Optimization and Stabilization of Composite Service Processing in Cloud System

Sheng Di[1], Derrick Kondo[1], Cho-Li Wang[2]
[1]INRIA, France, [2]The University of Hong Kong, Hong Kong
{sheng.di,derrick.kondo}@inria.fr, clwang@cs.hku.hk

***Abstract*—With increasingly mature virtual machine (VM) technology, the compute resources provided by Cloud systems can be divided or isolated on demand under a payment model. By leveraging such a feature, we design and implement a cloud system that can optimize the overall performance of processing user requests which are made up of composite services. Specifically, we aim to minimize the response time for each user request, and also maximize the fairness of the treatment for the competitive situation in short supply. We first design an optimal VM resource allocation scheme with a minimized VMM operation cost for each task. Then, for maximizing the fairness of the treatment in the competitive situation, we design a best-suited queuing policy and a resource sharing scheme adjusted based on Proportional-Share model, which can effectively disperse the resource contention. Experiments confirm two points: (1) the mean task response time is close to the theoretically optimal value in the non-competitive situation; (2) when the system runs in short supply, each request could still be processed efficiently, with just a slight extension on their response times compared to their ideal values. The solution that combines Lightest Workload First (LWF) queuing policy with our designed Adjusted Proportional-Share Model (LWF+APSM) exhibits the best and stable performance. It outperforms other solutions in the competitive situation, by 38% w.r.t. the worst-case response time and by 12% w.r.t. the fairness of the treatment.**

I. Introduction

Cloud computing [1] has become such a flexible platform that allows users to customize their own services based on specific purposes. Platform as a Service (PaaS) is one of its classical paradigms. A typical example is Google App Engine [2], which provides a platform for users to easily deploy and release their own services on the Internet.

Our Cloud scenario is similar to the PaaS model, in which the users can compose complex requests based on their specific needs, by combining a set of off-the-shelf web services in series. These services are supposed to be composed by some authorized users (a.k.a., service makers). Each service is associated with a price, which is assigned by its maker. When a user submits a compute request that calls other services, he/she needs to pay the usage of these services and the payment is determined by how much resource to be consumed. On the other hand, to maximize the resource consolidation, we leverage the VM technology to refine the resource allocation, which is completely transparent to users. The key question is how to split the physical resources according to different users' requirements, how to minimize the negative side-effect (a.k.a., overhead) of data transmission and virtual machine monitor (VMM) operations, and how to queue user requests when necessary.

Our objective is to optimize and stabilize the QoS of each *user request*, in both non-competitive (or adequate supply) and competitive (or short supply) situations. We use the term *user request*, *job* and *task* interchangeably in the following text. For the non-competitive situation, the available resources are relatively adequate for users' demands, so the optimality is mainly determined by task's intrinsic structure. In our Cloud system, each task is made up of a set of subtasks (instance of web service) connected one by one (like a serial workflow), and the whole response time (or wall-clock time) of each task is expected to be minimized. We formulate such a problem to be a convex-optimization model [3], with a particular task execution type and specific budget constraint. Thus, it is feasible to find the optimal solution quickly. However, a few practical issues (e.g, related to the VM-invocation overheads and network communication cost) have to be taken into account. Since the output of any non-terminal subtask will be treated as the input of its succeeding one, the data transmission delay cannot be overlooked if the data size is huge. On the other hand, since we will take advantage of the VM resource isolation to refine the resource allocation, the cost of VMM operations (such as the time cost in performing CPU-capacity changing command for VMs at runtime) is also supposed to be minimized. Our cost-minimization strategy is performing the data transmission and VMM operations concurrently, based on the characterization of their costs.

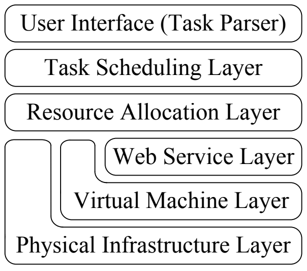
For the competitive situation where the system runs in short supply, we aim to keep each task's QoS at a high level and maximize the overall fairness of treatment meanwhile. This issue is quite challenging in that each task's execution is determined by a different structure that is made up of multiple subtasks corresponding to various services, and also associated with a varied budget to restrict its total payment. A competitive situation with limited available resources may easily delay some particular responses, tending to cause the unfairness of the treatment. In our experiment, we find that assigning different priorities to tasks in the task scheduling phase and the resource allocation stage would induce significantly different effects on the overall fairness and stability. Hence, we investigate the best-suited queuing policies for maximizing the processing fairness of QoS in a Cloud environment. The candidate queuing policies include First-Come-First-Serve (FCFS), Shortest-Optimal-Length-First (SOLF), Lightest-Workload-First (LWF), Shortest-SubTask-First (SSTF) (a.k.a., min-min), and Slowest-Progress-First (SPF). SOLF assigns higher priorities to the tasks with shorter theoretically optimal execution length estimated based on our convex-optimization model, which is similar to the Heterogeneous Earliest Finish Time (HEFT) [4]. LWF and SSTF can be considered Shortest Job First (SJF) and min-min algorithm [5] respectively. The intuitive idea of SPF is similar to Earliest Deadline First (EDF) [6], wherein we adopt two criteria to evaluate the task execution progress. In addition, we further exploit a best-fit resource allocation scheme to adapt to the competitive situation. Specifically, we investigate how to coordinate the divisible resource allocation among the running tasks in terms of their structures like workload or varied estimated progress.

Based on the cloud composite service model, we implement a distributed prototype that is able to solve/calculate complex matrix problems submitted by users. Experiments show that the worst-case performance under SWF is higher than that under other policies by about 38% when overall resource amount requested is about twice as the resource amount that can be allocated. Another key lesson we learned is that in the competitive situation, short jobs (with the short single-core execution length) are better to be assigned with more powerful resources than the theoretical values optimized by the convex-optimization theory.

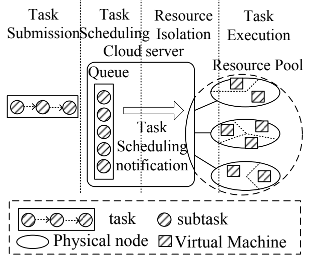
In the remainder of the paper, we will use the term *host*, *machine*, and *node* interchangeably. In Section II, we describe the architecture of our Cloud system, namely cloud composite service system. In Section III, we formulate the research problem in our Cloud environment, to be aiming to maximize individual task's QoS and the overall fairness of treatment meanwhile. In Section IV, we discuss how to optimize the execution of each task with minimized overheads, and how to stabilize the QoS especially in a competitive situation. We present experimental results in Section V. We discuss the related works in Section VI. Finally, we conclude the paper with a vision of the future work in Section VII.

II. System Architecture

The system architecture of our cloud composite service system is shown in Figure 1 (a). The top layer is user interface, which is used to spawn particular threads to receive and respond to user requests. A user request (a.k.a., a task) is made up of multiple subtasks, which are connected in series. Each subtask is an instance of an off-the-shelf service that has a very convenient interface (such API) to be called. For example, a user may submit a task which is composed of a set of simple matrix calculations: matrix-product, matrix-normalization, matrix-decomposition, and so on. Each such matrix calculation can be considered a subtask, and the whole task is expected to be completed as soon as possible under the constraint of its budget. Task scheduling is a key layer used to coordinate the priorities of the tasks such that they can be treated in a fair way. Resource allocation layer is responsible for calculating the optimal resource fraction for the subtasks, and performing the task execution on the isolated virtual resources. Each physical host runs multiple VMs, on each of which are deployed with all of the off-the-shelf services (e.g., the libraries or programs that compute matrix formulas). Each subtask will be executed on a VM, with an amount of virtual resource (a.k.a., resource fraction) tuned by the substrate VM monitor (VMM, a.k.a., hypervisor). Our work will be focused on the three issues that involve the five bottom layers, how to schedule the tasks with as fair treatment as possible, how to allocate the virtual resources to get the optimal performance, and how to perform the task execution with minimized overheads.


User Interface (Task Parser)
Task Scheduling Layer
Resource Allocation Layer
Web Service Layer
Virtual Machine Layer
Physical Infrastructure Layer

(a) System Architecture



(b) Task Processing Procedure

Figure 1. System Overview of Cloud Composite Service System

Each task is processed according to the pseudo-code shown in Algorithm 1, also as shown in Figure 1 (b). At

Algorithm 1 Procedure of Processing a Task

```
Input: Task $t$={$t_{(1)}$,$t_{(2)}$,$\cdots$,$t_{(m)}$};
Output: The computing output of the task $t$.
1: Predict workload for task $t$, denoted as $l(t)$=($l(t_{(1)})$,$\cdots$,$l(t_{(m)})$)$^T$;
2: for ($i$=1 $\rightarrow$ $m$) do
3:     Compute the optimal resource for $t_{(i)}$, $t_{(i+1)}$, $\cdots$, $t_{(m)}$, based on convex optimization;
4:     Put $t_{(i)}$ in a queue and wait until receiving an execution notification;
5:     Upon receiving the notification, perform the resource isolation for the VM selected by the task scheduler;
6:     Trigger the web service on the VM, and execute $t_{(i)}$;
7: end for
```

the beginning, the task submitted will be analyzed by a *task parser* (in the user interface module), in order to predict the subtask workloads based on their input parameters. The optimal resource vector for all the subtasks in the task t will then be computed based on convex optimization, and the output is denoted as $\boldsymbol{r}^*(t)$=($r^*(t_{(1)})$, $r^*(t_{(2)})$, $\cdots$, $r^*(t_{(m)})$)^T. After that, the first unprocessed subtask (denoted as $t_{(i)}$) will be put in a queue and registered with its optimal resource demand (denoted as $\boldsymbol{r}^*(t_{(i)})$), waiting for the task scheduling notification with a selected qualified physical host on top of which running an idle VM. As $t_{(i)}$ is scheduled, the hypervisor of the selected physical machine will perform the resource isolation for the selected VM to match $t_{(i)}$'s demand. The corresponding service on the VM will be called with $t_{(i)}$'s input parameters, and the output will be cached in the VM, waiting for the notification of the data transmission for its succeeding subtask.

We adopt XEN's credit scheduler [7] to perform the resource isolation among VMs on the same physical machine. With XEN [8], we can dynamically isolate some key resources (like CPU rate and network bandwidth) to suit the specific usage demands of different VMs. There are two key concepts in the credit scheduler, *capacity* and *weight*. Capacity specifies the upper limit on the CPU rate consumable by a particular VM, and weight means a VM's proportional-share credit. On a relatively free physical host, the CPU rate of a running VM is determined by its capacity. If there are over-many VMs running on a physical machine, the real CPU rates allocated for them are proportional to their weights. Both capacity and weight can be dynamically tuned at runtime no matter whether the target VMs are running some applications or not.

III. Problem Formulation

Assuming there are n tasks to be processed by the system, and they are denoted as t_i, where i=1,2,$\cdots$,n. Each task can be considered a series workflow, which is made up of multiple subtasks connected in series. We denote the subtasks of the task t_i to be $t_{i(1)}$, $t_{i(2)}$, $\cdots$, $t_{i(m_i)}$, where m_i refers to the number of subtasks in t_i. Such a design glitters a generic execution model, wherein any user request can be constructed by multiple embedded composite services.

Since each task is composed of a series of subtasks, its total execution time (or execution length) can be denoted as $T(t_i)$=$\sum_{j=1}^{m_i} \frac{l_{i(j)}}{r_{i(j)}}$, where $l_{i(j)}$ and $r_{i(j)}$ are referred to as the workload of subtask $t_{i(j)}$ (such as the number of instructions, data to read from disk) and the compute resource allocated (such as CPU rate, disk I/O bandwidth) respectively. Such a definition specifies a defacto broad set of applications (an affine transformation), each of which can be executed with varied resources over different stages, adapting to dynamic changes of resource intensities. We will use execution time, execution length, response length, and wall-clock time interchangeably in the following text. Subtask's workload can be characterized using $\{$resource_processing_rate$\times$subtask_execution_length$\}$ based on past traces or workload prediction approaches like polynomial regression method [9]. Each subtask $t_{i(j)}$ will call a particular service API, which is associated with a service price (denoted as $p_{i(j)}$). The service prices ($/unit) are determined by corresponding service makers in our model, since they are the ones who pay monthly resource leases to Infrastructure-as-a-Service (IaaS) providers (e.g., Amazon EC2 [10]). The total payment in executing a task t_i on top of service layer is equal to $\sum_{j=1}^{m_i}[r_{i(j)} \cdot p_{i(j)}]$. Each task is associated with a budget (denoted as $B(t_i)$) by its user in order to control its total payment. Hence, the problem of optimizing task t_i's execution can be formulated as Formula (1) and Formula (2) (convex-optimization problem).

$$\min\ T(t_i) = \sum_{j=1}^{m_i} \frac{l_{i(j)}}{r_{i(j)}} \tag{1}$$

$$s.t.\ \sum_{j=1}^{m_i} [r_{i(j)} \cdot p_{i(j)}] \le B(t_i) \tag{2}$$

There are two metrics to evaluate the system performance. One is *Response Extension Ratio* (RER) of each task (defined in Formula (3)).

$$RER(t_i) = \frac{t_i's\ real\ response\ time}{t_i's\ theoretically\ optimal\ length} \tag{3}$$

The RER is used to evaluate the execution performance for a particular task. The lower value the RER is, the higher execution efficiency the corresponding task is processed with in reality. A task's *theoretically optimal length* (TOL) is the sum of the theoretical execution time of each subtask with the optimal resource allocation. The optimal resource allocation is the solution to the above convex-optimization problem (Formula (1) and Formula (2)), to be described later. The *response time* here indicates the whole wall-clock time from its submission moment to its final completion moment. In general, the response time of a task is made up of the following 4 parts of all of its subtasks, subtask's waiting time, overhead before the subtask's execution (such as on resource allocation and data transmission), the subtask's execution time, process overhead after its execution. We try best to minimize the cost at each above part in our design.

The other metric is the fairness index of RER among all tasks (defined in Formula (4)), which is used to evaluate the fairness of the treatment in the system. Its value is ranged in [0, 1], and the bigger its value is, the higher fairness of the treatment is. Based on Formula (3), the fairness is also related to the different types of execution overheads. How to effectively coordinate the overheads among different tasks is a very challenging issue. This is mainly due to largely different task structure (i.e., the subtask's workload and the order of its connection), task budget, and dynamically varied resource availability over time.

$$fairness(t_i) = \frac{(\sum_{i=1}^{n} RER(t_i))^2}{n\sum_{i=1}^{n} RER^2(t_i)} \tag{4}$$

Our final objective is to minimize the RER for each individual task (or minimize the maximum RER) and maximize the overall fairness meanwhile, especially in a competitive situation where over-many tasks compete limited resources.

IV. Optimization of System Performance

In order to optimize QoS, we need to minimize the overheads raised at each step in the course of its execution. In general, there are two major reasons for overlarge RER and unfairness of the treatment, especially in a competitive situation: (1) the remarkable waiting time cost in task scheduling; (2) the possible overheads in performing the task execution. We exploit the best-fit solution to the above problem on the three facets, resource allocation, task scheduling, and minimization of overheads.

A. Adjusted Resource Allocation

We design an adjusted scheme to dynamically allocate isolated resources for running tasks. It sets tasks' resource fractions to be their theoretically optimal values in non-competitive situation. We also exploit the best-suited solution for the competitive situation such that each task execution can still be kept with a high QoS in a fair way.

In a non-competitive situation (i.e., the available resources are assumed to be unlimited), the resource fraction allocated to some task is mainly restricted by its user-set budget, which can be formulated as a convex-optimization problem, including a target function (Formula (1)) and a constraint (Formula (2)). We solve it below.

Theorem 1: To minimize $T(t_i)$ subject to the constraint (2), t_i's optimal resource vector $\boldsymbol{r}^*(t_i)$ is shown as Equation (5), where j=1, 2, $\cdots$, m_i.

$$r^*_{i(j)} = \frac{\sqrt{l_{i(j)}/p_{i(j)}}}{\sum_{k=1}^{m_i}\sqrt{l_{i(k)}p_{i(k)}}} \cdot B(t_i) \qquad (5)$$

Proof: Since $\frac{\partial^2 T(t_i)}{\partial r_j}$=$2\frac{l_{i(j)}}{r^3_{i(j)}}$>0, $T(t_i)$ is convex with a minimum extreme point. By combining the constraint (2), we can get the Lagrangian function as Formula (6), where λ refers to the Lagrange multiplier.

$$F(r_i) = \sum\nolimits_{j=1}^{m_i}\frac{l_{i(j)}}{r_{i(j)}} + \lambda(B(t_i) - \sum\nolimits_{j=1}^{m_i} r_{i(j)}p_{i(j)}) \quad (6)$$

We derive Equation (7) via Lagrangian multiplier method.

$$r_{i(1)}:r_{i(2)}:\cdots:r_{i(m_i)} = \sqrt{\frac{l_{i(1)}}{p_{i(1)}}}:\sqrt{\frac{l_{i(2)}}{p_{i(2)}}}:\cdots:\sqrt{\frac{l_{i(m_i)}}{p_{i(m_i)}}} \quad (7)$$

In order to minimize $T(t_i)$, the optimal resource vector $\boldsymbol{r}^*_{i(j)}$ should use up all the budget (i.e., let the total payment be equal to $B(t_i)$). Then, we can get Equation (5). ■

As follows, we discuss the significance of Theorem 1 and how to split physical resources among different tasks based on VM resource isolation in practice. According to Theorem 1, we can easily compute the optimal resource vector for any task based on its budget constraint. Specially, $r^*_{i(j)}$ is the theoretically optimal resource vector (or processing rate) allocated to the subtask $t_{i(j)}$, such that the total wall-clock time of task t_i can be minimized. That is, even though there were more available resources compared to the value $r^*_{i(j)}$, it would be useless for the task t_i due to its limited budget. Hence, our designed resource allocator will set each subtask's CPU capacity[1] (i.e., the maximum CPU rate) as its theoretically optimal resource vector, Formula (5).

If the system runs in short supply, it is likely the total sum of their optimal resources (i.e., $r^*(t_i)$) may exceed the total capacity of physical machines. At such a competitive situation, it is necessary to coordinate the priorities of the tasks in the resource consumption, such that none of tasks' real execution lengths would be extended noticeably compared to its theatrically optimal execution length (i.e., minimizing RER(t_i) for each task t_i). In our system, we improve the proportional-share mechanism with XEN's credit scheduler to control subtask's resource utilization.

Under XEN's credit scheduler, each guest VM on the same physical machine will get its CPU rate that is proportional to its weight[2]. Suppose on a physical host (denoted as h_i), n_i scheduled subtasks are running on n_i stand-alone VMs separately (denoted v_j, where j=1,2,$\cdots$,n_i). We denote the host h_i's total compute capacity to be c_i (e.g., 8 cores), and the weights of the n_i subtasks to be $w(v_1)$, $w(v_2)$, $\cdots$, $w(v_{n_i})$. Then, the real resource share (denoted by $r(v_j)$) allocated to the VM v_j can be calculated by Formula (8).

$$r(v_j) = \frac{w(v_j)}{\sum_{k=1}^{n_i} w(v_k)} c_i \qquad (8)$$

Now, the key question becomes how to determine the value of the weight for each running subtask (or VM) on a physical machine. Based on the definition of RER, a large value of RER tends to appear with a short task, which can also be confirmed by our experiments. This is mainly due to the fact that the overheads (such as data transmission cost, VMM operation cost) in the whole wall-clock time are often relatively constant regardless of the total task workload. That is, based on the definition of RER, short task's RER is more sensitive to the execution overheads than that of a long one. Hence, our design tends to assign higher priorities to short tasks in their resource allocation. Specifically, our intuitive idea is adopting a proportional-share model on most of the middle-size-tasks such that their resource fractions received are proportional to their theoretically optimal resource amounts ($r^*_{i(j)}$). Meanwhile, we enhance the credits of the subtasks whose corresponding tasks are relatively short and decrease the credits of the ones with long tasks. That is, we give some extra credits to short tasks to enhance their resource consumption priority. Suppose on a physical machine is running d subtasks (belonging to different tasks), which are denoted as $t_{1(x_1)}$, $t_{2(x_2)}$, $\cdots$, $t_{d(x_d)}$, where x_i = 1, 2, $\cdots$, or m_i, then, $w(t_{i(j)})$ will be determined by Formula (9). We call it *Adjusted Proportional-Share Model (APSM).*

$$w(t_{i(j)}) = \begin{cases} \eta \cdot r^*_{i(j)} & l_i \le \alpha \\ r^*_{i(j)} & \alpha < l_i \le \beta \\ \frac{1}{\eta} \cdot r^*_{i(j)} & l_i > \beta \end{cases} \qquad (9)$$

The weight values in our design (Formula (9)) are determined by four parts, the extension coefficient (η), theoretically optimal resource fraction ($r^*_{i(j)}$), the threshold value α to determine short tasks, and the threshold value β to determine long tasks. Obviously, the value of η is supposed

[1]the capacity-setting command is "xm sched-credit -d VM -c $r^*_{i(j)}$".

[2]the weight-setting command is "xm sched-credit -d VM -w *weight*".

to be always greater than 1. In reality, tuning η's value could adjust the extension degree for short/long tasks. Changing the values of α and β could tune the number of the short/long tasks. That is, by adjusting these values dynamically, we could optimize the overall system performance to adapt to different contention states. Specific values suggested in practice will be discussed with our experimental results.

B. Best-suited Task Scheduling Policy

In a competitive situation where over-many tasks are submitted to the system, it is necessary to queue some tasks that cannot find the qualified resources temporarily. The queue will be checked as soon as some new resources are released by the finished tasks. As multiple hosts are available for the task (e.g., there are still available CPU rates non-allocated on the host), the most powerful one with the largest availability will be selected as the execution host. A key question is how to select the waiting tasks based on their demands, such that the overall execution performance and the fairness can both be optimized.

Based on our two-fold objective that aims to minimize the RER and maximize the fairness meanwhile, we propose that the best-fit queuing policy is Lightest-Workload-First (LWF) policy, which assigns the highest scheduling priority to the shortest job that has the least workload amount to process. In addition, we also evaluate many other queuing policies for comparison, including First-Come-First-Serve (FCFS), Shortest-Optimal-Length-First (SOLF), Slowest-Progress-First (SPF), and Shortest-Subtask-First (SSF). We describe all the task-selection policies below.

- *First-Come-First-Serve (FCFS).* FCFS schedules the subtasks based on their arrival order. The first arrival one in the queue will be scheduled as long as there are available resources to use. This is the most basic policy, which is the easiest to implement. However, it does not take into account the variation of task features, such as task structure, task workload, thus the performance and fairness will be significantly restricted.
- *Lightest-Workload-First (LWF).* LWF schedules the subtasks based on the predicted workload of their corresponding tasks (a.k.a., jobs). Task's workload is defined as the execution length estimated assuming to be run on a standard process rate (such as single-core CPU rate). In the waiting queue, the subtask whose corresponding task has lighter workload will be scheduled with a higher priority. In our Cloud system that aims to minimize the RER and maximize the fairness meanwhile, LWF obviously possesses a prominent advantage. Note that various tasks' TOLs are different due to their different budget constraints and workloads, while tasks' execution overheads tend to be constant in the system. In addition, the tasks with lighter workloads tend to be with smaller TOLs, based on the definition of $T(t_i)$. Hence, according to the definition of RER, the tasks with lighter workloads (i.e., shorter jobs) are supposed to be more sensitive to their execution overheads, which means that they should be associated with higher priorities.
- *Shortest-Optimal-Length-First (SOLF).* SOLF is designed based on such an intuition: in order to minimize RER of a task, we can only minimize the task's real execution length because its theoretically optimal length (TOL) is a fixed constant based on its intrinsic structure and budget. Since tasks' TOLs are different due to their heterogeneous structures, workloads, and budgets, the execution overheads will impact their RERs to different extents. Suppose there were two tasks whose TOLs are 30 seconds and 300 seconds respectively and their execution overheads are both 10 seconds. Even though the sums of their subtask execution lengths were right the optimal values (30 seconds and 300 seconds), their RERs would be largely different: $\frac{30+10}{30}$ vs. $\frac{300+10}{300}$. In other words, the tasks with shorter TOLs are supposed to be scheduled with higher priorities, in order to minimize the discrepancy among tasks' RERs.
- *Slowest-Progress-First (SPF).* SPF is designed based on the task's real execution progress compared to its overall workload or TOL. The tasks with the slowest progress will have the highest scheduling priorities. The execution progress can be defined based on either the workload processed or the wall-clock time passed. They are called *Workload Progress* (*WP*) and *Time Process* (*TP*) respectively, and they are defined in Formula (10) and Formula (11) respectively. In the two Formulas, d refers to the number of completed subtasks, l_i=$\sum_{j=1}^{m_i} l_{i(j)}$, and $TOL(t_i)$=$\sum_{j=1}^{m_i} \frac{l_{i(j)}}{r^*_{i(j)}}$. SPF means that the smaller value of t_i's $WP(t_i)$ or $TP(t_i)$, the higher t_i's priority would be. For example, if t_i is a newly submitted task, its workload processed must be 0 (or d=0), then $WP(t_i)$ would be equal to 0, indicating t_i is with the slowest process.
$$WP(t_i) = \frac{\sum_{j=1}^{d} l_{i(d)}}{l_i} \tag{10}$$
$$TP(t_i) = \frac{wall\text{-}clock\ time\ since\ t_i' s\ submission}{TOL(t_i)} \tag{11}$$
Based on the two different definitions, the Slowest-Progress-First (SPF) can be split into two types, namely Slowest-Workload-Progress-First (SWPF) and Slowest-Time-Progress-First (STPF) respectively. We evaluated both of them in our experiment.
- *Shortest-Subtask-First (SSF).* SSF selects the shortest subtask waiting in the queue. The shortest subtask is defined as the subtask (in the waiting queue) which has the minimal workload amount estimated based on single-core computation. As a subtask is completed, there must be some new resources released for other tasks, which means that a new waiting subtask will then be scheduled if the queue is non-empty. Obviously,

SSF will result in the shortest waiting time to all the subtasks/tasks on average. In fact, since we select the "best" resource in the task scheduling, the eventual scheduling effect of SSF will make the short subtasks be executed as soon as possible. Hence, this policy is exactly the same as *min-min* policy [5], which has been effective in Grid workflow scheduling. However, our experiments validate that SSF is not the best-suited scheduling policy in our Cloud system.

C. Minimization of Processing Overheads

In our system, in addition to the waiting time and execution time of subtasks, there are three more possibly significant overheads which will also be counted in the whole response time, the time cost in performing VM resource isolation at runtime, the time cost in data transmission between sub-tasks, and the time of restoring VM's default setting after task execution.

In this section, we intensively study how to minimize the negative impact of all these overheads to the task's whole response time at runtime. Our idea is illustrated in Figure 2.

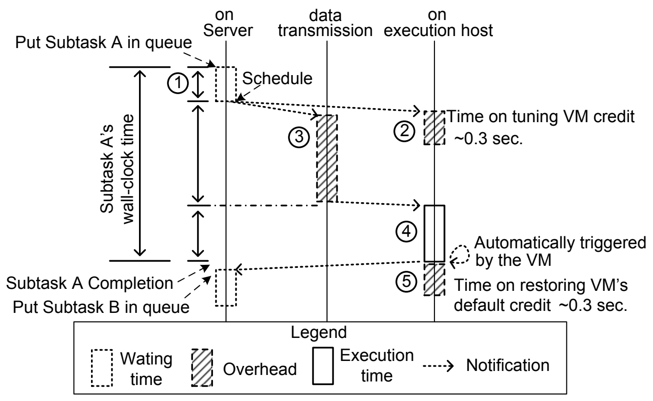


Figure 2. Illustration of Task Response Time

Figure 2 shows the wall-clock time of a subtask in our system, through an example subtask A. At the beginning (Step 1), it will be put in the scheduling queue until there are qualified available resource matching its demand. Then, the scheduler will notify the XEN hypervisor on the selected physical host to perform VM resource isolation (Step 2). Since we perform the VM resource isolation for different VMs by XEN hypervisor [8], we mainly characterize the time cost of dynamically performing XEN's credit-tuning command. We find that the XEN command that tunes a VM's CPU rate at runtime often costs constantly (in about 0.3 seconds), regardless of the VM's properties (such as the memory size and the working state of the VM). This cost cannot be overlooked especially for the short tasks whose TOLs (i.e., theoretically optimal wall-clock time) are short (say several seconds). Consequently, our design adopts two principles to minimize its impact, minimizing the number of VM CPU credit tuning operations in the course of task execution, and also performing the commands in stand-alone threads whose time cost could be excluded from the task's wall-clock time. For example, we always tune VM's capacity and weight values via an integrated command[1] instead of two separate commands as mentioned previously.

In addition, as soon as a physical host is selected for a subtask, the scheduler will immediately perform the data transmission (Step 3) if needed, e.g., when the subtask is not the initial one in the whole task. Specially, if the physical hosts of the previous subtask and the current subtask (in the same task) are different, the output of the previous one needs to be transmitted from its execution host to the new host as the current subtask's input. Such a data transmission will be carried out in a new thread, by notifying the previous execution host to push the data into the host assigned to the current subtask. Such a design makes multiple steps (including the VM resource isolation, data transmission, and possible other tracing/logging operations) run concurrently, mitigating the negative impact of the execution overheads to the whole response time as much as possible.

As soon as the input data arrives at the execution VM on the selected physical host, the corresponding service will be triggered to finish the subtask's workload (Step 4) through the isolated virtual resource. Whenever the execution is done, a daemon on the VM will send a notification to its hypervisor to restore its default setting (including the capacity and weight). The default values of the capacity and weight are both set equal to one-core CPU rate. In our system, for the super-short subtasks (say the one whose TOL is less than or around 2 seconds), we run them directly on VMs without any credit-tuning operation. Otherwise, the credit-tuning effect may work on another subtask instead of the current subtask, due to the inevitable delay (about 0.3 seconds) of the credit-tuning command and the super-short length of the subtask. That is, such a strategy that directly runs super-short subtasks could effectively control the overhead for them and also reduce the possible contention of executing other resource isolation commands on the same machines. All in all, the minimized wall-clock time of each subtask is supposed to be equal or close to the sum of the times cost in Step 1, Step 3 and Step 4.

V. Performance Evaluation

A. Experimental Setting

We implement a cloud composite service prototype that can help solving complex matrix-based problems, each which if allowed to consist of a series of embedded matrix computations. For example, a user may submit a request like *Solve*$((A_{m\times n}\cdot A_{n\times m})^k, B_{m\times m})$. Such a task could be split into three steps (or subtasks): (1) matrix-matrix multiply: $C_{m\times m}=A_{m\times n}\cdot A_{n\times m}$; (2) matrix-power: $D_{m\times m}$ =

[1]the command is "xm sched-credit -d VM -c $r^*_{i(j)}$ -w *weight*"

Table I
WORKLOADS (SINGLE-CORE EXECUTION LENGTH) OF 10 MATRIX OPERATIONS (SECONDS)

Matrix Scale	M-M-Multi.	QR-Decom.	Matrix-Power		M-V-Multi.	Frob.-Norm	Rank	Solve	Solve-Tran.	V-V-Multi.	Two-Norm
500	0.7	2.6	m=10	2.1	0.001	0.010	1.6	0.175	0.94	0.014	1.7
1000	11	12.7	m=20	55	0.003	0.011	8.9	1.25	7.25	0.021	9.55
1500	38	35.7	m=20	193.3	0.005	0.03	29.9	4.43	24.6	0.047	29.4
2000	99.3	78.8	m=10	396	0.006	0.043	67.8	10.2	57.2	0.097	68.2
2500	201	99.5	m=20	1015	0.017	0.111	132.6	18.7	109	0.141	136.6

$C^k_{m\times m}$; (3) Least squares solution of $D{\cdot}X{=}B$ based on QR-Decomposition: *Solve*$(D_{m\times m},\ B_{m\times m})$.

In our experiment, we are assigned with 8 physical nodes to use from the most powerful cluster in HongKong (namely Gideon-II [11]), and each node owns 2 quad-core Xeon CPU E5540 (i.e. 8 processors per node) and 16GB memory size. There are 56 VM-images (centos 5.2) maintained by Network File System (NFS), so 56 VMs (7 VMs per node) will be generated at the bootstrap. XEN 4.0 [8] serves as the hypervisor on each node and dynamically allocates various CPU rates to the VMs at run-time using the credit scheduler.

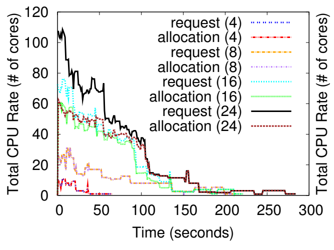
Through a graphical user interface, users can submit their matrix computation requests. In our experiment, we make use of *ParallelColt* [12] to perform the math computations, each consisting of a set of matrix operations. ParallelColt [12] is such a library that can effectively calculate complex matrix operations, such as matrix-matrix multiply and matrix decomposition, in parallel (with multiple threads) based on Symmetric Multiple Processor (SMP) model.

In each test, we randomly generate a number of user requests, each of which is composed of 5∼15 sub-tasks. Each sub-task is randomly selected from 10 basic matrix operations (i.e., 10 services in in Table I). The generated matrix problem must be valid w.r.t. matrix's shape. We also characterize the single-core execution length (or workload) for each service in the table. Among the 10 matrix-computation services, three services are coded via multiple threads, including matrix-matrix multiply, QR-decomposition, matrix-power, hence their computation can get an approximate-linear speedup when allocated multiple processors. The other 7 matrix operation services are implemented using single thread, thus they cannot get speedup when being allocated with more than one processor. Hence, we set the capacity of any subtask performing a single-threaded service to be single-core rate, unless its theoretically optimal resource to allocate is less than one core.

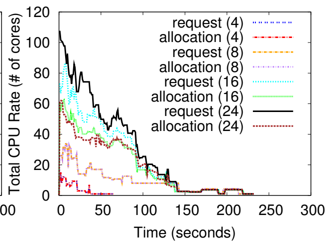
We will evaluate different queuing policies and resource allocation schemes under different competitive situations with different numbers (4-24) of tasks simultaneously.

B. Experimental Results

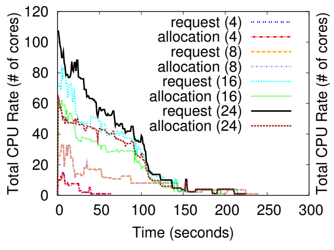
We first characterize the various contention degrees with different number of tasks submitted. The contention degree is evaluated via two metrics, *Allocate-Request Ratio* (abbreviated as *ARR*) and *Queue Length* (abbreviated as *QL*). System's ARR at a time point is defined as the ratio of the total allocated resource amount to the total amount requested by subtasks at that moment. QL at a time point is defined as the total number of subtasks in the waiting list at that moment. There are 4 test-cases each of which uses different number of tasks (4, 8, 16, and 24) submitted. The 4 test-cases correspond to different contention degrees. Figure 3 shows the summed resource amount allocated and the summed amount requested over time under different competitive situations, with exactly the same experimental settings except for different scheduling policies. The numbers enclosed in parentheses indicate the number of tasks submitted.



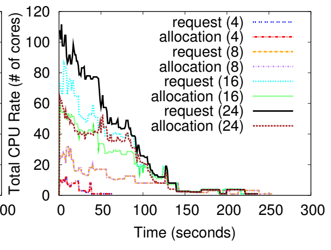
(a) FCFS



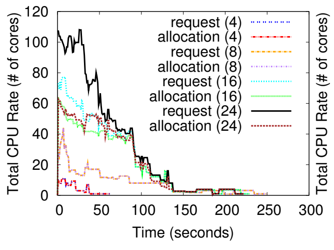
(b) LWF



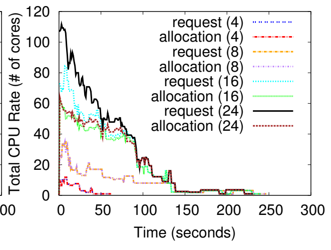
(c) SOLF



(d) SSTF



(e) SWPF

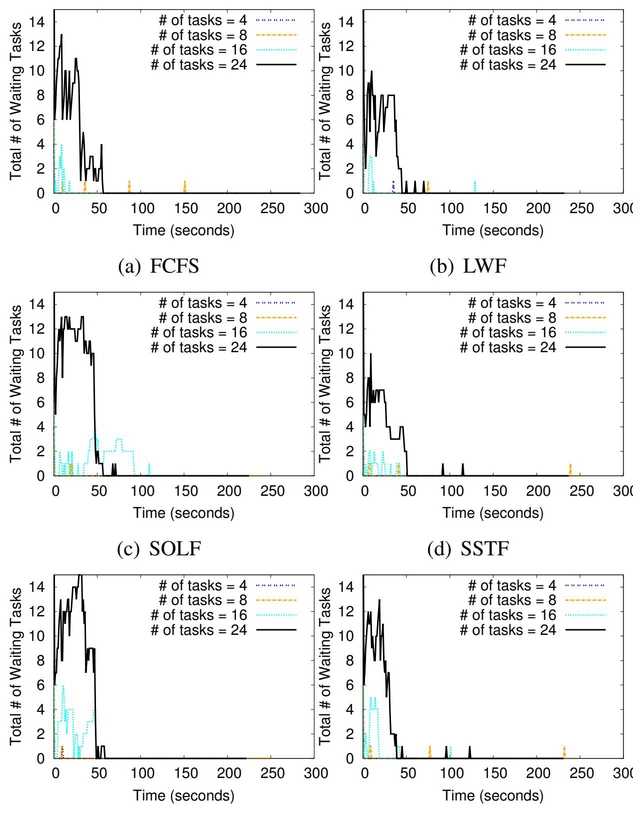


(f) STPF

Figure 3. Allocation vs. Request With Different Contention Degrees

We find that with the same number of submitted tasks, ARR exhibits similarly with different scheduling policies. The resource amount allocated can always meet the resource amount requested (i.e., ARR keeps 1 and two curves overlap in the figure) when there are a small number (4 or 8) of tasks submitted, regardless of the scheduling policies. This confirms our resource allocation scheme can guarantee the service level in the non-competitive situation. As the system runs with over-many tasks (such as 16 and 24) submitted, there would appear a prominent gap between the resource allocation curve and the resource request curve. This clearly indicates a competitive situation. For instance, when 24 tasks are submitted simultaneously, ARR stays around 1/2 during the first 50 seconds. It is also worth noting that the longest task execution length under FCFS is remarkably longer than that under LWF (about 280 seconds vs. about 240 seconds). This implies scheduling policy is essential to the performance of the Cloud system.

Figure 4 presents that the queue length (QL) increases with the number of tasks submitted. It is worth noticing that QL under different scheduling policies exhibits quite different. In the duration with high competition (the first 50 seconds in the test), SSTF and LWF both lead to short average waiting time (about 5-6 seconds and 6-7 seconds respectively). By contrast, under SOLF, SWPF, or STPF, the QL is much longer (about 10-12 seconds), implying a higher cost on waiting to be scheduled.



(a) FCFS (b) LWF (c) SOLF (d) SSTF (e) SWPF (f) STPF

Figure 4. Queue Lengths With Different Contention Degrees

In addition, we explore the best-suited scheduling policy, and validate the effectiveness of the adjusted resource allocation scheme with various coefficients. We set $\{\alpha, \beta\}$ to the 9 combinations of $\{$5 sec., 10 sec., 20 sec.$\}\times\{$100 sec., 200 sec., 300 sec.$\}$), and η is set to $\sqrt{2}$. Experiments show that the best-suited scheduling policy is LWF and our designed resource allocation method (denoted as Adjusted-PSM or APSM) which treats task priorities based on task workloads can effectively improve the task execution performance. In the competitive situation, APSM outperforms the simple proportional-share model (PSM) prominently.

Table II shows the response extension ratio (RER) of our system running in short supply (when there are 24 tasks submitted). It is observed that LWF+APSM is the best choice, which significantly outperforms other strategies by at least $\frac{7.230}{5.239}-1=38\%$ w.r.t. the max. value of RER, and by at least $\frac{0.714}{0.638}-1=12\%$ w.r.t. the fairness index of RER. In addition, as there are 16 tasks submitted, RER's maximum values under LWF+APSM, SSTF+APSM, and FCFS+APSM are 2.234, 4.248, and 3.528 respectively, and the fairness indexes are 0.884, 0.738, and 0.770 respectively. This further confirms the remarkable advantage of our strategy (LWF+APSM) working in a competitive state.

Table II
COMPARISON OF RER IN A COMPETITIVE SITUATION

strategy	min.	avg.	max.	fairness
FCFS+PSM	0.732	3.665	21.097	0.345
FCFS+APSM	0.644	3.779	21.755	0.358
LWF+PSM	0.712	1.809	5.974	0.703
LWF+APSM	0.666	1.790	**5.239**	**0.714**
SOLF+PSM	0.720	3.331	17.004	0.482
SOLF+APSM	0.730	2.780	10.803	0.575
SSTF+PSM	0.746	2.108	8.706	0.573
SSTF+APSM	0.769	2.119	7.230	0.638
SWPF+PSM	0.708	6.106	57.928	0.209
SWPF+APSM	0.649	6.233	59.627	0.206
STPF+PSM	0.707	2.830	14.867	0.476
STPF+APSM	0.713	3.147	15.853	0.474

We analyze the reasons why experimental results differ a lot under different scheduling policies below. SWPF and STPF perform badly among all policies. In particular, SWPF works so poorly that its RER is even greater than 50. Let us review the Formula (10) and Formula (11). In SPF, smaller value of $WP(t_i)$ or $TP(t_i)$ will lead to higher priority, indicating that the task runs with the slowest progress. However, based on the two formulas, longer task (with larger l_i and $T(t_i)$) also tends to make $WP(t_i)$ and $TP(t_i)$ smaller. That is, such a policy actually tends to assign higher priority to longer task. Such a side-effect works oppositely against to the shortest job first intuition, e.g., LWF and STPF, thus many tasks would suffer higher waiting cost on task scheduling. In contrast, LWF and SSTF significantly outperform others, probably due to the fact that they both suffer significantly lower waiting cost in task scheduling (as confirmed in Figure 4). In comparison to SSTF, LWF possesses a particular advantage by taking into account the task's overall workload, which tends to get smaller RER. It is also observed that LWF+APSM works better than SOLF+APSM by 38% at the worst case. This is mainly due to the fact that workload is a immutable metric while task length is relatively mutable. In other words, a task's real execution length is hard to control in that it may be influenced by many unpredictable factors in practice. Hence, the accuracy of the estimated theoretically optimal length (TOL) may be of large errors, misleading task scheduling.

In addition, from Table II, we find that our designed APSM is indeed able to improve the performance in most of cases. For the example of the maximum RER, LWF+APSM and SOLF+APSM outperform LWF+PSM and SOLF+PSM by $\frac{5.974}{5.239}-1=14\%$ and $\frac{17.004}{10.803}-1=57.4\%$ respectively.

Finally, we evaluate the effectiveness of our design in the non-competitive situation (when there are only 4 tasks submitted), as shown in Table III. In such a situation, all tasks can always be allocated with theoretically optimal resources due to the non-competitive state in the system. Thus, the performance under different queueing policies did not differ a lot. Specifically, the mean value of the task execution length is only slightly higher than its theoretically optimal value by 21.4%−37.6%. The maximum RERs and the fairness indexes of all strategies are always lower than 3 and greater than 0.8 respectively. In comparison to the competitive situation, PSM usually outperforms APSM slightly in the non-competitive situation. That is, the APSM will get the resource allocation among tasks be a little over-adjusted against the optimal solution. Such a lesson inspires us to optimize the performance for the both situations by an adaptive solution, which will be our future work.

Table III
COMPARISON OF RER IN A NON-COMPETITIVE SITUATION

strategy	min.	avg.	max.	fairness
FCFS+PSM	0.966	1.300	2.052	0.891
FCFS+APSM	0.878	1.243	2.041	0.878
LWF+PSM	0.933	1.308	2.092	0.876
LWF+APSM	0.863	1.320	2.331	0.840
SOLF+PSM	0.901	1.376	2.723	0.811
SOLF+APSM	0.871	1.324	2.205	0.863
SSTF+PSM	0.911	1.270	2.000	0.893
SSTF+APSM	0.891	1.327	2.318	0.839
SWPF+PSM	0.882	1.125	1.581	0.929
SWPF+APSM	0.860	1.214	2.183	0.845
STPF+PSM	0.941	1.262	2.044	0.881
STPF+APSM	0.883	1.369	2.440	0.829

VI. RELATED WORK

Although job scheduling problem [13] in Grid computing [14] has been extensively studied for years, most of them (such as [15], [16]) are not suited for our cloud composite service processing environment. Grid jobs are often with long execution length, while Cloud tasks are often short based on [17]. Hence, scheduling/execution overheads (such as waiting time and data transmission cost) may impact Cloud task's response time more than Grid job's, implying they must be carefully minimized in the Cloud model.

Recently, many new scheduling methods are proposed for different Cloud systems. M. Zaharia et al. [18] designed a task scheduling method to improve the performance of Hadoop [19] for a heterogeneous environment (such as a pool of VMs each customized with different compute abilities). Unlike the FCFS policy and speculative execution model originally used in Hadoop, they designed a so-called Longest Approximate Time to End (LATE) policy, that assigns higher priorities to the jobs with longer remaining execution lengths. Their intuition is maximizing the opportunity for a speculative copy to overtake the original and reduce job's response time. M. Isard et al. [20] proposed a fair scheduling policy (namely Quincy) for a high performance compute system with virtual machines, in order to maximize the scheduling fairness and minimize the data transmission cost meanwhile. Compared to these works, our Cloud system works with a strict payment model, under which the optimal resource allocation for each task can be computed based on convex optimization theory. M. Mao et al. [21] propose a solution by combining dynamic scheduling and earliest deadline first (EDF) strategy, to minimize user payment and meet application deadlines meanwhile. Whereas, they overlook the competitive situation by assuming the resource pool is always adequate and users have unlimited budgets.

In addition to scheduling model, many Cloud management researchers focus on the optimization of resource assignment. Unlike Grid systems whose compute nodes are exclusively consumed by jobs, the resource allocation in Cloud systems are able to be refined by leveraging VM resource isolation technology. M. Stillwell et al. [25] exploited how to optimize the resource allocation for service hosting on a heterogeneous distributed platform. Their research is formalized as a Mixed Integer Linear Program (MILP) problem and treated as a rational LP problem instead, also with fundamental theoretical analysis based on estimate errors. In comparison to their work, we intensively exploit the best-suited scheduling policy and resource allocation scheme for the competitive situation. We also take into account user payment requirement, and evaluate our solution on a real-VM-deployment environment which needs to tackle more practical technical issues like minimization of various execution overheads. X. Meng et al. [22] analyzed VM-pairs' compatibility in terms of the forecasted workload and estimated VM sizes. SnowFlock [23] is another interesting technology that allows any VM to be quickly cloned (similar to UNIX process fork) such that the resource allocation would be automatically refined at runtime. S. Kuribayashi [24] also proposed a resource allocation method for Cloud computing environments especially based on divisible resources. The key advantage of our design in comparison is to optimize each task's QoS and the overall fairness at a satisfactory level meanwhile, especially for a competitive situation.

VII. CONCLUSION AND FUTURE WORK

In this paper, we designed and implemented a loosely-coupled Cloud system with web services deployed on multiple VMs, aiming to improve and stabilize the QoS of each user request at runtime. Our contribution is three-fold: (1) we intensively studied the best-suited task scheduling policy under such a composite service processing model; (2) we explored an optimal resource allocation scheme and an adjusted strategy that can suit the competitive situation; (3) the processing overhead is minimized in our design. Experiments confirm that the best solution for the competitive situation is applying the Lightest-Workload-First (LWF) in task scheduling plus a Proportional-Shared resource allocation with the credit being set to the adjusted task workload. It outperforms other solutions in the competitive situation, by 38% w.r.t. the worst-case response time and by 12% w.r.t. the fairness of the treatment. In a non-competitive situation, different queuing policies perform similarly, and the task execution length is only slightly higher than its theoretically optimal value by 21.4%−37.6%, and the fairness can be kept over 0.8. In the future, we plan to explore the most accurate coefficients (e.g., η) for our adjusted resource allocation, in both theory and practice. We also plan to further exploit an adaptive solution that can dynamically optimize the performance in both competitive and non-competitive situations.

Acknowledgments

This work was made by the ANR project Clouds@home (ANR-09-JCJC-0056-01), and also in part by a Hong Kong UGC Special Equipment Grant (SEG HKU09).

References

[1] M. Armbrust and et al., "Above the clouds: A berkeley view of cloud computing," EECS, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, Feb 2009.

[2] Google app engine: online at http://code.google.com/appengine/.

[3] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2009.

[4] M. Rahman, S. Venugopal, and R. Buyya, "A dynamic critical path algorithm for scheduling scientific workflow applications on global grids," in *In Proceedings of the 3rd IEEE International Conference on e-Science and Grid Computing*, 2007.

[5] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, "Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems," in *Proceedings of the Eighth Heterogeneous Computing Workshop (HCW '99)*, Washington, DC, USA: IEEE Computer Society, 1999, p. 30.

[6] EDF Scheduling: http://en.wikipedia.org/wiki/earliest_deadline_first_scheduling.'

[7] Xen-credit-scheduler: http://wiki.xensource.com/xenwiki/creditscheduler.

[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03)*. New York, NY, USA: ACM, 2003, pp. 164–177.

[9] L. Huang, J. Jia, B. Yu, B.G. Chun, P. Maniatis, and M. Naik, "Predicting Execution Time of Computer Programs Using Sparse Polynomial Regression," in *Proceedings of 24th International Conference on Neural Information Processing Systems (NIPS'10)*. 2010, pp. 1–9.

[10] Amazon elastic compute cloud: on line at http://aws.amazon.com/ec2/.

[11] Gideon-II Cluster: http://i.cs.hku.hk/∼clwang/Gideon-II.

[12] P. Wendykier and J. G. Nagy, "Parallel colt: A high-performance java library for scientific computing and image processing," *ACM Trans. Math. Softw.*, vol. 37, pp. 31:1–31:22, September 2010.

[13] C. Jiang, C. Wang, X. Liu, and Y. Zhao, "A survey of job scheduling in grids," in *Proceedings of the joint 9th Asia-Pacific web and 8th international conference on web-age information management conference on Advances in data and web management (APWeb/WAIM'07)*, Berlin, Heidelberg: Springer-Verlag, 2007, pp. 419–427.

[14] I. Foster and C. Kesselman, *The Grid 2: Blueprint for a New Computing Infrastructure (The Morgan Kaufmann Series in Computer Architecture and Design)*.Morgan Kaufmann, November 2003.

[15] E. Imamagic, B. Radic, and D. Dobrenic, "An approach to grid scheduling by using condor-G matchmaking mechanism," in *Proceedings of the 28th International Conference on Information Technology Interfaces*, 2006, pp. 625–632.

[16] Y. Gao, H. Rong, and J. Z. Huang, "Adaptive grid job scheduling with genetic algorithms," *Future Generatio Computer Systems*, vol. 21, pp. 151–161, January 2005.

[17] S. Di, D. Kondo, and W. Cirne, "Characterization and comparison of cloud versus grid workloads," in *Proceedings of the IEEE International Conference on Cluster Computing (Cluster'12)*, 2012.

[18] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI'08)*, Berkeley, CA, USA: SENIX Association, 2008, pp. 29–42.

[19] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*, 2010, pp. 1–10.

[20] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP '09)*, New York, NY, USA: ACM, 2009, pp. 261–276.

[21] M. Mao and M. Humphrey, "Auto-Scaling to Minimize Cost and Meet ApplicationDeadlines in Cloud Workflows," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*, 2011, pp. 49:1–49:12

[22] X. Meng and et al., "Efficient resource provisioning in compute clouds via vm multiplexing," in *Proceeding of the 7th international conference on Autonomic computing (ICAC'10)*, New York, NY, USA: ACM, 2010, pp. 11–20.

[23] H. A. L. Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan, "SnowFlock: rapid virtual machine cloning for cloud computing," in *Proceedings of the 4th ACM European conference on Computer systems (EuroSys'09)*, New York, NY, USA: ACM, 2009, pp. 1–12.

[24] S.-i. Kuribayashi, "Optimal joint multiple resource allocation method for cloud computing environments," *International Journal of Research and Reviews in Computer Science (IJRRCS)*, vol. 2, pp. 1–8, 2011.

[25] M. Stillwell, F. Vivien and H. Casanova, "Virtual Machine Resource Allocation for Service Hosting on Heterogeneous Distributed Platforms," in *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS'12)*, Shanghai, China, pp. 786–797.