JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 34, 1287-1306 (2018)
DOI: 10.6688/JISE.201809_34(5).0011

A Concurrent Approach for Improving the Efficiency of Android CTS Testing

CHIEN-HUNG LIU, WOEI-KAE CHEN AND SHU-LING CHEN*
Department of Computer Science and Information Engineering
National Taipei University of Technology
Taipei, 106 Taiwan
E-mail: {cliu; wkchen}@ntut.edu.tw
**Department of Industrial Management and Information*
Southern Taiwan University of Science and Technology
Tainan, 700 Taiwan
E-mail: slchen@stust.edu.tw

The Compatibility Test Suite (CTS) is a set of JUnit tests presented by Google to help manufactures to ensure if their Android devices are in compliance with the Android compatibility standards. However, the CTS contains a huge number of test cases and, hence, it usually would take several hours to complete the CTS tests. This could seriously affect the development schedule of Android devices, especially when the CTS test is included in the daily system integration. To reduce the time to perform CTS tests and shorten the time-to-market of Android devices, this paper presents a concurrent approach for improving the CTS test efficiency. Particularly, the CTS tests are decomposed into multiple tasks that are executed on different devices concurrently. The execution results of the devices are then merged to generate the CTS test report. In addition, various task scheduling algorithms are employed and different partitioning methods are presented in the approach. A cloud-based testing platform is developed to support the proposed approach. To evaluate the effectiveness of the proposed approach, several empirical experiments were conducted. The experimental results show that the efficiency of CTS test can be much improved as the number of devices increases. Moreover, the results also indicate that the Longest Job First scheduling and mixed partitioning methods can result in better test efficiency.

Keywords: Android testing, software testing, Android compatibility testing, compatibility test suite, CTS

1. INTRODUCTION

Recently, with the widespread popularity of smartphone, the number of Android devices has increased tremendously. According to statistics, the installed base of Android smartphones worldwide in July, 2016 was over 2.156 billion [1]. Particularly, according to Gartner, Android has about 84.1% market share of the global smartphone operating systems in 2016 [2]. From these statistics, it can be observed that Android has become the top-selling mobile device platform and will continue to dominate the smartphone market in the near future.

As the open-source Android platform is continuously getting popular worldwide, it has attracted many smartphone manufacturers to develop Android devices. According to [3], there are 4,000 unique Android devices on the market from more than 400 manufac-

Received February 2, 2017; revised March 22, 2017; accepted April 22, 2017.
Communicated by Ren-Hung Hwang.

turers in 2015. In particular, these devices often have different versions of Android operating system because Android has evolved very quickly in recent years and has more than 10 major versions so far. Moreover, different types of Android devices, even from the same manufacturer, usually have customized hardware implementations or options, such as different sizes and resolutions of screens, cameras, and network configurations. This can raise a concern about the compatibility of Android devices called *Android fragmentation* [4, 5] which can result in poor user experiences.

For building a healthy ecosystem to ensure that the Android devices developed by diverse manufactures are compatible so that applications can be executed on the variety of compatible Android devices correctly, Google has presented the Android compatibility program. The core of the program is composed of the Compatibility Definition Document (CDD) [6] and the Compatibility Test Suite (CTS) [7]. The CDD enumerates the software and hardware requirements of a compatible Android device and should be followed by the device manufactures. The CTS is used to ensure that an Android device meets the requirements of the CDD. That is to verify whether an Android device is in compliance with the CDD. Note that achieving compatibility (*i.e.*, passing CTS test) is a prerequisite for Android devices to have access to Google Play and Google Mobile Services [8]. In addition, CDD and CTS will have different version every time when the Android operating system is updated.

The CTS consists of two major components: *a test harness* that runs on a desktop for managing the test execution and *a set of JUnit test cases* which are packaged as .apk files and executed on the target device attached to the desktop. The CTS test cases are organized hierarchically into various test packages and classes to cover different areas of the Android platform for ensuring the compatibility of the devices. These CTS test cases are usually executed in the daily system integration process when building a device in order to reveal the possible incompatibilities as soon as possible.

The CTS testing is important since Android device manufactures have to submit the CTS test report of a newly built device to Google in order to get the approval for installing the Google Play application on the device. However, CTS is a commercial-grade test suite that currently contains over 20,000 test cases and expects to grow continuously. Thus, it usually takes at least several hours to execute all the test cases of CTS for most Android devices although the test execution time is device dependent. For example, it may take 5~6 hours to complete the CTS tests on a HTC One smartphone. As the consequence, the development schedule of Android devices could be seriously affected if the CTS test is included in the daily system integration or continuous integration testing of the device implementation.

To reduce the execution time of CTS for shortening the time-to-market of Android devices, this paper presents an approach to improve the efficiency of CTS tests. Particularly, the proposed approach partitions the CTS test suite into multiple testing tasks and executes these tasks on multiple Android devices concurrently. The test results on different devices are then collected and merged together to generate the CTS test report. In addition, to improve the CTS test efficiency further, the proposed approach also supports various task scheduling algorithms and task partitioning methods. A cloud-based testing platform is developed to support the proposed approach.

To evaluate the effectiveness of the proposed approach, several empirical experiments were conducted. The experimental results show that the efficiency of CTS tests can be improved as the number of devices increases. Moreover, the results also indicate that the Longest Job First scheduling and the mixed partitioning methods can result in better test efficiency.

The rest of the paper is structured as follows. Section 2 briefly reviews existing work related to our approach. Section 3 presents the proposed approach to partitioning and performing the CTS tests concurrently. Section 4 describes the design of the cloud-based testing platform for supporting the approach. Section 5 presents and discusses the experimental results. Section 6 provides concluding remarks and future work.

2. RELATED WORK

Due to the diversity of Android devices and operating systems, the Android fragmentation problem has increasingly captured the attention of software developers. Thus, Android compatibility testing has drawn much attention recently. However, most of existing work focuses on the compatibility of Android applications which is to test and verify whether an application can be executed correctly on different Android devices. Test cases for such compatibility testing can be application dependent. On the other hands, the CTS testing concerns about the compatibility of an Android device which is to verify if the CDD has been carried out correctly by the device. Test cases for the CTS compatibility is dependent of Android operating systems. The following briefly describes several studies related to CTS testing and our work.

Wei *et al.* [5] studied the symptoms and root causes of 191 real-world fragmentation-induced compatibility (FIC) issues collected from popular open source Android applications. They found that API evolution and problematic hardware driver implementation are two dominant root causes of FIC. Based on the study, they also reported that checking device information and component availability before invoking certain APIs is the common pattern used for fixing the FIC issues in practice. Further, an API-context pair model has been proposed to capture the common patterns in fixing FIC issues. Based on the model, a static analysis technique, called FicFinder, has been designed to detect the FIC issues automatically in Android applications. The evaluation of FicFinder on 27 large-scale subjects has indicated that FicFinder can uncover many FIC issues and provide useful information to developers.

Ham and Park [9] proposed a mobile application compatibility test system design for Android fragmentation. Particularly, they have proposed two methods that aim to detect the fragmentation problems at the code level and at the API level, respectively. In the proposed methods, the source code of the mobile applications are analyzed. Those code statements that might be dependent on the Android devices or APIs are extracted and compared with the compatibility database of each device. Based on the proposed methods, a system has been designed to test the compatibility of mobile applications. Further, a case study is presented in [10] to illustrate the effectiveness of the proposed methods.

Kaasila *et al.* [11] presented an online testing platform, called Testdroid, for Android applications. The Testdroid allows users to record their interactions with the GUI of an Android application and generate a corresponding test script automatically. The test script then can be uploaded to the platform along with the application and is executed on a set of available physical devices concurrently. The test results across multiple devices then can be accessed through the platform. By analyzing the test results, the compatibility issues of the application among different Android devices can be identified and resolved by developers.

Huang [12] presented a mobile application automated compatibility testing service, called AppACTS. The AppACTS has a master/slave architecture based on the Hadoop Distributed File System (HDFS). The master node is responsible for controlling the testing process. Each slave node is responsible for executing tests and connects a mobile server that controls the physical devices. The users of AppACTS can upload their mobile applications, select the target devices, perform the tests, and view the test results. A set of commands, such as install/uninstall, start, press, and swipe, are provided and users can use these commands to execute and test the applications.

Zhang *et al.* [13] proposed a mobile compatibility testing method that can generate a compatibility test sequence for a set of devices in order to cover most of the compatibility features, such as camera, screen resolution, and network connections, and reduce the test cost of Android compatibility testing. In particular, a tree model is proposed to represent the compatibility features. Based on the tree model, the set of devices are clustered using the K-Means algorithm. The clusters are then ranked by the market shares of the devices. Finally, one popular device is selected form each cluster in sequence for the mobile compatibility testing.

Liu *et al.* [14] presented a Cloud Testing Platform (CTP) that can execute the compatibility testing for Android applications automatically on a set of real devices concurrently. The CTP is based on the OpenStack cloud platform [15]. For each device, the CTP will allocate a corresponding virtual machine (VM) to interact with the device for executing the tests. To perform the compatibility testing, the users need to select the target devices and upload the test script and the application under test to CTP. The CTP then executes the tests on the selected devices in parallel and generates the test reports, including the screen snapshots and video files of the tests.

Joorabchi *et al.* [16] proposed a technique, called CHECKCAMP, to automatically detect the inconsistencies between the iOS and Android implementations of the same application. Specifically, the technique first instruments the application on iOS and Android platforms and generates traces for a set of user scenarios. By capturing and analyzing the traces, an abstract model for each platform then can be inferred. The abstracted models are compared to expose any differences using a set of code-based and GUI-based criteria. A visualization of the models is produced to show any detected inconsistencies between the different implementations. An evaluation was conducted to show the effectiveness of the proposed approach.

Liu *et al.* [17] proposed a strategy to allocate Android physical devices efficiently among the compatibility testing tasks. Specifically, they have abstracted the components of testing platforms into testing models. Based on the models, they propose a wait-time fairness scheduling strategy to allocate devices among the testing tasks. A prototype tool, called Apsaras, is presented and several experiments were conducted using 80 physical devices. The results show that the proposed approach can allocate devices more efficiently and more fairly as compared with traditional methods, such as first-come-first-served.

In addition to above researches, there are several Android application testing services available on the market [18-22]. These testing services provide various automated tests and allow users to test their mobile applications on a set of selected devices in parallel. By using these services, the compatibility test cost of mobile applications can be greatly reduced.

3. THE APPROACH OF CONCURRENT CTS TESTING

This section presents the proposed approach of concurrent CTS testing. The strategies for CTS test task scheduling and partitioning which may affect the efficiency of the approach are also described.

3.1 The Approach of Concurrent CTS Testing

As compared with the techniques that prioritize or select only a subset of test cases to perform a test, one way to improve the test efficiency is to perform the test in parallel by dividing a single test job into multiple tasks and executing the tasks concurrently to shorten the overall test time at the expense of increasing the test cost. Especially, as cloud testing becomes prevalent, the cost of concurrent test executions can be largely reduced by leveraging the characteristics of cloud computing, such as elastic provision, virtualization, and pay-as-you-go price plan [23]. Thus, dividing and concurrently executing tests would become an attractive alternative to improve test efficiency if the potential benefit of increasing test efficiency is larger than the additional test cost.

Since Android CTS is a set of JUnit tests that have been organized hierarchically in terms of test packages and classes, it would be possible to partition the CTS tests into multiple test packages (or classes) and execute the partitioned packages (or classes) in parallel if there has no dependency between the packages (or classes). Therefore, to improve CTS test efficiency, this paper adopts the notion of parallel test execution and proposes a concurrent CTS testing approach using cloud. As shown in Fig. 1, the main idea of the proposed approach is to partition the CTS tests into multiple testing tasks and execute the tasks on different devices concurrently. The test results of the tasks are then merged to generate the CTS test report for the devices under test.

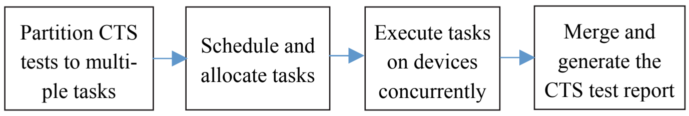

Partition CTS tests to multiple tasks
Schedule and allocate tasks
Execute tasks on devices concurrently
Merge and generate the CTS test report
Fig. 1. The process of concurrent CTS testing.

Fig. 2 shows the overview of the testing platform developed to support the concurrent CTS testing approach. A user can submit a CTS test request to the Integration Server on demand or schedule the CTS integration tests periodically through a web-based user interface. The CTS test packages (or classes) are then checked out from the repository and executed concurrently on multiple Android physical devices under test according to specified configurations. Each physical device is controlled by a VM of the cloud environment through wireless connection. The execution results of the devices are collected and combined to generate a CTS test report in XML format. The test log is also recorded.

Once the CTS test is completed, the user can see the test result of each device and the integrated CTS test report from the web-based user interface.

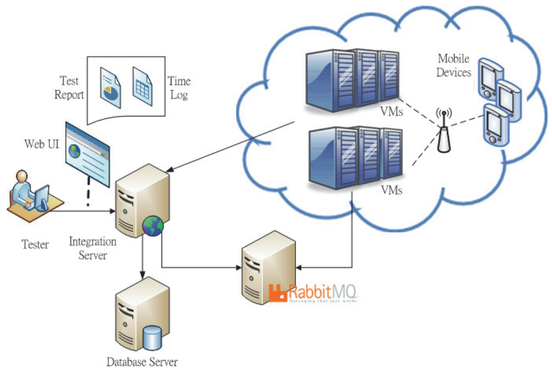


Fig. 2. The overview of the concurrent CTS testing platform.

3.2 The Strategy of Task Scheduling and Partitioning

Although parallel execution intuitively can improve the efficiency of CTS tests, the amount of improvement can be affected by many factors, such as the number of devices used in the test, the algorithms of task scheduling, and the number and the size of the partitioned tasks. The followings describe the strategies used to schedule and partition the CTS testing tasks in the proposed approach.

3.2.1 The number of devices and the size of tasks

It is not surprising that the test efficiency would be improved as the number of devices used in the concurrent CTS testing increases. However, the speedup improvement usually becomes saturated gradually when the number of devices exceeds a certain threshold. The saturation of speedup may result from the overheads of concurrent processing, such as task partitions, communications, and result integration. Thus, to be cost-effective for concurrent CTS tests, the proposed approach monitors the speedup tendency and conducts experiments to obtain the expected saturation point for the number of devices used in the concurrent CTS testing.

Moreover, the efficiency can also be affected by the number and size of the tasks allocated to the devices. The number and size of tasks would depend on the task partitioning methods which are described in section 3.2.3. As of the size measurement unit, the proposed approach uses the test execution time to measure the size of the task, not the number of test cases contained in the task. This is because that two tasks with an equal number of test cases may not take equal time to complete since the execution time of each test case can be varied. By using the test execution time as size measurement unit to partition a CTS test job, more equal-sized tasks in terms of execution time can be obtained. We assume that each test package (and test class) has been executed before so that its execution time has been known. This is achievable in practice since CTS tests are usually continuously re-executed to uncover any regression errors of the device implementations.

3.2.2 The scheduling of tasks

Task scheduling concerns about how to map tasks appropriately to available compute nodes in order to optimize certain performance indexes, such as completion time and resource utilization, while satisfying given constraints, such as task precedence and deadline. Task scheduling is considered important, particularly in parallel and distributed computing systems, since different task scheduling schemes can result in different performance of the system.

To observe how different task scheduling algorithms influence the efficiency of concurrent CTS testing, we have implemented several typical task scheduling algorithms, including Random, First-Come First-Served (FCFS), Longest Job First (LJF), and Shortest Job First (SJF). Ideally, a better speedup of CTS tests can be obtained if the algorithms can map the tasks to a set of devices so that the CTS testing workload can be distributed as evenly as possible among the devices. The results of these task scheduling algorithms will be described in section 5.2.

It should be noted that task scheduling problem has been studied extensively in the literature and many heuristic algorithms have been proposed to address the problem. For our study, task scheduling can be straightforward since the execution time of each task can be measured and derived in compiler time, the objective of the scheduling is simply to reduce the overall completion time of concurrent CTS testing, and there are no precedence constraints between the tasks.

Fig. 3 shows the main idea of the LJF scheduling scheme used in our approach. The SJF scheme is similar to LJF and can be inferred easily. Assumes that the CTS test job is partitioned into ten tasks and the sizes (*i.e.*, execution time) of the tasks are 20.5, 57.8, …, and 86.7 seconds, respectively. Suppose that four devices are used in the test. To map these ten tasks to four devices using LJF, the tasks are firstly sorted in descending order according to their sizes and put into a waiting queue. Then, each task is sequentially and dynamically allocated to a device whenever the device is available. The process is repeated until all of the tasks have been assigned. As shown in Fig. 3, after task scheduling and allocation, the total task completion time of each device is 86.7, 76.7, 77.9, and 78.0 seconds respectively and is approximately equivalent.

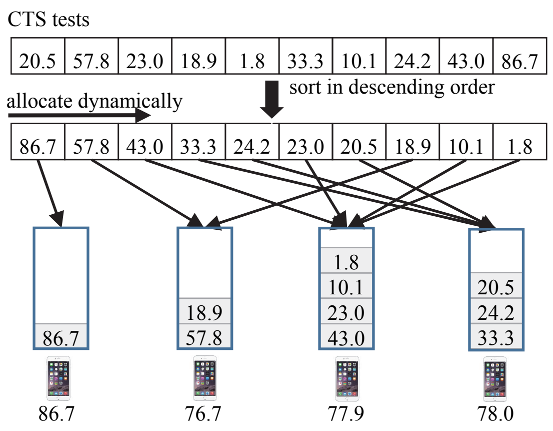
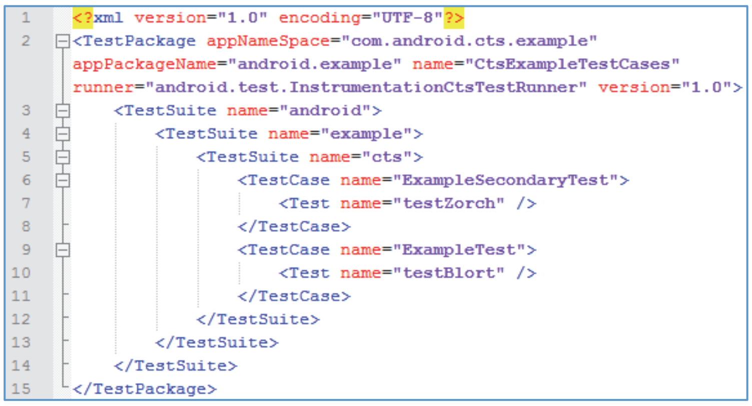


Fig. 3. An example of the LJF scheduling scheme.

3.2.3 The partitioning of tasks

In addition to the number of devices and task scheduling, the efficiency of concurrent CTS tests can also be affected by the task partitioning which can result in different number of tasks and different sized tasks. As mentioned before, the test suite of CTS are organized in different test packages hierarchically. Fig. 4 shows an example of CTS test suite organization where each test package contains a set of test suites (*i.e.*, test classes). The test classes are corresponding to the Android API classes being tested. A test class may contain other test classes and/or multiple test cases. Each test case is designed for exercising a method of a particular Android API class. Thus, based on the structure of CTS, the approach employs three types of task partitioning methods: the *package partitioning*, *class partitioning*, and *mixed partitioning* which decompose the CTS test suite into various test packages, test classes, and mixtures of test packages and classes, respectively.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<TestPackage appNameSpace="com.android.cts.example"
appPackageName="android.example" name="CtsExampleTestCases"
runner="android.test.InstrumentationCtsTestRunner" version="1.0">
    <TestSuite name="android">
        <TestSuite name="example">
            <TestSuite name="cts">
                <TestCase name="ExampleSecondaryTest">
                    <Test name="testZorch" />
                </TestCase>
                <TestCase name="ExampleTest">
                    <Test name="testBlort" />
                </TestCase>
            </TestSuite>
        </TestSuite>
    </TestSuite>
</TestPackage>
```

Fig. 4. An example of CTS test suite organization.

Specifically, in the package partitioning, each task contains a single test package. In the class partitioning, each task contains a single test class. For the mixed partitioning, it will break a large test package into several smaller test classes. This would result in a set of tasks in which a task can be either a test package or a test class. Therefore, the package partitioning can create fewer larger-grained tasks. On the other hand, the class partitioning can generate a large number of fine-grained tasks. The mixed partitioning, however, would have the number of tasks somewhere between the other two partitioning methods.

To apply the mixed partitioning method, we need to select target test packages and break them into test classes. To identify which test packages to break, we analyze the size (*i.e.*, execution time) of each package for Android CTS 4.2 test suite. Fig. 5 shows the analysis result in which the execution time percentages of major test packages are highlighted. The result indicates that only very few test packages can take much longer time to execute than others. For example, the android.core.tests.libcore.package.libcore and android.media packages take 28% and 11% of the total execution time to complete, respectively. Particularly, the top five largest packages together will take nearly 65% of the overall execution time of CTS testing. This suggests that the mixed partitioning method may just need to break a few packages for achieving a better test efficiency. Note that the gain of test efficiency could be offset by the overheads of dividing test packages, task download, communications, and merging test results if too many packages are disintegrated.

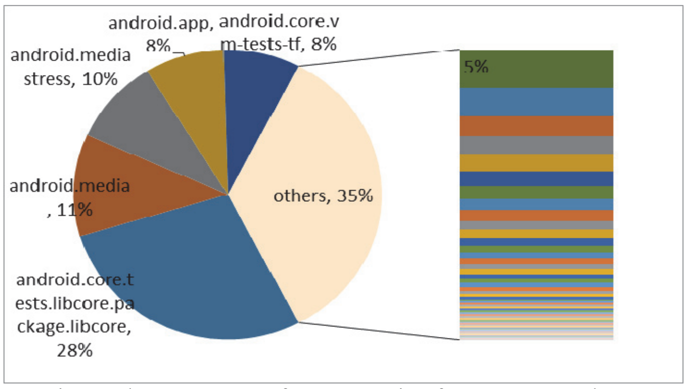


Fig. 5. The percentage of text execution for CTS test package.

4. DESIGN OF THE CTS CLOUD TESTING PLATFORM

To support the proposed approach, a cloud-based testing platform, called AndroidCTS, has been developed. The AndroidCTS is constructed based on the Eucalyptus [24], a widely used open source cloud platform. Fig. 6 shows the system architecture of the AndroidCTS. Basically, the AndroidCTS consists of two parts: the CTS Server and the CTS Station. The CTS Server and CTS Station communicate with each other using the Message Queue (Rabbit MQ [25]) and File System.

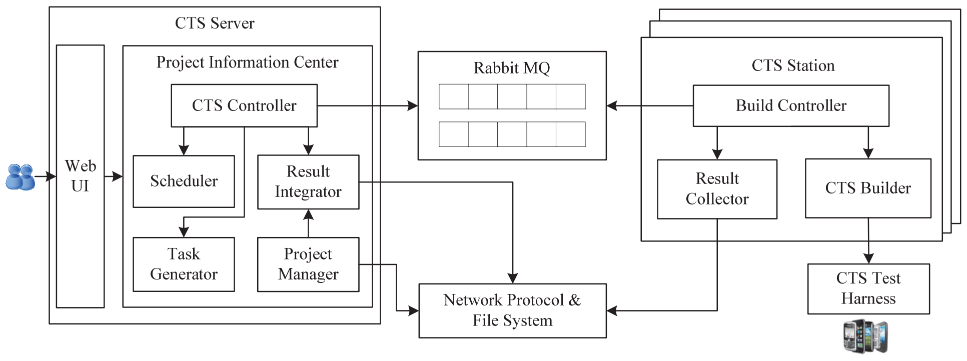
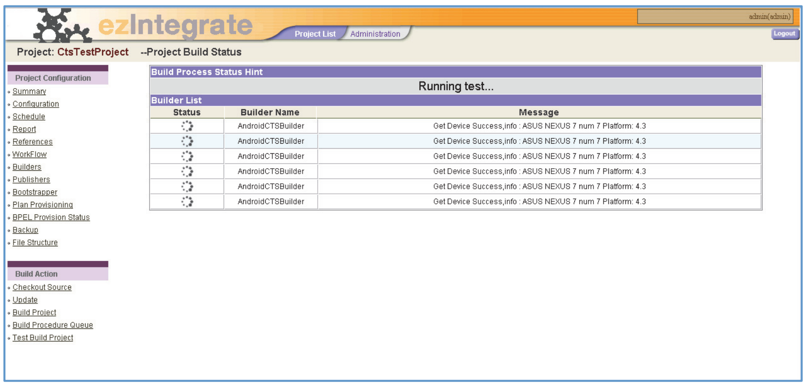


Fig. 6. The system architecture of the AndroidCTS.

The CTS Server consists of the Web UI and Project Information Center subsystems where the Web UI is responsible for providing the user interface and the Project Information Center is responsible for controlling and managing the CTS tests. Particularly, the Project Information Center is composed of the CTS Controller, Task Generator, Scheduler, Result Integrator, and Project Manager. The CTS Controller controls the concurrent CTS testing process. The Task Generator is in charge of task partitioning. The Scheduler is responsible for task scheduling. The Result Integrator is used to merge test results and generate CTS test reports. The Project Manager is in charge of managing the test projects and resources.

The CTS Station is composed of the Build Controller, CTS Builder, and Result Collector. There are multiple concurrent instances of CTS Station. Each CTS Station instance runs on a VM and connects a device emulator and Google CTS test harness. Moreover, the Build Controller is in charge of controlling the CTS test on the device. The CTS Builder is responsible for executing the task on the device. The Result Collector is used to collect and return the test result of the task back to the CTS Server.

To execute CTS testing using the AndroidCTS, users first need to configure the CTS test project, including selecting the type and the number of devices, the test packages, the task scheduling algorithm, and the task partitioning method used for the testing. Then, after users submitting the test execution request, the platform will automatically build the CTS Station instances based on the specified type and the number of the devices. Fig. 7 shows the screen snapshot of the CTS building process. Once a CTS Station instance is built successfully, the instance will retrieve and perform the test tasks automatically. The test result of each instance finally will be integrated to generate a CTS test report.



Project: CtsTestProject --Project Build Status

Build Process Status Hint

Running test...

Builder List

Status	Builder Name	Message
	AndroidCTSBuilder	Get Device Success,info : ASUS NEXUS 7 num 7 Platform: 4.3
	AndroidCTSBuilder	Get Device Success,info : ASUS NEXUS 7 num 7 Platform: 4.3
	AndroidCTSBuilder	Get Device Success,info : ASUS NEXUS 7 num 7 Platform: 4.3
	AndroidCTSBuilder	Get Device Success,info : ASUS NEXUS 7 num 7 Platform: 4.3
	AndroidCTSBuilder	Get Device Success,info : ASUS NEXUS 7 num 7 Platform: 4.3
	AndroidCTSBuilder	Get Device Success,info : ASUS NEXUS 7 num 7 Platform: 4.3

Fig. 7. A screen snapshot of the AndroidCTS.

5. PERFORMANCE EVALUATION OF THE APPROACH

To evaluate the effectiveness of the proposed concurrent CTS testing approach, the following research questions are proposed.

*RQ*1. How does the efficiency of CTS tests being improved using the proposed concurrent CTS testing approach?
*RQ*2. How does the efficiency of CTS tests being affected by task scheduling algorithms?
*RQ*3. How does the efficiency of CTS tests being affected by different partitioning methods?

To address the above questions, several experiments were conducted. The experimental environment includes a Eucalyptus cloud platform that consists of 1 cluster controller and 5 nodes. The controller and each node has a Hexa-Core Intel Xeon 2.0GHz CPU, 64GB of memory, and 600GB of storage. The operation systems of both controller and nodes are Linux Ubuntu Desktop 12.04. Each node can contain multiple VMs where each VM is m1.large type and has an Intel Xeon 2.0GHz CPU, 2048MB of memory, and 6GB of storage. The operation system of each VM is Linux Ubuntu Desktop 10.10. The device connected to each VM is Android 4.2.2 emulator with 512MB of memory. The version of CTS test suite is 4.2 that contains more than 18,000 test cases and 60 packages. For the experiments, three packages (about 240 test cases) were excluded since they can only be executed on the physical Android devices. The total execution time of these test packages is about 8 hours on a single VM.

5.1 Speedup of Different Number of Devices

To evaluate the amount of the efficiency improvement for the proposed CTS testing approach, we partition the CTS test into multiple tasks where each task contains a single test package. These tasks are scheduled and allocated to multiple VMs in FCFS order. The test results of the tasks are then merged and the execution time of VMs is recorded. Table 1 shows the longest and shortest execution time for different number of VMs. The results suggest that the overall CTS test time can be reduced as the number of VMs increases. However, the relatively large standard deviation of the execution time also indicates that there exist workload variations among the VMs and the task assignment seems unbalanced.

Table 1. The CTS test execution time for different number of devices (hh:mm:ss).

Device quantity / Execution time	1	2	4	8	16	32
The Shortest	08:02:57	04:13:28	01:42:22	00:46:25	00:12:56	00:02:41
The Longest	08:02:57	05:35:19	03:58:20	03:05:23	02:42:09	02:43:22
Std. deviation	–	00:57:53	01:03:38	00:46:12	00:37:18	00:30:35

Fig. 8 shows the speedup ratio of this experiment. The result indicates that the CTS test efficiency was improved proportionally to three times as the number of VMs increases up to 16 and then it became saturated. Thus, from the results of Table 1 and Fig. 8, the answer to *RQ*1 is that "the efficiency of CTS tests can be improved using the proposed approach. The amount of improvement can be affected significantly by the number of VMs used in the testing and the workload variations of the VMs."

5.2 Speedup of Different Task Scheduling Algorithms

To evaluate how the task scheduling methods affect the CTS test efficiency in the proposed approach, the tasks in the above experiment were scheduled with different algorithms. Table 2 shows the test completion time for the FCFS, random, SJF, and LJF scheduling algorithms under different number of VMs where the test completion time is the longest execution time required by each VM to complete its assigned tasks. The results suggest that the overall CTS test time can be influenced by using different scheduling algorithms. The effects of scheduling algorithms would decrease as the number of VMs increases.

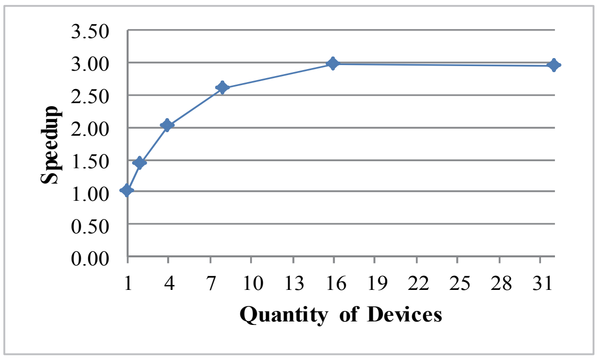


Fig. 8. The speedup for package partition under FCFS scheduling.

Table 2. CTS test completion time for different scheduling algorithms (hh:mm:ss).

Device number / Completion time	1	2	4	8	16	32
FCFS	08:02:57	05:35:19	03:58:20	03:05:23	02:42:09	02:43:22
Random	08:02:57	05:53:23	04:11:50	03:07:01	02:51:56	02:48:53
SJF	08:02:57	06:03:08	04:08:59	02:59:31	02:50:21	02:45:05
LJF	08:02:57	04:23:10	02:45:29	02:33:44	02:34:20	02:45:28

Fig. 9 shows the speedup ratios of different scheduling algorithms. The result indicates that the speedup of the LJF algorithm can quickly reach the saturation point as compared with others. Specifically, the speedup of LJF can reach 2.92 times as the number of VMs increases up to 4 while other scheduling methods may require up to 16 VMs before their speedup ratios can nearly reach the same value of LJF. The reason why LJF outperforms other scheduling methods (speed up 4 times faster) perhaps can be observed from the distributions of test execution time on individual VMs in Fig. 10.

As shown in Fig. 10, the distributions of test execution time on individual VMs are depicted when the CTS testing is performed using 2, 4, 8, and 16 VMs, respectively. From Fig. 10, we can observe that the variation of test execution time on individual VMs for LJF is smaller than that of others when the CTS testing is performed using 2, 4, or 8 VMs. Moreover, for each scheduling algorithm, the variation of test execution time on different VMs becomes closer as the number of VMs used in the CTS testing increases. Further, when the number of VMs increases up to 16, the variation of test execution time on individual VMs for all the scheduling methods as well as their speedup ratios become close. This result may suggest that the LJF scheduling method is more likely to reduce the overall workload (*i.e.*, test execution) variations among the VMs than others for the proposed approach. The reason may be because the short jobs will be executed with a low priority in LJF. Consequently, when the concurrent CTS testing is close to finish, only small sized tasks are left behind and, hence, the workload variations of these tasks could be small.

Thus, according to the results of Table 2 and Fig. 9, the answer to *RQ*2 is that "the test time of CTS and the number of VMs required to achieve maximum CTS test efficiency can be affected by the task scheduling algorithms. For the proposed approach, the LJF scheduling algorithm could quickly reach the speedup saturation point of test efficiency with less number of VMs.

Notice that in Fig. 10 we can observe that, no matter how many VMs are employed in the CTS testing, one VM persistently takes more than 2.5 hours to complete its task

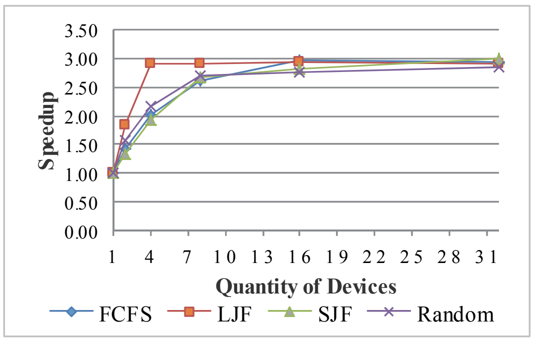
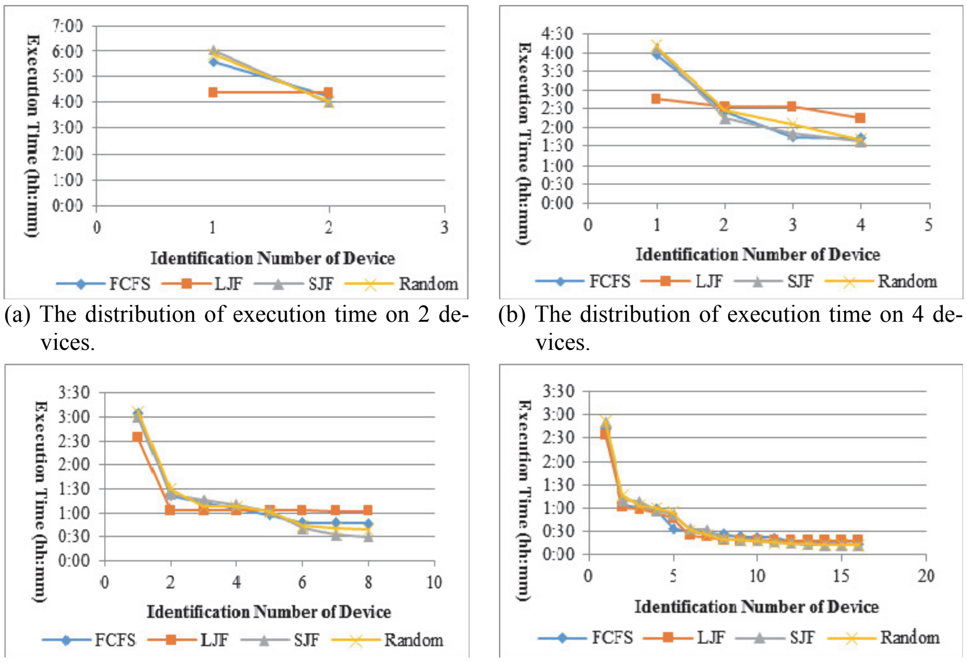


Fig. 9. The speedup for package partition under different scheduling.



(a) The distribution of execution time on 2 devices.

(b) The distribution of execution time on 4 devices.

(c) The distribution of execution time on 8 devices.

(d) The distribution of execution time on 16 devices.

Fig. 10. The distributions of test execution time on different number of devices.

while the time needed by the rest of VMs to finish their tasks can decrease to less than 1 hour as the number of VMs increases. This indicates that some tasks (*i.e.*, test packages) might take a long time to complete as compared with others. These tasks can become the bottleneck of the efficiency improvement unless they are divided into smaller ones with task partitioning methods.

5.3 Speedup of Different Task Partitioning Methods

To evaluate how the task partitioning methods affect the CTS test efficiency in the proposed approach, the CTS test suite was decomposed in different ways using the package, class, and mixed partitioning methods. As mentioned in section 3.2.3, the CTS is decomposed so that each task is a single test package for the package partitioning method or a single test class for the class partitioning method. For the mixed partitioning method, we decompose several large test packages of CTS into test classes and a test task may be either a single test package or a single test class.

Specifically, to evaluate how the mixed partitioning method affects the test efficiency, the mixed partitioning of CTS tests is further classified into Mixed-1, Mixed-2, Mixed-3, Mixed-4-1, and Mixed-4-2 groups based on which test packages are selected to disintegrate. The Mixed-1 decomposes only the top one largest test package into corresponding test classes while the Mixed-2 and Mixed-3 decompose the top two largest and top three largest test packages into test classes, respectively. Moreover, according to Fig. 5, both android.app and android.core.vm-tests-tf are the fourth largest packages and have about the same size. Thus, two additional groups Mixed-4-1 and Mixed-4-2 are presented in which the top four largest test packages are partitioned into corresponding test classes. However, the fourth largest package in Mixed-4-1 and Mixed-4-2 are different. Table 3 shows the number of test tasks generated by each partitioning method and the target packages that are selected to break for the mixed partitioning method.

Table 3. The number of test tasks for partitioning methods.

Partitioning method	Number of packages	Number of test classes	Total number of test tasks	The packages being disintegrated
Package	57	0	57	
Class	0	2029	2029	
Mixed-1	56	457	513	android.core.tests.libcore.package.libcore
Mixed-2	55	457+38	550	android.core.tests.libcore.package.libcore, android.media
Mixed-3	54	457+38+13	562	android.core.tests.libcore.package.libcore, android.media, android.mediastress
Mixed-4-1	53	457+38+13+ 47	608	android.core.tests.libcore.package.libcore, android.media, android.mediastress, an-droid.app
Mixed-4-2	53	457+ 38+13 +225	786	android.core.tests.libcore.package.libcore, android.media, android.mediastress, an-droid.core.vm-tests-tf

Fig. 11 shows the speedup ratios of different partitioning methods combined with FCFS task scheduling. The result indicates that the mixed partitioning groups commonly have a better test efficiency than the package and the class partitioning methods. In particular, the speedup of Mixed-3 outperforms others as the number of VMs increases. The reason that the mixed partitioning method could result in a better improvement of CTS test efficiency may be because the large test packages usually dominate the execution time of CTS testing and can be the bottleneck of the efficiency improvement as indicated in Fig. 10. Thus, by breaking a large test package into several smaller test classes, the workload variation among the tasks can be more likely reduced and, hence, the test efficiency can be improved.

However, the result of Fig. 11 also suggests that the mixed task partitioning cannot be carried out too far. As shown by the results of Mixed-4-1and Mixed-4-2, if we partition too many packages into classes, the overheads imposed by the task partitioning, such as task download, communications between server and devices, and test results integration, could offset the improvement of test efficiency.

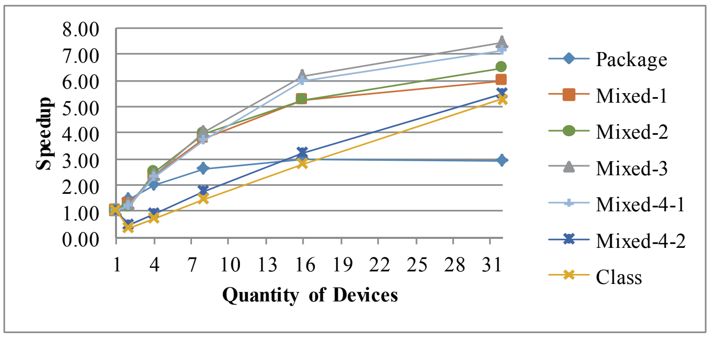


Fig. 11. The speedup of different partitioning methods under FCFS.

Therefore, based on the result of Fig. 11, the answer to *RQ*3 is that "the CTS test efficiency can be affected by the task partitioning methods. For the proposed approach, the mixed task partitioning method generally can result in a better test efficiency than others." Especially, a much better test efficiency can be achieved if the top three largest packages, *i.e.*, android.core.tests.libcore.package.libcore, android.media, and android.mediastress, are disintegrated into corresponding classes.

Moreover, the above separate experimental results indicate that the CTS test efficiency can be better improved if using a large number of devices, the LJF scheduling, or the mixed task partition. To take into account all these observations together, an additional experiment was conducted. Fig. 12 shows the speedup ratios of CTS tests under different number of devices with the LJF and FCFS scheduling algorithms and the mixed partitioning methods. The result indicates that the speedup of Mixed-3 combined with LJF scheduling outperforms others as the number of VMs increases. This result is consistent with the findings of previous experiments. Further, the result of Fig. 12 also indicates that, when combined with different mixed partition groups, LJF has better speedups than FCFS as the number of VMs increases, except for Mixed-4-2. This result is in accordance with the finding in Fig. 9 where package partition was used. Note that the speedup values of FCFS combined with mixed partitioning groups in Fig. 11 and those in Fig. 12 are a little different because these two experiments were conducted using different Android emulators. The result of Fig. 11 was obtained using Android SDK emulator 4.2.2 while the result of Fig. 12 was obtained using VirtualBox Android emulator 4.2.2. Although the speedup values of FCFS with mixed partition in Fig. 11 and their counterparts in Fig. 12 are slightly different, they have similar tendency as the number of VMs increases, and in both cases Mixed-3 has the highest speedup. This outcome also implies that the speedup of concurrent CTS tests can be somewhat affected by the emulators or devices used in the experiments.

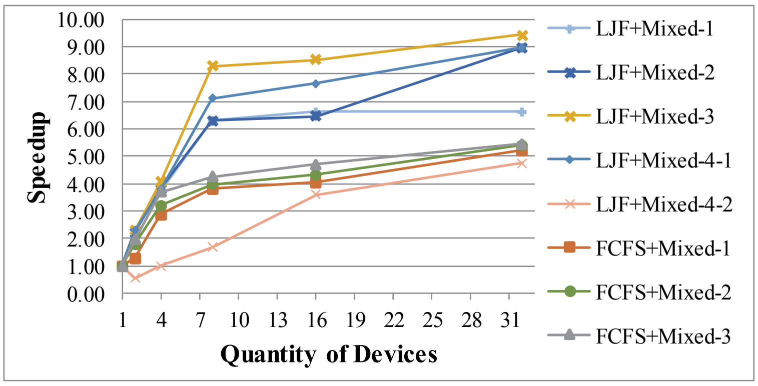


Fig. 12. The speedup of mixed partitioning methods with FCFS and LJF.

5.4 Threats to Validity

The threats to internal validity can come from the implementation of AndroidCTS and how the experiments were carried out. Although AndroidCTS has been carefully tested through automated unit testing and acceptance testing, it might still have some defects that could possibly compromise the validity of the results. Moreover, the experiments were conducted using Android emulators instead of physical devices. The results of test execution time might be different if different Android emulators or physical devices are used for running the CTS tests. Further, another possible threat is that a random scheduling algorithm is used in the experiments. To reduce the threat caused by randomization, experiments involving random task scheduling should be repeated several times to take the randomness into account.

The threats to external validity lie in the subjects (*i.e.*, CTS test suite) selected for the experiments. The experiments had dropped out three CTS packages (android.holo, android.opengl, and android.renderscript) since they can only be executed on the physical devices. The execution time might be slightly affected because of leaving out the packages. However, the sizes of these three packages take up only a small portion of the entire CTS suite (240 out of 18,000 test cases) and, hence, the results of partitioning methods and parallel execution might not be affected much. Moreover, the subject of this empirical study is CTS 4.2 test suite. Nevertheless, there are several CTS versions currently available since Android operating system has evolved quickly in recent years. The sizes and structures of different CTS versions might not be the same as CTS 4.2. Thus, the experimental results might not extend to other CTS versions.

5.5 Discussions and Suggestions

From the aforementioned studies and experimental results, the followings are some observations.

1. The notion to improve CTS test efficiency by dividing a single test job to multiple tasks and executing the tasks concurrently on different Android devices to shorten the overall test time can be feasible and easy to implement if a supporting cloud testing infrastructure, such as AndroidCTS, has been provided.
2. The speedup of test efficiency can be improved if the number of VMs (or devices) used in the CTS testing increases. However, the speedup of the improvement can become saturated if too many VMs are used. Further, the more VMs the testing uses, the higher expense the testing costs. The number of VMs used for the concurrent testing might be dependent of the tradeoff between the test efficiency and test cost. Other concerns, such as the deadline of testing, might be considered as well.
3. The LJF scheduling algorithm with dynamic task allocation can reach the saturation point of efficiency improvement much faster than others. This is because LJF could reduce the workload variations of the tasks at the end of testing since, at that time, only short jobs are left behind, and the workload variations of short jobs could be small. If the available number of VMs (or devices) is less than 8, the LJF scheduling method can perform better than others.
4. The distribution of test cases in CTS is not well-balanced. The top five largest packages take almost 65% of the total execution time to complete the testing. The efficiency of concurrent CTS testing is largely dominated by these packages.
5. The size and the number of the tasks created by the partitioning methods are critical. The partitioning methods that generate a small number of large-size or a large number of small-size tasks may not lead to a better speedup of test efficiency. The mixed partitioning method, however, may result in a better improvement since the workload of the tasks can be distributed more evenly while the overheads imposed by the partitioning can be reduced without compromising the gain of efficiency.
6. The increment of test efficiency obtained from the partitioning methods can be higher than that obtained from the scheduling algorithms. This suggests that task partitioning plays an important role in the concurrent CTS testing approach. For Android CTS 4.2, the test efficiency can be much improved if the top three largest packages are disintegrated to corresponding test classes.
7. According to the experimental results, a good practice for a test engineer to perform CTS tests in parallel could be that at least 16 Android devices, LJF scheduling, and the Mixed-3 partition are deployed so as to gain an optimal test efficiency. If, however, the number of available devices has a limitation, the practice that uses at least 4 devices in combined with LJF scheduling and Mixed-3 partition is suggested. Moreover, if there exists a concern to disintegrate the CTS packages, at least 4 devices, LJF scheduling, and the package partition should be deployed in order to obtain a satisfactory CTS test efficiency improvement, say twice and above.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a concurrent approach to improve the efficiency of CTS tests for shortening the time-to-market of Android devices. The main idea of the approach is to partition the CTS tests into multiple tasks that are executed on different devices concurrently. In particular, the proposed approach also considers various task scheduling and partitioning methods. A cloud-based testing platform called AndroidCTS that supports the approach is developed. Several experiments were conducted and the results indicate that the CTS test efficiency can be improved depending on the number of VMs, the task scheduling algorithm, and the task partitioning method used in the proposed approach. The observations of the study are also described.

In particular, the speedup of CTS testing can be improved proportionally as the number of VMs (or devices) increases. However, the improvement will become saturated when the number of VMs (or devices) is more than 16. Further, the LJF scheduling method can reach the speedup saturation point 4 times faster than other traditional scheduling methods. Moreover, the CTS test efficiency can be dominated by only a few large packages. By breaking the top three largest packages into corresponding classes, the speedup of test efficiency can be much improved.

In the future, we plan to extend the proposed approach to support more task scheduling algorithms as well as more task partitioning methods. Further, we plan to conduct the empirical study of the proposed approach using physical Android devices instead of emulators so that all CTS packages can be executed and the experimental results can be observed in a close to real-world environment. Moreover, we also plan to consider the cost-benefit of the proposed approach while taking into account the deadline of CTS testing.

ACKNOWLEDGMENT

This work was supported in part by the Ministry of Science and Technology, Taiwan, under the grant No. MOST 105-2221-E-027-086, which is gratefully acknowledged.

REFERENCES

1. Stastics, Installed base of smartphones by operating system from 2015 to 2016 (in million units), https://www.statista.com/statistics/385001/smartphone-worldwide-installed-base-operating-systems/.
2. Gartner, Gartner says worldwide smartphone sales grew 3.9 percent in first quarter of 2016, http://www.gartner.com/newsroom/id/3323017.
3. Expanded Ramblings, 108 amazing Android statistics, http://expandedramblings.com /index.php/android-statistics/. [Accessed: Jan. 2017]
4. Android Fragmentation, http://en.wikipedia.org/wiki/Fragmentation_%28programming%29.
5. L. Wei, Y. Liu, and S.-C. Cheung, "Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps," in *Proceedings of the 31st*

IEEE/ACM International Conference on Automated Software Engineering, 2016, pp. 226-237.
6. Android 7.1 Compatibility Definition Document, http://source.android.com/compatibility/android-cdd.html.
7. Compatibility Test Suite, https://source.android.com/compatibility/cts/.
8. Google Mobile Services, https://www.android.com/gms/.
9. H. K. Ham and Y. B. Park, "Mobile application compatibility test system design for Android fragmentation," *Software Engineering*, *Business Continuity*, *and Education*, in the series *Computer and Information Science*, Vol. 257, 2011, pp. 314-320.
10. H. K. Ham and Y. B. Park, "Designing knowledge base mobile application compatibility test system for Android fragmentation," *International Journal of Software Engineering and Its Applications*, Vol. 8, 2014, pp. 303-314.
11. J. Kaasila, D. Ferreira, V. Kostakos, and T. Ojala, "Testdroid: Automated remote UI testing on Android," in *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia*, 2012.
12. J.-F. Huang, "AppACTS: Mobile app automated compatibility testing service," in *Proceedings of the 2nd IEEE International Conference on Mobile Cloud Computing*, *Service*, *and Engineering*, 2014, pp. 85-90.
13. T. Zhang, J. Gao, J. Cheng, and T. Uehara, "Compatibility testing service for mobile applications," in *Proceedings of IEEE Symposium on Service-Oriented System Engineering*, 2015, pp. 179-186.
14. C.-H. Liu, S.-L. Chen, and W.-K. Chen, "Improving resource utilization of a cloud-based testing platform for Android applications," in *Proceedings of IEEE International Conference on Mobile Services*, 2015, pp. 202-208.
15. Openstack, https://www.openstack.org/.
16. M. E. Joorabchi, M. Ali, and A. Mesbah, "Detecting inconsistencies in multi-platform mobile apps," in *Proceedings of the 26th IEEE International Symposium on Software Reliability Engineering*, 2015, pp. 450-460.
17. T. Liu, C. Cao, J. Chen, Z. Lu, and X. Ma, "Apsaras: Efficient allocation of physical devices for Android testing," in *Proceedings of the 40th IEEE Annual Computer Software and Applications Conference*, 2016, pp. 269-274.
18. Testdroid Cloud, http://testdroid.com/.
19. CloudMonkey, https://www.cloudmonkeymobile.com/.
20. Xamarin Test Cloud, http://xamarin.com/.
21. Appurify, http://appurify.com/.
22. Firebase Test Lab for Android, https://developers.google.com/cloud-test-lab/.
23. J. Gao, X. Bai, W. T. Tsai, and T. Uehara, "SaaS testing on clouds – issues, challenges, and needs," in *Proceedings of the 7th IEEE International Symposium on Service-Oriented System Engineering*, 2013, pp. 409-415.
24. Eucalyptus, http://www.eucalyptus.com/.
25. Rabbit MQ, http://www.rabbitmq.com/.

Chien-Hung Liu (劉建宏) received his Ph.D. degree in Computer Science and Engineering from the University of Texas at Arlington in 2002. He is currently an Assistant Professor of Computer Science and Information Engineering Department at National Taipei University of Technology, Taiwan. His research interests include software testing, software engineering, and cloud computing.

Woei-Kae Chen (陳偉凱) received M.S. and Ph.D. degrees in Computer Engineering from North Carolina State University in 1988 and 1991, respectively. He is currently a Professor at the Department of Computer Science and Information Engineering and the director of Software Development Research Center of the National Taipei University of Technology, Taiwan. His research interests include software testing, software engineering, visual programming, and cloud computing.

Shu-Ling Chen (陳淑玲) received her Ph.D. degree in Industrial and Manufacturing Systems Engineering from the University of Texas at Arlington in 2002. She is currently an Assistant Professor of Industrial Management and Information Department at Southern Taiwan University of Science and Technology. Her research interests include information management systems, e-business, and software testing.