2 \rightarrow e_3$, then we say $e_1 \rightarrow e_3$. Two events e_1 and e_2 are in

[1] This can be done using Google Cloud Platform or Amazon Visual Server services or a standard cloud server.

[2] This assumption is practical because it is closely modeled after a popular real-world development setting where the developers prepare a server (on a cloud platform or as an in-house server) for the dynamic analysis of the service component before executing a dynamic analyzer.

[3] In our prototype, we use Pin [20] to build our dynamic analyzer as a pintool.


Update of vector clock for t_1, t_2, m, and x
An execution trace
VC(t1) VC(t2) VC(m) VC_R(x) VC_W(x)
Initial: <1,0> <0,1> <0,0> <0,0> <0,0>
e_1: t_1 acquire(m)
<1,0> <0,0>
e_2: t_1 write(x)
<1,0> <1,0>
e_3: t_1 release(m)
<2,0> <1,0>
e_4: t_2 acquire(m)
<1,1> <1,0>
e_5: t_1 read(x)
<2,0> <2,0>
e_6: t_2 write(x)
<1,1> <1,1>
No race
No race
A race
t_1, t_2 : two threads x: memory location
m: lock object e_i : an event
Figure 1. An illustration of Djit+ detecting data races from an execution trace

data race if and only if neither $e_1 \rightarrow e_2$ nor $e_2 \rightarrow e_1$ and at least one of them is a write.

Figure 1 illustrates the algorithm of Djit+ [25] on a simple execution trace, which includes six events (e_1, e_2, e_3, e_4, e_5, e_6) performed by two threads (t_1, t_2) on one lock object (m) and one memory location (x). A *timestamp* is a number. A *vector clock VC* is a vector of timestamps, in which the entry $VC[t]$ keeps a timestamp for the thread t. Initially Djit+ assigns one vector clock for each thread (e.g., $VC(t_1)$ for t_1), one vector clock for each lock object (e.g., $VC(m)$ for m)), and two vector clocks for each memory location (one for read access (e.g., $VC\_R(x)$) and another for write access ($VC\_W(x)$)). Djit+ updates vector clocks when the events occur to maintain the happens-before relation of the execution trace. Specifically, on a thread t acquiring a lock m, it updates the thread's vector clock $VC(t)$ to be the pairwise maximum of the corresponding timestamps of the two involved vector clocks $VC(t)$ and $VC(m)$, i.e., $VC(t)[i] = max(VC(t)[i], VC(m)[i])$ for all i. For example, when e_4 occurs, $VC(t_2) = max(VC(t_2), VC(m))$. Let $timestamp(t)$ be the *current timestamp* of the thread t. On a thread t releasing a lock m, it copies $VC(t)$ to $VC(m)$ followed by increasing $timestamp(t)$ by 1. For example, when e_3 occurs, $VC(t_1)$ is updated from <1,0> to <2,0>. On a thread t reading from a memory variable x, it copies $timestamp(t)$ to $VC\_R(x)[t]$ of the memory location x followed by performing a write-read race checking (e.g., check if there is a race between e_2 and e_5). Specifically, it checks whether $VC\_W(x)[i] \leq VC(t)[i]$ for all i. On a thread t writing to a memory location x, it copies $timestamp(t)$ to $VC\_W(x)[t]$ followed by performing a write-write race checking (e.g., check if there is a race between e_2 and e_6). and a read-write race checking (e.g., check if there is a race between e_5 and e_6). They check whether $VC\_W(x)[i] \leq VC(t)[i]$ for all i and whether $VC\_R(x)[i] \leq VC(t)[i]$ for all i, respectively. In either case, if the former is larger than $VC(t)[i]$ for some i, it reports a data race (e.g., a race occurs between e_5 and e_6).

Compared to Djit+, *FastTrack* [8] uses *epoch* (a scalar) instead of a vector to represent the vector clock of each write access to each shared memory location, and dynamically swaps between the *epoch* and vector representations to track the read accesses to the same shared memory location. *LOFT* [3] is built on top of *FastTrack*, which further identifies and eliminates a class of vector clock updates and assignments irrelevant to track between-thread happened-before relations online. We denote such a precise online data race detector (e.g., Djit+, FastTrack, or LOFT) by ***detector D***, which will be used in our model to be presented in Section III.

Note that some data races are harmful, which affect the correctness of the program behavior, while the others are benign (i.e., harmless). A race may be intentionally introduced by developers to improve the program performance or produced by compiler optimization that automatically removes some happened-before relations (e.g., empty locking) codified by programmers to order thread execution sequences at certain code regions, and yet the program source code may be too complicated for the static analysis (in a compiler) to detect the usefulness of the lock under certain optimization strategies and thus speculatively remove them, resulting in races that can only appear under certain compiler optimization options.

B. Dynamic Binary Instrumentation

The implementation of dynamic analysis tools relies on the technology of dynamic binary instrumentation [20], which is widely supported by industrial-strength frameworks such as Pin [20]. Instrumentation code is dynamically attached at the run time of the compiled binary files. Such a framework provides Application Programming Interface (API) for programmers to write their own dynamic analysis tools. With such a tool, any process in the user space of a system, including dynamically linked libraries and created codes, can be subject to dynamic analysis.

With dynamic binary instrumentation support, when a race is detected, we report the pair of program instructions in the code listing that generates the two racy memory accesses. We refer to such a pair of program instructions as a **racy pair** [3]. In an execution trace, multiple data races may be detected and reported. For ease of presentation, we refer to this set of racy pair in an execution trace as a **race set**.

C. Compiler Optimization Configuraiton Setting

Many compilers allow developers to turn on different flags when transforming the source code of a program into binary files. For instance, GNU gcc compiler [9] allows developers to turn on the optimization level to –O0 (default), –O1 (level 1 optimization), –O2 (level 2 optimization), –O3 (level 3 optimization), –Os (optimization for size), –Ofast (optimization for speed), and so on [15]. Some subsets of optimization flags are compatible, which can be turned on together. For instance, –O0 and –g (enabling debugging) can be turned on together. We refer to such a valid combination of flags as a configuration setting.

III. SDA-Cloud

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

A. Overview

Figure 2 shows the architectural blueprint of SDA-Cloud, which consists of a static virtual machine (VM) image (which

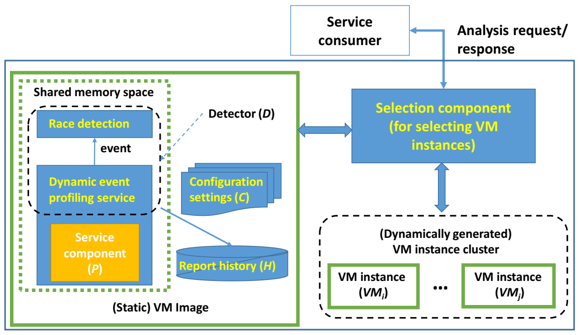

Service consumer
Analysis request/ response
Shared memory space
Race detection
event
Detector (D)
Dynamic event profiling service
Configuration settings (C)
Service component (P)
Report history (H)
(Static) VM Image
Selection component (for selecting VM instances)
(Dynamically generated) VM instance cluster
VM instance (VM_i)
...
VM instance (VM_j)

Figure 2. Architecture of SDA-Cloud

is created and administrated[4] by SDA-Cloud) and a selection component. A service component P is modeled as a multithreaded program that accepts an input and computes a corresponding output. The source code of P and its test cases are both located in the same VM image. The runtime of P is executed in the user space of the system running on that VM. The service component under profiling shares the memory space with the race detection module, which provides an efficient within-process data transfer (by means of function invocation and pointers to shared memory locations) for the required dynamic analysis. This joint process is part of a VM instance. A compiler configuration setting c is a string, which represents the optimization flags enabled while compiling the service component P.

The procedure of running SDA-Cloud consists of two major phases: initialization (Phase 1) and analysis (Phase 2). Upon receiving an analysis preparation request message from a service consumer, Phase 1 creates the infrastructure needed for the dynamic analysis to be conducted in Phase 2. Phase 1 initializes not only VM instances (VM_i in Figure 2) from a VM image but also the configuration setting of the compiler in each VM instance, compiles the service component using the user-provided scripts, and links it to the detector D. Services computing advocates the pay-per-use model [28]. To better allocate the test budget to multiple configuration settings, SDA-Cloud computes an analysis-aware weight for each configuration setting and uses a probabilistic approach to select corresponding VM instances to conduct the race detection. With the historic data of race detection results, SDA-Cloud periodically adjusts the weight of each configuration setting to guide the follow-up detection periods. SDA-Cloud also maintains the necessary data structures to facilitate this probabilistic selection strategy as well as to conduct the dynamic analysis (race detection).

B. Phase 1: Initialization

In SDA-Cloud, an analysis VM image is modelled as a 4-tuple (P, C, D, H), where P is a service component, C is a set of compiler configuration settings, D is a dynamic race detector with dynamic binary instrumentation support, and H is the *report history* reported by D. A report history H is a sequence of race sets, initially empty (i.e., $H = \langle\rangle$). Moreover, once a compiler is chosen (which should be predefined in any service engineering projects; otherwise, it is impossible to start developing the source code listing of P), it is straightforward to define the set of configuration settings because, to the best of our knowledge, many software projects define similar sets of configuration settings (e.g., as defined in the makefiles [7] of these projects).

Given an analysis VM image (P, C, D, H), an instance of the VM image (a VM instance for short) is a 5-tuple (F, c, D, H, b), where $c \in C$ such that the source code of P is compiled with the compiler in the VM instance using the configuration setting c, producing one corresponding version of binary file denoted as F. b is the upper bound budget ratio of the VM instances of configuration setting c specified by the service consumers. Because SDA-Cloud selects VM instances to execute with probabilities, one possible case is that some VM instances may be selected excessively and others are at the risk of starvation. To avoid this scenario, SDA-Cloud allows service consumers to specify b as the maximum ratio of total budget to be distributed on the VM instances of the configuration setting c.

To initialize the dynamic analysis infrastructure, SDA-Cloud creates a set of VM instances based on the analysis VM image (P, C, D, H), one[5] for each configuration setting c in C. It keeps all these instances as the children of a service selection component of SDA-Cloud to simplify the service discovery process[6] (will be presented in the next sub-section). The above initialization is performed when SDA-Cloud receives an analysis preparation request (exposed as an API call), which includes the maximum **budget *B***[7]and the upper bound budget ratio b of each configuration setting being invoked and analyzed. This value B is the total number of analysis invocations to be performed across all VM instances to be used in Phase 2.

After initialization has been done, SDA-Cloud then replies the service consumer with the set of configuration settings that have been successfully used in initializing VM instances in SDA-Cloud.

C. Phase 2: Analysis

SDA-Cloud maintains a number of metrics for its analysis-aware service selection strategy.

For each VM instance, SDA-Cloud collects two metric values N_i and R_i: Given an instance $VM_i = (F_i, c_i, D, H_i, b_i)$,

[4] We note that many cloud management tools like VMware allow a client program to create new VM or import system images, and provide APIs for the client program to control and probe the status of the VM instance as well as to send commands to the user or system processes in that VM instance.

[5] This can be further generalized to a set of VM rather than merely one VM. However, for modeling, using one VM suffices to concisely present the main idea of our work.

[6] To the best of our knowledge, the notion of UDDI for a service has been abandoned by the industry.

[7] For instance, in existing work on dynamic analysis for race detection [3][8], n is 100 to 1000 per combination of binary image and a test input.

SDA-Cloud keeps track of the number of times N_i that the instance VM_i has been used to analyze the executions of F_i. The sets of races detected from different executions of F_i may be different. Thus, SDA-Cloud also computes the set of *unique* data races R_i kept in H_i. N_i is obtained by counting the length of H_i, and R_i is obtained by taking a set union of all the race sets kept in H_i. Initially, N_i is 0 and R_i is an empty set. SDA-Cloud also maintains the number of invocations that have been performed as S, which initially is 0.

With above metrics, SDA-Cloud periodically monitors the progress of race detection under each configuration setting and uses a probabilistic approach to select and execute VM instances. The length of analysis period is specified by service consumers as m number of VM instance invocations (If the remaining invocation budget B - S is smaller than m, we set m as $B - S$).

Before each analysis period starts, SDA-Cloud assigns each VM instance a weight p_i, which indicates the capability of detecting data races with this VM instance. p_i is computed as follows. SDA-Cloud firstly calculates an intermediate value p_i' as $|R_i| \div N_i$ for each VM instance which provides an historic estimate on the effectiveness of using that VM instance to dynamically detect data races. It then normalizes p_i' of each VM instance against the sum of p_i' of all VM instances and denotes the normalized value as the weight p_i. In this way, the sum of p_i of all VM instances is 1.

When an analysis period begins, the selection component in SDA-Cloud adopts a weighted-random strategy to select VM instances for m times. That is, each VM_i is selected with the probability of p_i in each of the m selections of this period. The randomness property ensures that each VM instance has the chance to be selected. The weight setting makes those VM instances with larger p_i can be selected with higher probability. If the selection number of an instance VM_i has reached the upper bound $b_i \times m$, SDA-Cloud excludes this VM_i in the remaining selections of this period. For each selection, after determining which VM instance (F_i, c_i, D, H_i, b_i) to be selected, SDA-Cloud invokes the detector D in VM_i. The detector D then executes the binary file F_i according to the scripts provided by the developers (set up when preparing the VM image), monitors the execution traces, and reports the set of detected races r.

After one analysis period is done, SDA-Cloud updates the analysis metric values as follows: increases S by m, appends r to the report history H_i of VM_i, and increases N_i for this VM instance by 1. If $B > S$, SDA-Cloud will then start the next analysis period.

D. Discussion

Data races are due to improper accesses of concurrently executing threads on shared memory locations. Due to the large number of memory accesses, race detection techniques incur great overhead (e.g., 100 times slowdown [15]) to program executions. To the best of our knowledge, the existing work all deploys the runtime of race detection and the program under detection in the same memory space of same machine. Cloud platform provides a great amount of resources to software testing in general (bug detection in particular). According to a recent survey on software testing in cloud [16], testing parallelization is a hot topic studied in the community. Cloud9 [5] is an existing cloud-based testing service, which speeds up the testing process by dynamically partitioning the symbolic execution of program under test, which is originally executed in one computing node, into multiple computing nodes. For race detection, however, such a partition idea is impractical due to the high cost of maintaining global state by transferring a large number of events. Our SDA-Cloud saves the transfer cost by putting the runtime of program under detection and race detection in the same memory space. Also, SDA-Cloud partitions the testing process against different configuration settings and proposes an analysis-aware strategy to dynamically allocate the testing budget to different testing units. The analysis-aware selection strategy enables SDA-Cloud to not only allocate more budgets to those VM instances that are estimated to be able to detect more unique races but also give chances to those VM instances that currently detect less races.

The following evaluation section shows that our SDA-Cloud model is feasible and practical to deploy the dynamic analysis (race detection in particular) as a cloud-based service.

IV. Evaluation

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

A. Implementation

We had implemented a prototype of SDA-Cloud on a Dell server to support our experimentation. This server was equipped with 32 Intel Xeon 2.90 GHz cores and 128 GB physical memory. The VM image was configured to run the 32-bit desktop Ubuntu Linux 12.04.1 operating system with 3.16GHz Duo2 processor and 3.8GB physical memory. This platform provided the genuine thread control of the execution of each program benchmark in the experiment. Our previous work [3][4][15] had demonstrated that this platform was able to repeat the third-party experimental results on dynamic race detection.

The prototype was able to receive requests (commands) from users, create and select VM instances, execute a detector in each VM instance, and collect race detection results. We configured the race detector D by implementing an existing dynamic race detector LOFT [3] on top of Pin [20]. 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.

Our prototype has not fully implemented SDA-Cloud. Specifically, we have not fully implemented the automatic addition of the configuration setting string to the makefile of each benchmark. When we performed the experiment, we found that different subjects used different kinds of configuration scripts and these scripts were written in different

Table 1. Descriptive statistics of the benchmark

Benchmark		Application Domain	SLOC	Object Code Size (in '000 bytes) under Each Configuration Setting					
				–O0	–O1	–O2	–O3	–Ofast	–Os
PARSEC 3.0	blackscholes 2.2	Financial Analysis	914	18	21	21	23	24	21
	swaptions 2.1	Financial Analysis	1119	54	93	88	94	91	58
	canneal 2.2	Engineering	2825	415	415	415	415	415	415
	vips 7.22.1	Media Processing	138536	7460	9770	10704	12248	12210	8059
	raytrace 1.2	Rendering	13764	16311	18378	19606	20366	20166	16139
	bodytrack 2.0	Computer Vision	16423	2493	3305	3245	3445	3435	2118
	dedup 3.0	Enterprise Storage	3347	222	333	358	388	389	252
	streamcluster 2.2	Data Mining	1769	57	77	80	103	104	65
Real world	mysql 5.0.92	RDBMS	996920	18368	25867	28089	29571	29575	24051
	httpd 2.0.65	Web Server	201280	1375	1521	1640	1709	1711	1460

types of languages and styles. We therefore leave the preparation of scripts with different configuration settings to the developers who are in charge of maintaining these scripts to compile the corresponding subjects. We will investigate a generalized approach to make this step automatic in the future.

B. *Benchmark and their Binaries*

The benchmark we used consists of a set of widely-used subjects including subjects of PARSEC 3.0 benchmark, the popular database system mysql [22] and the web server httpd [12] as the service components. These subjects have also been widely used in existing concurrency bug detection work (e.g., [2][3][4][19]). We also consider that these subjects can represent different service components in the technology stack of a service. The PARSEC 3.0 benchmark contains 13 subjects in total: blackscholes, bodytrack, canneal, dedup, streamcluster, swaptions, vips, raytrace, freqmine, facesim, ferret, fluidanimate, and x264. The last five subjects (i.e., freqmine, facesim, ferret, fluidanimate, and x264) were excluded from our experiment due to the following reasons: freqmine does not use the Pthread library, we discarded it because our implementation of the race detector was built on top of the Pthread library; facesim, ferret, fluidanimate and x264 crashed when we ran them with our race detector under the Pin environment on our platform. Thus, we included the remaining 8 subjects in our experiment and executed them with the *simsmall* test suite and eight worker threads as what the existing work (e.g., [3][32]) did. This test suite has also been widely used to run as a whole to execute each PARSEC subject in previous studies (e.g., [3][13][26]). For each subject, this test suite contains a set of correlated test cases. We were unaware of any document to specify how to separate the test suite into individual test cases. Thus, to guarantee the correct usage of the test suite, we executed it as a whole against each PARSEC subject. By doing so, a test run of such a subject in our study refers to an execution of the whole test suite instead of an individual test case.

For the mysql database and the httpd web server, we adopted the test inputs used in the existing work [3].

We compiled each benchmark using each of the six configuration settings stated in Section II.C to generate one compiled service component. In total, we generated 60 binary versions of these subjects. The sizes of these binaries (measured by *du* [6]) are shown in Table 1.

C. *Experimental Procedure*

For each subject, we set the budget B to 120. For Phase 1, we requested the prototype to set up six VM instances, one for each configuration setting. We wrote a small program module to instruct SDA-Cloud to conduct Phase 2. To gain statistical power, we repeated the above procedure for 1000 times. We automated the whole process so that there was no need of human intervention to collect the whole set of data from the experiment. To demonstrate the feasibility of our model, we measured the mean number of unique data races reported by each VM instance.

D. *Results*

1) *Qualitative analysis on SDA-Cloud software architecture*

Our prototype can largely complete the experiment with little human effort. It indicates that it is feasible to instruct the dynamic analysis detector in VM instances to profile and analyze executions and send the results to the prototype of SDA-Cloud. However, to complete the whole experiment, it took more than one whole month. Apparently, it indicates that a real system of SDA-Cloud needs to use a cluster of VM instances per configuration setting instead of one VM instance. This change can be supported by extending the current architecture of SDA-Cloud. Recall that in our model, a VM instance may be selected and executed multiple times in an analysis period. Currently our implementation has only one VM instance for one configuration setting. Thus, if one VM instance of some configuration setting is selected multiple times in one analysis period, the invocations of this VM instance have to be sequentially executed. A real system can be designed to dynamically create multiple VM instances for the same configuration setting to conduct multiple analyses concurrently. However, as we have mentioned before, the current implementation can be trivially extended by creating a set of identical VM instances for each configuration setting.

Besides, in our experiment, a VM instance has once become non-responsive (for reasons out of our control). In this case, the VM instance was terminated and re-initialized. In our model, for a VM instance (F_i, c_i, D, H_i, b_i), the binaries F_i and the detector D were preloaded in the image of the VM

Table 2. Data races reported by each VM instance

Benchmark	Number of Reported Races from each VM Instance					
	–O0	–O1	–O2	–O3	–Ofast	–Os
blackscholes 2.2	0	0	0	0	0	0
swaptions 2.1	0	0	0	0	0	0
canneal 2.2	1	1	1	1	1	1
vips 7.22.1	11	11	11	11	11	8
raytrace 1.2	3	3	4	3	3	ND
bodytrack 2.0	5.1	5.2	5.1	5	5	5.1
dedup 3.0	11.2	11.3	11.3	11	11	12.6
streamcluster 2.2	27.7	29.1	15.7	15.8	15.5	25.8
mysql 5.0.92	29.4	37.1	36.7	35.1	36.4	ND
httpd 2.0.65	32.0	65.8	73.4	69.6	67.5	85.4
Total	120.4	163.5	158.2	151.5	150.4	137.9

instance, the data H_i and b_i were kept in the prototype application of SDA-Cloud rather than in the VM instance, and our prototype of SDA-Cloud identified a VM instance by the configuration setting string. Therefore, our architecture can directly support the re-initialization of VM instances.

A subject may become unable to proceed further (e.g., due to their poor software design). A typical dynamic detector usually incorporates a timeout mechanism to make the process timeout, terminate the current execution of the detector, and restart it again. The design of our model has not considered this scenario yet. On the other hand, in our evaluation, we had not encountered this scenario.

In Phase 1, SDA-Cloud requires each subject to be compiled. We found that on using the configuration setting –Os, two benchmarks (raytrace and mysql) were not compiled successfully (we filled the mark "ND" in the corresponding cell in Table 2). In our model, Phase 1 can notify service consumers which VM instances can be used so that the statistics will not confuse the consumers.

2) *Quantitative Analysis of SDA-Cloud*

Table 2 shows the mean number of data races detected by each VM instance at the end of the experiment (i.e., 1000 trials and budget $B = 120$ for each trial). The results show that using different VM instances can detect different numbers of unique data races. It is especially the case on the two real-world benchmarks. We have also analyzed the overall number of unique data races detected from the all six VM instances. Our data analysis shows that for the benchmarks from top to bottom listed in Table 2, the numbers of unique races are 0, 0, 1, 11, 7, 10, 14, 33, 41, and 163, respectively. The data show that SDA-Cloud is a viable approach to detect data races.

We had also checked the amount of unique data races detected across all these VM instances. We observed that the number of detected data races increased rapidly with respect to the first five analysis requests of each VM instance, and then it became not quite frequent to detect additional data races. This finding is interesting, which indicates that the pay-per-use model allows service consumers to examine the current results and ends the analysis session if the amount of races detected does not increase frequently so as to save the cost involved.

As expected, we have also observed that the time needed to instruct VM instances and maintain the result statistics is 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 generates very small amount of data transfers per analysis request. On the other hand, each execution of a service component generated hundreds of millions of events (which is consistent to the findings reported in [3][8]), which significantly slow down each round of analysis [20].

V. RELATED WORK

As to our work, one closely related area is the testing as a service on and for cloud computing. We have reported a comprehensive survey [16] on this topic. For brevity, we revisit some most representative ones in this section. Zhu and Zhang [33] propose to use existing tools as individual services and use a service orchestration approach to integrate these services. Yu et al. [30][31] propose to organize different tasks by using different testing strategies so as to consider the environment factors in testing in the cloud. Candea et al. [5] propose Cloud9, which applies symbolic execution to generate test cases and expose errors. In certain sense, Cloud9 can also be considered as a dynamic analysis as a service on the cloud. Different from Cloud9, our work focuses on VM selection for data race detection and Cloud9 does not support changing configuration settings definable by service consumers. Both Candea et al. [5] and Jenkins et al. [14] interestingly propose models that can be customized according to different usage scenarios. Tsai et al. [27] propose a new service composition to testing. Our work 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 has used a kind of static software architecture to organize the service selection, VM instances, the relationship between the subject under analysis and the dynamic race detector. Yan et al. [29] also take 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., [18][23]) or generation of test inputs [24]. Hou et al. [11] also apply the notion of quota when invoking a service. 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. [34] test services involved in a dynamic service discovery mechanism, but our work does not involve such a mechanism.

It is worthy of note that there are also model-based approaches to organize services for testing as a service [35].

VI. Conclusion

In this paper, to the best of our knowledge, we have presented the first work to offer trace-based dynamic analysis as a service using a multi-VM architecture approach. We have casted our work to dynamic online data race detection on service components. Our approach respects the holistic approach required by dynamic binary instrumentation that underpins such detectors. Such a detector and the service component should share the same memory space so as to avoid incurring unacceptable slowdown required to transmit huge amount of data access events (usually in the order of billions or more) between the profiling module and the detection module. Unlike other approaches on testing as a service, our architecture approach uses a VM instance as an atomic unit, and provides an analysis-aware selection among VM instances to fulfill a service request. We have evaluated our work by implementing a prototype for it. Although our prototype is not comprehensive, yet it suffices to demonstrate the feasibility of this approach. The evaluation has shown that our approach can be practical. In the future, we will study different service selection strategies, compare them, and evaluate the scalability of this approach.

VII. Acknowledgement

This work was partially supported by the Research Grants Council of Hong Kong SAR (project numbers 11201114, 111313, 125113, and 123512). This work was submitted to the research area of service oriented software engineering for peer review. We are grateful for the invaluable comments provided by the anonymous reviewers.

References

[1] T. Ball and J. R. Larus, "Efficient path profiling," *Proc. MICRO*, pp. 46–57, 1996.

[2] Y. Cai and W.K. Chan, "Magiclock: Scalable detection of potential deadlocks in large-scale multithreaded programs," *IEEE TSE*, vol. 40, no. 3, pp. 266–281, 2014.

[3] Y. Cai and W.K. Chan, "Lock trace reduction for multithreaded programs," *IEEE TPDS*, vol. 24, no. 12, pp. 2407–2417, 2013.

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

[5] G. Candea, S. Bucur, and C. Zamfir, "Automated software testing as a service", *Proc. SoCC,* pp. 155−160, 2010.

[6] *Du (Linux/Unix command): Estimate file space usage*, http://linux.about.com/library/cmd/blcmdl1_du.htm

[7] S. I. Feldman, "Make - A program for maintaining computer programs", *SPE*, vol. 9, no. 4, pp. 255–265, 1979.

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

[9] *GCC Compiler*, http://gcc.gnu.org/

[10] K. Gottschalk, S. Graham, H. Kreger, and J. Snell, "Introduction to Web Service Architecture", *IBM Systems Journal*, vol. 41, no. 2, pp. 170–177, 2002.

[11] S.-S. Hou, L. Zhang, T. Xie, and J.-S. Sun, "Quota-constrained testcase prioritization for regression testing of service-centric systems," *Proc. ICSM*, pp. 257– 266, 2008.

[12] *Httpd Web Server 2.0.65*, http://httpd.apache.org/

[13] A. Jannesari, K. Bao, V. Pankratius, and W.F. Tichy, "Helgrind+: An efficient dynamic race detector," *Proc. IPDPS*, pp. 1–13, 2009.

[14] W. Jenkins, S. Vilkomir, P. Sharma, and G. Pirocanac, "Framework for testing cloud platforms and infrastructures," *Proc. CSC*, pp. 134−140, 2011.

[15] C. Jia, and W. K. Chan, "A study on the efficiency aspect of data race detection: a compiler optimization level perspective," *Proc. QSIC*, pp. 35–44, 2013.

[16] C. Jia, Y. Cai, Y.T. Yu, and T.H. Tse. "5W+1H pattern: A perspective of systematic mapping studies and a case study on cloud software testing", *JSS*, 2015.

[17] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *CACM*, vol. 21, no. 7, pp. 558–565, 1978.

[18] B. Li, D. Qiu, H. Leung, and D. Wang, "Automatic test case selection for regression testing of composite service based on extensible BPEL flow graph," *JSS*, vol. 85, no. 6, pp. 1300–1324, 2012.

[19] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," *Proc. ASPLOS*, pp. 329–339, 2008.

[20] 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," *Proc. PLDI*, pp. 191–200, 2005.

[21] L. Mei, W. K. Chan, and T. H. Tse, "An adaptive service selection approach to service composition," *Proc. ICWS*, pp. 70–77, 2008.

[22] *MySQL 5.0.92*, http://www.mysql.com/

[23] C.D. Nguyen, A. Marchetto, and P. Tonella, "Test case prioritization for audit testing of evolving web services using information retrieval techniques," *Proc. ICWS,* pp. 636–643, 2011.

[24] Y. Ni, S.-S. Hou, L. Zhang, J. Zhu, Z.J. Li, Q. Lan, H. Mei, and J.-S. Sun, "Effective message-sequence generation for testing BPEL programs," " *IEEE TSC,* vol. 6, no. 1, pp. 7–19, 2013.

[25] E. Pozniansky and A. Schuster, "MultiRace: efficient on-the-fly data race detection in multithreaded C++ programs," *CCPE*, vol. 19, no. 3, pp. 327–340, 2007.

[26] P. Thomson, A.F. Donaldson, and A. Betts, "Concurrency testing using schedule bounding: an empirical study," *Proc. PPoPP*, pp. 15–28, 2014.

[27] W.-T. Tsai, P. Zhong, J. Balasooriya, Y. Chen, X. Bai, and J. Elston, "An approach for service composition and testing for cloud computing", *Proc. ISADS*, pp. 631−636, 2011.

[28] C. Weinhardt, A. Anandasivam, B. Blau, and J. Stößer, "Business models in the service world", *IT Professional*, vol. 11, no. 2, pp. 28–33, 2009.

[29] M. Yan, H. Sun, X. Wang, and X. Liu, "Building a TaaS platform for web service load testing," *Proc. CLUSTER*, pp. 576−579, 2012.

[30] L. Yu, W.-T. Tsai, X. Chen, L. Liu, Y. Zhao, L. Tang, and W. Zhao, "Testing as a service over cloud," *Proc. SOSE,* pp. 181−188, 2010.

[31] L. Yu, X. Li, and Z. Li, "Testing tasks management in testing cloud environment, " *Proc. COMPSAC*, pp. 76−85, 2011.

[32] K. Zhai, B. Xu, W.K. Chan, and T.H. Tse, "CARISMA: a context-sensitive approach to race-condition sample-instance selection for multithreaded applications," *Proc. ISSTA*, pp. 221–231, 2012.

[33] H. Zhu and Y. Zhang, "Collaborative testing of web services," *IEEE TSC,* vol. 5, no. 1, pp. 116–130, 2012.

[34] K. Zhai, B. Jiang, W.K. Chan, and T.H. Tse, "Taking advantage of service selection: A study on the testing of location-based web services through test case prioritization," *Proc. ICWS.,* pp. 211–218, 2010.

[35] L. Zhang, X. Ma, J. Lu, T. Xie, N. Tillmann, P. de Halleux. Environmental modeling for automated cloud application testing. *IEEE Software*, vol. 29, no. 2, pp. 30−35, 2012.