Testing as a Service over Cloud

Lian Yu, Wei-Tek Tsai[1], Xiangji Chen, Linqing Liu, Yan Zhao, Liangjie Tang, Wei Zhao[2]
School of Software and Microelectronics, Peking University, P.R. China
[1]School of Computing, Informatics, and Decision Systems Engineering Department of Computer Science and Engineering, Arizona State University, USA.
[1]Deparment of Computer Science and Technology, Tsinghua University, Beijing, PRC
[2]China Research Center, IBM, Beijing, 100094, PRC

Abstract

Testing-as-a-service (TaaS) is a new model to provide testing capabilities to end users. Users save the cost of complicated maintenance and upgrade effort, and service providers can upgrade their services without impact on the end-users. Due to uneven volumes of concurrent requests, it is important to address the elasticity of TaaS platform in a cloud environment. Scheduling and dispatching algorithms are developed to improve the utilization of computing resources. We develop a prototype of TaaS over cloud, and evaluate the scalability of the platform by increasing the test task load; analyze the distribution of computing time on test task scheduling and test task processing over the cloud; and examine the performance of proposed algorithms by comparing others.

Keywords: TaaS (Testing as a Service), SaaS (Software as a Service), cloud computing, ontology, scheduling and dispatching.

1 Introduction

Software testing is expensive and labor intensive. It requires 40%-70% of software development costs, and even more for mission-critical applications. One of the goals of software testing research is to automate as much as possible, thereby significantly reducing its cost, minimizing human error, and making regression testing easier. Conventionally, vendors of testing tools deliver a collection of testing products to their customers, who need to install, learn and maintain products.

TaaS (Testing-as-a-service) provides users with testing services such as auto-generation of test cases and test auto-execution, and test result evaluation [1][2]. Test tasks and TaaS capability can be modeled using ontology techniques, and they can be and intelligently matched based on the shared ontology model. We explored TaaS feasibility by providing unit testing services over the Web, including basis path testing, condition testing, and data-flow testing [2].

Among many TaaS issues, this paper addressed the following four issues:

1) Clustering the tenant requests according to their similarity in terms of hardware, OSs, supporting kits and testing services, such that the configuration time and setup time are minimized.

2) Scheduling the clustered tasks such that the test requirements and deadline can be met.

3) Monitoring all testing resources and task status assigned.

4) Managing cloud resources including creating, maintaining and migrating processes and processors at runtime according to the scheduling decision.

The paper addresses the above four issues as follows:

- *Cluster tenant requests*: Test tenants may have a variety of test tasks submitted to the TaaS platform. Preparing separate testing environments for different tenants and different tasks will incur significant cost and time. This paper clusters tenant requests using data mining techniques using a test task ontology model.
- *Schedule the clustered tasks*: Different tenant requests have different service levels of agreements, such as deadline constraints. In addition, due to the uncertainty of testing processes, for example, transmission error in the cloud and system crash, the slack times of test tasks will change, and priority of the test tasks will also dynamically change. On the other hand, TaaS cloud host an array of testing services on a pool of virtual machines (VMs) with different configurations. Based on the test tasks and the status of TaaS cloud, this paper will develop a heuristic scheduling algorithm to dynamically allocate test tasks to appropriate (virtual) machines for processing.
- *Monitor testing resources and task status*: The TaaS platform has monitors that can supply status information such as the length of the task queue for the scheduler to select processors or VMs for test execution.
- *Manage cloud processes, processors and VM dynamically*: The feature enables elasticity and flexibility. If there is no appropriate VM to handle a test task, or existing VMs are fully occupied, the scheduler will issue a request to create new a VM to process the new task. If a VM has accomplished its tasks, monitor will send a command to release the VM

for other tasks. On the other hand, if the utility of the VMs on a host machine is low, a VM may be migrated to other host machine.

This paper is organized as follows. Section 2 illustrates the architecture of a TaaS platform. Section 3 constructs an ontology testing model and presents matching rules. Section 4 describes test-task scheduling and fault tolerance technology. Section 5 illustrates testing service composition. Section 6 presents a TaaS cloud infrastructure. Section 7 illustrates experiments to evaluate the proposed TaaS platform. Section 8 describes the related work. Finally, Section 9 concludes this paper.

2 Cloud TaaS Architecture

Figure 1 shows a cloud TaaS architecture with five layers: 1) test tenant and test service contributor layer, 2) test task management layer, 3) test resource management layer, 4) test layer (and it has three components: testing service composition, testing service pool, and testing task reduce), and 5) test database layer.

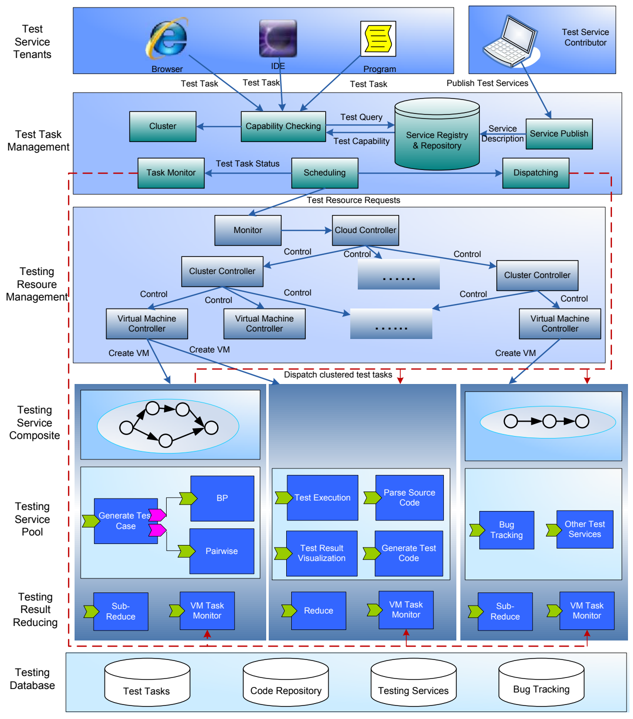

Figure 1: Architecture of TaaS over Cloud

1) Test service tenant and contributor layer: This layer has two parts, supporting test service tenants and contributors to interact with TaaS. The tenants of TaaS can use a portal, or an integrated development environment (IDE) to access testing services via the Web, even employ a programmatic approach to interact with the TaaS platform. Test service contributors publish and deploy their implemented services to the TaaS platform.

2) Test task management layer: It acts as testing service bus (TSB), which stays between end users and a variety of testing services, and consists of five core components: a) testing capability checking, b) test task clustering, scheduling and dispatching, c) test task monitor, d) service registry and repository, and e) service publish. The scheduled test tasks are dispatched to corresponding VMs. Test task monitors collect task status from each VM.

3) Testing resource management layer: It plays the role of the cloud infrastructure. It monitors physical machines and VMs beneath, and allocates testing resources according to the requests of the task controller. This layer has three sub-layers: a) the preprocessing, b) the resource management level, and c) the computing node level (the VM deployed with specific applications). The cloud infrastructure also provides an interface to various cloud services including VM management, user/group management, accounting, monitoring and provisioning.

4) Test layer: It consists of three sub-layers: a) service composition, b) service pool, and c) test-reduce. Service composition specifies a workflow and related services to orchestrate an execution sequence to complete the testing task. Service pool hosts an array of test services for different types of testing, e.g., unit-testing tasks, including test case generation service, test execution service, and result presentation service. Test-reduce aggregates test results, which belong to a tenant from different VMs, and reports the results to corresponding test tenants. Test tasks on TaaS platform are running on VMs.

5) Testing database layer: This stores test tasks of tenants, targets-under-test, service images, and bug tracking results.

3 TaaS Tasks

This section describes the TaaS processes; utilizes ontology techniques to model testing related concepts and their relations, and SWRL (Semantic Web Rules language) to model the rules matching tenants' test tasks with TaaS test capabilities; clusters test tasks according to their similarity; and schedules clustered tasks to proper testing resources based on test-resource and test-task-status monitors.

3.1 Test Task Management Processes

Task management performs test task clustering and scheduling. This paper uses data mining clustering technique to testing tasks requested by different tenants, and explores scheduling algorithms to allocate testing resources to test tasks of tenants.

1) Tenant's test requests are matched with test capabilities of the TaaS platform to decide if the platform can fulfill the requests. Then clustering

service categories test task requests from different tenants into test task clusters according to their similarity based on a task ontology model.

2) Scheduling phase sorts test activities in each cluster according to their priorities, and assigns tasks to VMs based on the workloads monitored and testing resource status. Test tasks in a cluster may be split into several sub-clusters and scheduled to different VMs to meet their deadlines.
3) Dispatching phase loads task activities to VMs according to the scheduling results, and performs concurrently test case generation and/or test execution on VMs.
4) Aggregation phase shuffles the testing results, which may be carried out on different VMs, in terms of tenants such that test tenants get the test result reports as a whole. It also acts upon test result analysis and generates test reports back to each tenant.

3.2 Modeling Test Tasks and Capabilities

This section uses ontology techniques to model testing-related concepts and their relations, and uses SWRL to model the rules matching tenants' test tasks with TaaS test capabilities.

Test tasks of tenants are modeled as an OWL class, *TestTask*, which contains OWL sub-classes of *TargetUnderTest*, *TestResource*, *TaskContraints*, and *TestType*, and has a *TestActivity* and *TestSchedule* as shown in Figure 2.

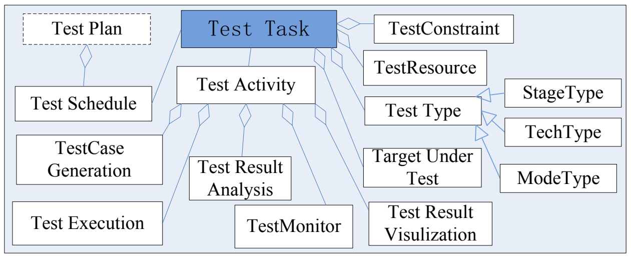

Figure 2: OWL Model of Test Tasks

TestResource shows the requirements of hardware and software to enable a test task, for example, a computer to generate test cases and dependent software package. *TestConstraint* specifies test constrains to meet, such as path coverage or condition coverage. *TestType* can be categorized in terms of development stages (e.g., unit testing, integration testing or system testing), techniques (white-box or black-box or grey-box) and test mode (e.g. static test or dynamic test). *TestActivity* includes *TestCaseGeneration*, *TestExecution*, *TestMonitor*, *TestResutlAnalysis* and *TestVisualization*. Each OWL class has associated properties to indicate its attributes and specify the connections with other OWL classes.

TargetUnerTest can select *TestActivity* services from TaaS. According to *TestActivity* selected, *TestEnvironment* will be chosen to execute *TestActivity*. During *TestCaseGeneration* stage, it will use *TestType* to generate test cases, and these test cases should satisfy *TestConstraint*.

Test capability of TaaS is modeled as an OWL class, *TestCapability*, which consists of several OWL classes as shown in Figure 3.

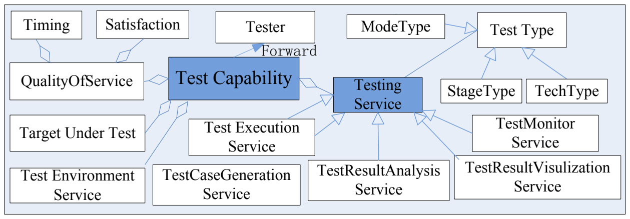

Figure 3: OWL Model of TestCapability

OWL class *TestingService* represents the testing-related services on TaaS platform, and has several subclasses to match each testing activity in *TestTask*. OWL class *QualityOfService* indicates the competence to meet test schedule and the degree of tenant satisfaction. If there is no capability that matches a test task from a tenant, the request will be forwarded to *Tester*, which can be a human being or a team and takes proper measure to handle the special situation.

This section defines nine rules using SWRL to match test tasks of tenants with TaaS capabilities. Figure 6 shows the rule of matching *TestCaseGenerationService* with *TestTask*. The SWRL rule reads as "If the test types of the test task are the same as one of *TestCaseGeneration* Services, and features of the *TestCaseGeneration* Service satisfy the task requirement, and then match the test task with the *TestCaseGeneration* Service".

```
......
TestCaseGeneration_Service(?gen_service) ∧
swrlb:stringEqualIgnoreCase(?tt_techType, ?tc_techType)∧
swrlb:stringEqualIgnoreCase(?tt_stageName, ?tc_stageName)∧
swrlb:stringEqualIgnoreCase(?tt_modeType , ?tc_modeType) ∧
swrlb:stringEqualIgnoreCase(?tt_Coverage,?tc_Coverage) ∧
swrlb:stringEqualIgnoreCase(?tt_loop, ?tc_loop) ∧
swrlb:stringEqualIgnoreCase(?tt_strategy, ?tc_strategy) ∧
swrlb:stringEqualIgnoreCase(?tt_paraInput, ?tc_paraInput)
 → matchForTestCaseGenerationService(?test_task, ?gen_service)
```

Figure 4: Match TestCaseGenerationService

3.3 Clustering Test Tasks

Before carrying out a test task, *TargetUnderTest* will be analyzed to get the test environment. The sub-classes and individuals of *TargetUnderTest* are defined as shown Table 1.

Table 1. Sub-classes and Individuals of *TargetUnderTest*

	Sub-class	Individual
Target Under Test	File Type	Source file, Binary file
	Target Type	Websites,Win32 App
	Language Type	C++, java, c#

According to the *FileType*, one can infer that whether it needs to compile test target. Through *TargetType*, one can decide whether a web server is needed. By *LanguageType*, one can specify a compiler tool.

After analyzing *TargetUnderTest* and *TestActivity*, one will get *TestEnvironment*, which contains hardware and software environments to execute test activities as shown in Table 2.

Table 2. Sub-classes and Individuals of Hardware/Software

Test Environment	Hardware	Sub-class	
		CPU	
		Memory	
		Hard disk	
		Switch	
		Printer	
		Network card	
	Software	Sub-class	Individual
		Operation System	Windows, Unix
		Web Server	Tomcat, IIS
		Database	SQL Server, MySQL, Oracle
		Compiling Tool	gcc, NET, jdk
		Testing Tool	Test management, Static analysis, Basis Path Test

Since test task requests of tenants associate with *TestEnvironment*, one clusters tenant requests according to *TestEnvironment*. Requests with similar *TestEnvironment* can be executed in one machine, reducing the cost of setting up test environments.

In the proposed cloud infrastructure, the system maintains VM images. These images are associated with particular test environments deployed with testing services. If a new test environment is needed, the system copies the associated image in VM image database to a certain VM or a physical machine and starts these machines.

To assign tasks to the cloud infrastructure, this section proposes a quick cluster algorithm based on K-Medoid [11], based on VM images in the cloud. It clusters tenant requests according to the threshold (*maxThreshold*), to specify the max distance between two requests in one cluster, and can be used for a large number of cluster tasks.

First one sets each image in the VM image database as a medoid, and then one assigns each task to the medoid that has the minimum distance with this task. If the minimum distance is larger than the threshold set, this task's environment will be added as a new medoid.

After the first cluster, medoids have been selected. Then one re-clusters the non-medoid tasks and set each non-medoid task to the closest medoid.

The detailed process is shown as follows:

```
1. Set all test environment in cloud as the medoids
2. For each request $i$ in request list $L$
       Calculate the minimal distance $b$ between $i$ and each medoids.
       If $d$ <= maxThreshold
           Associate $i$ with medoid of the minimal distance
       Else
           Add a new medoid $i$
3. For each non-medoid request $j$
       Associate $j$ to the closest medoid
```

After clustering, one gets a set of medoids. If the medoid is the image in the cloud infrastructure, one can dynamically create new VMs according to this image to handle all the tasks associated with this medoid. If not, one forwards it and the associated tasks to *Tester*, and this means the cloud infrastructure cannot supply these test resources.

This algorithm is designed to deal with large number of tasks, because its complexity is O(n). However, the threshold set directly affects the cluster number.

3.4 Scheduling Test Tasks

The goal of scheduling is to find an assignment function that achieves near-optimal solutions. This paper considers waiting time and value of each task as selection criteria, i.e., the tasks having high waiting time, low slack time, and value will be scheduled first.

- Task waiting time: the duration between current time and the time when a task was submitted.
- Task slack time: the duration between current time and its deadline.
- Task value: the value of a task is decided by the creator of this task. e.g., one can divide all tenants into five levels of priorities, and tasks created by different level tenants have different values.

Intuitively, tasks having a low waiting time and value have less opportunity to be executed earlier.

Given a set of tasks *S*, there is list of tasks *T* for a cluster, $T = \{t_1, t_2, \ldots, t_n\}$, a set of machines *M* for this cluster, $M = \{m_1, m_2, \ldots, m_n\}$, the solution can be found in steps below. Task waiting list *W* stores all the latest added tasks, and then it will be dumped to *S*.

1) Cluster all the tasks in *S*.
2) Get the valid machines for each cluster.
3) If *T* is not empty, compute the priority of all tasks and then sort them in decreasing order. We assign the tasks to machines iteratively. The task sorting algorithm is concerned mainly with the slack time, waiting time and the task values that always are decided by the creator of this task.

. So a task has a deadline, its priority [3] can be calculated by the following formula (1).

$$p = [\gamma \times (i - 1 - \mu) + 2 \times k - 2] \times (i + \mu) \div 2 + i \quad (1)$$

If a task does not have a deadline, its priority can be calculated by the following formula (2).

$$p = [\gamma \times (Max - j - 1 - \mu) + 2 \times k - 2)] \times (i + \mu) \div 2 + i \quad (2)$$

Where μ is given by (3).

$$\mu = (k - 2) / \gamma \quad (3)$$

The term *i* is the slack of a task, *j* is the waiting time of a task, *Max* is the maximum value of the waiting time interval, and k is its value. γ is the imported parameter to adjust the weight between slack time/waiting time and task value.

4) If machine set is empty which means platform does not have available machines with the required environment for this task cluster, notify the monitor to create a VM for this cluster.
5) Consult the cloud controller to get the resource utilization of each available VM associated with cluster medoid's test environment. Score all the valid machines taking the following items into consideration: length of the task waiting queue; fairness; and high CPU and Memory utilization. If all the related machines of this cluster are overloaded, it will notify the cloud controller to create more machines with the same test environment. If the medoid of this cluster is associated currently with none of test environments in the TaaS platform, it means VMs with this cluster's test environment are not ready, and the scheduler will notify the cloud controller to set up new test environments in the TaaS cloud platform for this cluster.
6) The machine m_j that owns highest score is chosen as the candidate machine to assign task t_i.
7) Dispatch task t_i to machine m_j. Then remove t_i from task set *T*.
8) Monitor in TaaS watches in real-time all the tasks and machines status,
9) If any fault detected, related tasks will be re-added into task set *T* for re-scheduling.
10) Whenever a task has been dispatched, check the task waiting list *W*. If it is not empty, add these tasks to task set, scheduling will be halted and jump to step 1. In this step we can calculate the task priority dynamically because whenever tenants create new tasks, left tasks' priority will be re-calculated and these tasks will be re-sorted and scheduler.

Generally speaking, this process is repeated until all the tasks are scheduled for processing on the machines.

3.5 Handling Fault Tolerance

When dealing with multiprocessor systems, fault-tolerance is important because components may fail [4]. This paper proposes a fault-tolerant scheduling scheme cloud computing. In general, optimal fault-tolerant scheduling of tasks is NP-complete [4], and thus different heuristics have been proposed to schedule real-time tasks to maximize certain criteria. The proposed algorithm is based on the principles of primary/backup tasks [4], and it can tolerate single faults in any processor.

As the TaaS cloud platform monitors all machines and assigned tasks, if any fault detected in a task when it is executed on a machine, this task will be put back to *TaskList*, which contains all the to-be assigned tasks. Similarly, if any fault detected in a machine, all the tasks executed on this machine will also be put back to *TaskList*. These tasks in *TaskList* are named the backup of their primary tasks. If a machine fails many times, it will be marked as "failed", so that later tasks will not be assigned to this machine until it has been repaired and marked with "ready". The pseudo-code of the algorithm is shown as follows:

```
Monitor running machines and tasks
If a fault detected in a machine
    Add all running tasks in this machine to tasks waiting
    list for re-scheduling
    Mark these tasks' status from "primary" to "backup"
    If fault detected in this machine many times
        Mark machine's status from "ready" to "repair"
        so that later tasks will not be dispatched to this
        machine
Else a fault detected in a task
    Add this task to task waiting list for re-scheduling
    Mark this task's status from "primary" to "backup"
```

4 Building TaaS Cloud

TaaS cloud infrastructure is built to auto-deploy testing services on VMs, provide on-demand computing resource to testing services and manage VM instances to make most efficient use of the resources. It is designed from the ground up to be easy to install and as non-intrusive as possible. The cloud infrastructure is composes of four components, cloud controller, cluster controller, VM controller and monitor.

4.1 Cloud Controller

The cloud controller is the entry point to the cloud for administrators, developers, project managers, and end users. The cloud is divided to more than one cluster according to physical location or logical division, and each cluster has a cluster controller. The cloud controller manages all the cluster controllers. It has four components: 1) query, 2) contract manager, 3) auto-provisioning manager, and 4) cluster manager. It is responsible for querying the cluster controllers for information about usage of resources, making high-level scheduling decisions, such as adding physical machines and creating new VM instances on physical machines according to resources usage of clusters, and implementing them by making requests to the cluster controllers. It also interfaces to the TaaS manager.

4.2 Cluster Controller

The cluster controller has three components: a profiling engine, a load detector, and a migration & reallocation manager. The dynamic provisioning process is described as follows.

1) The VM controllers gather the usage statistics of CPU, memory and storage from each physical machine *i* (*i*=1…n) and periodically relay these statistics to the cluster controller.

2) The profiling engine uses the statistics collected from the VM controllers to construct resource usage profiles during a period of time for each VM and aggregate profiles for each physical machine.
3) The load detector continuously monitors these usage profiles to detect hotspots and cold-spots. Informally, a hotspot is created when the aggregated usage of any resource (CPU, memory and storage) exceeds a threshold [5], a cold-spot occurs when the usage is at a low level.
4) The migration & re-allocation manager realizes hotspot and cold-spot mitigation via dynamically re-allocating resources and migrating VMs, and sends allocation or migration commands to VM controllers, which complete operations correspondingly.

4.3 Virtual Machine Controller

The VM controller running on a physical machine is responsible for collecting resource usage status in the physical machine on which several Xen VMs may be running. The VM controller is also responsible for executing remote management command sent from the cluster controller such as creating a VM, starting up or

With Xen virtualization technique, a physical host can run several VMs at the same time. These VMs can parallel process requests, which can maximize the resource usage and minimize the energy consumption. To monitor these VMs, the system deploys a monitor agent into each of them. This agent is responsible for monitoring the status of application servers on the VM and status of the services deployed on these servers.

4.4 VM Agents and Monitors on TaaS Cloud

Monitoring is a critical module for a robust and large-scale system. The information the monitors collect is an important input for testing task scheduling. The monitor subsystem supervises the TaaS Cloud on five different levels:

- Platform level: This monitors an OS, regardless if it runs on a VM or a physical machine.
- Server level: This monitors a Web Server (such as Tomcat) or a database is called a "server".
- Service level: This monitors a service provided by a server (such as a WebApp deployed on the Tocmcat).
- Cluster level: This monitors the cluster controller.
- Cloud level: This monitors the cloud controller.

The TaaS cloud monitoring is implemented by a monitor server and a set of monitors deployed on the cloud controller, cluster controller, the VM controller and each VM.

5 Experiments

This section presents experiment settings, the parallel processing of test tasks, computing time distribution and algorithm comparisons.

Three physical machines are used in the experiments, each of which has 2 CPUs and 3GB memory. One is installed the cloud controller, the cluster controller and NFS, and the other two are installed XenServer. VMs are created on physical machines and running Windows XP. Testing tasks with different test requirements are running on each VM.

5.1 Processing Test Tasks in Parallel

This experiment observes the time to process 100 test tasks on two physical machines, on which different numbers of VMs are created: 1) create two VMs with 2 virtual CPUs; and 2) create 4 VMs with 1 virtual CPU.

It is observed that the processing time is reduced half by adding two more VMs. In addition, the CPU usage is increased from 50 to 98%. On the other hand, the memory usage almost does not change.

5.2 Computing Time Distribution

When TaaS platform accepts an array of test tasks from tenants, it proceeds three main steps: 1) clustering these tasks according to their test environments; 2) scheduling these clustered tasks according to their priority and dispatching to a certain VM in TaaS cloud; 3) processing these tasks by calling proper testing services in the cloud.

Figure 5 shows the time distribution among the three portions with task sizes varying from 10 to 500.

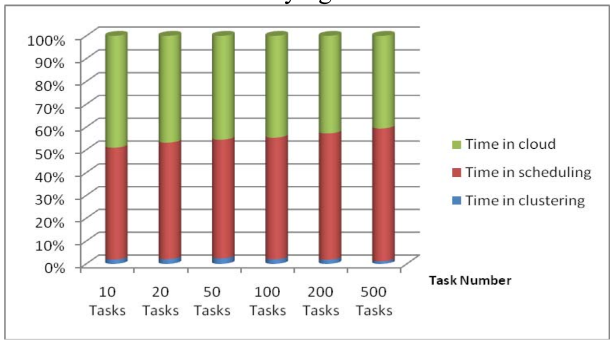

Figure 5: Computing Time Distribution of TaaS Platform

It is observed that up to 99% of time is spent on test task scheduling and processing and only 1% of time on clustering. When the task size is 10, the ratio of scheduling time to testing time is 45/54. With the increasing of task sizes, the ratios increase to 54/44. The scheduling time includes execution time of scheduling algorithm and the task waiting time in waiting queue. Along with the sizes of test tasks, the waiting time increases.

5.3 Algorithms Comparisons

To evaluate the proposed quick cluster algorithm, we compare its cluster time with that of K-Medoids. This experiment uses these two algorithms to cluster tasks with the sizes varying from 100 to 500. Both of these two algorithms cluster these tasks into three clusters.

Figure 6 shows the cluster times of K-Medoids and quick cluster, where the vertical axis represents cluster time (millisecond), and the horizontal axis represents the number of tasks. It is observed that the quick cluster has better efficiency, especially when there have a large number of task requests.

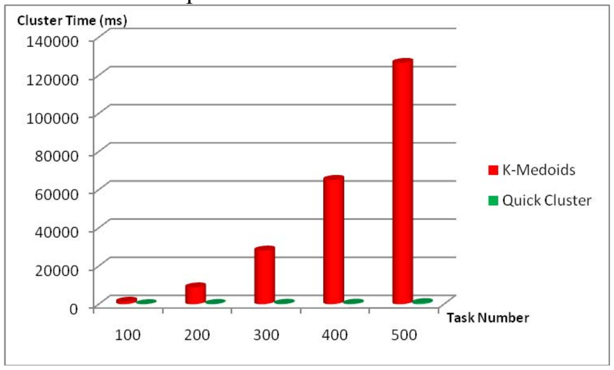

Figure 6: Clustering time of K-Medoids and Quick Cluster

To evaluate the scheduling algorithm, we introduce the Hit Value, and its formula is shown following formula (5). Note that for each task *i*, v_i is its value, T_Finish_i is the finish time, and T_Submit_i is the submitted time.

$$\text{HitValue} = \sum_{i=1}^{n} \frac{v_i}{\sqrt{T\_Finish_i - T\_Submit_i}} \quad (5)$$

Figure 7 shows the hit value comparison between FCFS (First Come First Serve) algorithm and HPF (High Priority First) presented in this paper. One can see that hit value achieved by HPF is more than the hit value achieved by FCFS by 10%-20% especially when dealing with a large number of tasks.

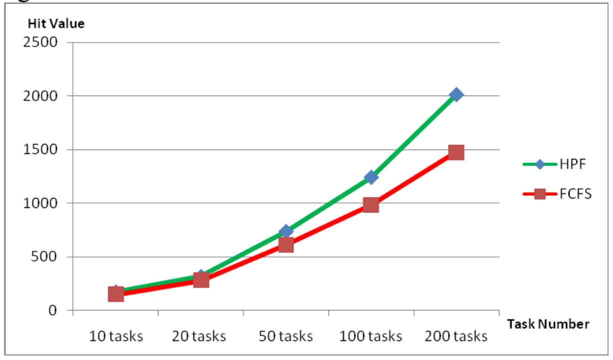

Figure 7: Hit Value of FCFS and HPF

We also observe that the computation time of HPF is similar to that of FCFS. That is HPF improves the scheduling efficient and does not incur high computation time compared with FCFS.

6 Related Work

This paper explores virtualization techniques and test task management to fulfill TaaS on a cloud. The section surveys virtualization in cloud computing and test task scheduling approaches, and finally present existing tools and applications on ontology-based testing and cloud computing for testing.

6.1 Virtualization in Cloud Computing

Kernel-based VM (KVM) is a Linux kernel virtualization infrastructure. It currently supports native virtualization using Intel VT or AMD-V [7]. VMware vCloud delivers a single way to run, manage, and secure applications [8]. However, this is a commercial tool and cannot know their architecture and design. In TaaS over Cloud, the virtualization technology is based on Xen, which is a VM monitor for x86 that supports execution of multiple guest operating systems with unprecedented levels of performance and resource isolation.

Eucalyptus is an open-source software framework for cloud computing that implements what is commonly referred to as Infrastructure as a Service (IaaS) [10]. However, to the best of our knowledge, it cannot be real-time monitoring the tasks and VMs. In addition, it cannot create/delete/live-migrate VMs according to their resource utilization.

6.2 Test Task Management on Cloud

Test task management on TaaS platform utilizes task clustering and scheduling. In general, main algorithms can be divided into the following categories [11]: partitioning methods, hierarchical methods, density-based methods, grid-based methods, and model-based methods. This paper explores an improved partitioning method.

Priority-based scheduling methods include the earliest deadline first (referred to as EDF) [12], the shortest idle priority (least slack first, referred to as LSF) [13], the earliest release of the highest priority [13], the value of the highest priority [14] and other policies.

Priority determined by only one characteristic parameter is not enough, e.g., the EDF strategy assigns high priority to the task with the earliest deadline, and LSF strategy assigns the highest priority to a task with shortest relaxation time. These algorithms show their optimality in normal case, but in the case of overloading, the system cannot guarantee that it is able to reach a maximum hit value. This paper presents a High Priority First algorithm, abbreviated as HPF by considering two parameters, e.g., task waiting time and task value.

6.3 Cloud Computing for Testing

Hyperic is a platform for web and enterprise application management which can monitor various web application infrastructure including physical virtual environments that is the same with what the proposed system provides. Cloud Foundry is a self-service, pay-as-you-go, public cloud deployment platform for Java web applications that unifies the build, run, and manages application lifecycle for Java [16]. SOASTA is another system that harnesses the power of cloud computing to provide cloud testing [17]. TaaS not only provides testing services and but also uses cloud monitoring to optimize computing resources through auto-creating/deleting/migration VMs according to their utilization to improve user experience.

Testing service-oriented started around 2002 [18][19][20], and since then many techniques are available including online testing, ranking, automated test case geneartion, monitoring, simulation, and policy enforcement, reliability modeling, dynamic profiling, and data provenance. TaaS shares the same goals as these techniques; however, TaaS can take advantages of the cloud resources but needs to control these resources dynamically. Bai *et al.* [15] propose an ontology-based approach for Web Services (WS) testing, and define a test ontology model (TOM) to specify the test concepts, relationships, and semantics from two aspects: test design (such as test data, test behavior, and test cases) and test execution (such as test plan, schedule and configuration).

7 Conclusion

This paper proposed an automated testing platform TaaS on a cloud. This platform adopts cloud computing technique to build the elastic resource infrastructures, and provide various kinds of testing services to testing users. To validate TaaS platform, we used unit testing services to perform the experiments. This platform helps testers to set up unit testing environment, select a suitable unit testing method and testing service for the test task, automatically generate test cases, automatically execute test cases, at last collect the test results, and report to testers. The process is automatically completed, thus maximally saves tester's effort for performing a unit testing task.

In the future, we plan to deploy more testing services on TaaS cloud platform, and collect a variety of runtime information to perform corresponding analysis on scalability and reliability.

Acknowledgement:

This work is partially supported by the National Science Foundation of China (No.60973001), IBM China Research Lab (No.20090101), and U.S. Department of Education FIPSE project.

References:

[1] Lian Yu, Le Zhang, Huiru Xiang, Yu Su, Wei Zhao, Jun Zhu, "A Framework of Testing as a Service", Proceedings of the Conference of Information System Management 2009.

[2] Lian Yu, Shuang Su, Jing Zhao, et al, "Performing Unit Testing Based on Testing as a Service (TaaS) Approach", Proceedings of International Conference on Service Science (ICSS) 2008, pp. 127-131.

[3] Wang YY, Wang Q, Wang HA, Jin H, Dai Gz. "A real-time scheduling algorithm based on priority table and its implementation", Journal of Software, 2004, 15(3):360~370. (In Chinese)

[4] D. Mosse, et al., "Analysis of a fault-tolerant multiprocessor scheduling algorithm," in Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers, Twenty-Fourth International Symposium on, 1994, pp. 16-25.

[5] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif, "Black-box and Gray-box Strategies for Virtual Machine Migration", 4th USENIX Symposium on Networked Systems Design & Implementation, Pp. 229–242 of the Proceedings.

[6] Paul Barham, Boris Dragovic, Keir Fraser, *et al,* "Xen and the Art of Virtualization", Symposium on Operating Systems Principles (SOSP), 2003, 1-58113.

[7] Kernel-based Virtual Machine: http://en.wikipedia.org/wiki/Kernel-based_Virtual_Machine

[8] VMware Cloud Computing: http://www.vmware.com/solutions/ cloud-computing/

[9] Xen Cloud Platform : http://www.xen.org/products/cloudxen.html

[10] Graziano Obertelli, Sunil Soman, Lamia Youseff, Dmitrii Zagorodnov. "The Eucalyptus Open-source Cloud-computing System", Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid table of contents Pages, 2009. 124~131.

[11] J. Han and M.Kamber, "Data Mining: Concepts and Techniques", Morgan Kaufmann Publishers Inc.2000

[12] C. Liu, J. Layland "Scheduling algorithms for multiprogramming in a hard real-time environment", Journal of the ACM, 1973, 20(1): 46~61.

[13] Y.S. Liu, X.G. He, CJ Tang, L. Li "Special Type Database Technology", Beijing: Science Press, 2000 (in Chinese).

[14] Jensen ED, Locke CD, Toduda H. "A time-driven scheduling model for real-time operating systems", In: Proceedings of the IEEE Real-Time Systems Symposium. Washington, DC: IEEE Computer Society Press, 1985. 112~122.

[15] X. Bai, S. Lee, W. T. Tsai and Y. Chen, "Ontology-Based Test Modeling and Partition Testing of Web Services", in the 6th International Conference on Web Services (ICWS), China, 2008, pp. 465-472.

[16] Cloudfoundry: http://www.cloudfoundry.com/

[17] Cloud Test by SOASTA: http://www.soasta.com/

[18] W. T. Tsai, R. Paul, Y. Wang, C. Fan, and D. Wang, "Extending WSDL to Facilitate Web Services Testing", Proc. of IEEE High-Assurance Systems Engineering, 2002, pp. 171-172.

[19] W. T. Tsai, R. Paul, W. Song, and Z. Cao, "Coyote: An XML-based Framework for Web Services Testing", Proc. of IEEE High-Assurance Systems Engineering, 2002, pp. 173-174.

[20] W. T. Tsai, "Verification of Web Services Using an Enhanced UDDI Server", Proc. of IEEE WORDS, 2003.