Optimization of Composite Cloud Service Processing with Virtual Machines

Sheng Di, *Member, IEEE,* Derrick Kondo, *Member, IEEE* and Cho-Li Wang, *Member, IEEE*

Abstract—By leveraging virtual machine (VM) technology, we optimize cloud system performance based on refined resource allocation, in processing user requests with composite services. Our contribution is three-fold. (1) We devise a VM resource allocation scheme with a minimized processing overhead for task execution. (2) We comprehensively investigate the best-suited task scheduling policy with different design parameters. (3) We also explore the best-suited resource sharing scheme with *adjusted* divisible resource fractions on running tasks in terms of Proportional-Share Model (PSM), which can be split into absolute mode (called AAPSM) and relative mode (RAPSM). We implement a prototype system over a cluster environment deployed with 56 real VM instances, and summarized valuable experience from our evaluation. As the system runs in short supply, Lightest Workload First (LWF) is mostly recommended because it can minimize the overall response extension ratio (RER) for both sequential-mode tasks and parallel-mode tasks. In a competitive situation with over-commitment of resources, the best one is combining LWF with both AAPSM and RAPSM. It outperforms other solutions in the competitive situation, by 16+% w.r.t. the worst-case response time and by 7.4+% w.r.t. the fairness.

Keywords—Cloud Resource Allocation, Task Scheduling, Resource Allocation, Virtual Machine, Minimization of Overhead

1 Introduction

Cloud computing [1], [2] has emerged as a flexible platform allowing users to customize their on-demand services. Platform as a Service (PaaS) is a classical paradigm, and a typical example is Google App Engine [3], which allows 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 submit complex requests each being composed of off-the-shelf web services. Each service is associated with a price, which is assigned by its creator. When a user submits a compute *request* (or a *task*) 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, virtual machine (VM) resource isolation technology [4], [5], [6], [7], [8], [9] can effectively isolate various types of resources for the VMs running on the same hardware. We leverage such a feature to refine the resource allocation, which is completely transparent to users.

In cloud systems [10], over-commitment of physical resources is fairly common in order to achieve high resource utilization. According to a Google trace [11] with 10k+ hosts, for example, Reiss. et al. [12] presented the resource amounts requested are often greater than the total capacity of Google data centers, and the requesting amounts are usually twice as the real resource amounts consumed by tasks. Such an over-commitment of resources may result in relatively short-supply situation (a.k.a., competitive situation) occasionally, degrading the overall Quality of Service (QoS) [11].

In our cloud model, each *user request* (or *task*) is made up of a set of subtasks (or web service instances), and in this paper, we aim to answer four questions below.

- how to optimize resource allocation for a task based on its budget, where the subtasks inside the task can be connected in parallel or in series.
- how to split the physical resources according to tasks' various requirements in both competitive and non-competitive situation.
- how to minimize data transmission overhead and operation cost of virtual machine monitor (VMM).
- how to schedule user requests with minimized task response time in a competitive situation.

Based on our characterization of Google trace [11], [13] which contains 4,000 types of cloud applications, we find that there are only two types of Google tasks, sequential-mode task and parallel-mode task. The former contains multiple subtasks connected sequentially (like a sequential workflow) and the latter executes multiple subtasks in parallel (e.g., mapreduce). We try to optimize the performance for both of the two cases.

The cloud system may experience two different situations, either non-competitive status or competitive status.

1) For a non-competitive situation, the available resources are relatively adequate for user demands, so the optimality is mainly determined by task's intrinsic structure (e.g., how its subtasks are connected) and budget. In particular, some subtask's output needs to be transferred to its succeeding subtask as input, and the data transmission delay cannot be overlooked if the data size is huge. On

- *S. Di is with Argonne National Laboratory, USA, D. Kondo is with INRIA, Grenoble, France, and C.L. Wang is with Department of Computer Science, The University of Hong Kong, Hong Kong, China*

the other hand, since we will take advantage of the VM resource isolation, the cost of VMM operations (such as the time cost in performing CPU-capacity changing command at runtime) is also supposed to be minimized.

2) For a competitive situation, how to keep each task's QoS at a high level and maximize the overall fairness of the treatment meanwhile is quite challenging. On one hand, 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. On the other hand, a competitive situation with limited available resources may easily delay some particular responses, leading to the unfairness of treatment.

In our experiment, we find that assigning different priorities to tasks in the task scheduling stage and in the resource allocation stage would bring out significantly different effects on the overall performance and fairness. Hence, we investigate the best-suited queuing policies for maximizing the overall performance and fairness of QoS. As for the sequential-mode tasks, 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) [14]. LWF and SSTF can be considered Shortest Job First (SJF) and min-min algorithm [15] respectively. The intuitive idea of SPF is similar to Earliest Deadline First (EDF) [16], wherein we adopt two criteria to evaluate the task execution progress. In addition, we also investigate the best-suited scheduling policy for the parallel-mode tasks. The candidate scheduling policies include FCFS, SSTF, LWF, Longest SubTask First (LSTF) and the hybrid approach with a mixture of queuing policies, e.g., LWF+LSTF.

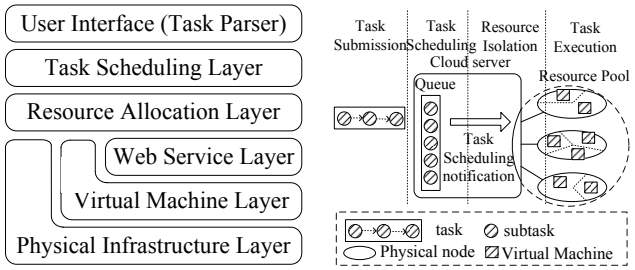
We also explore a best-fit resource allocation scheme (called adjusted proportional-share model) to adapt to the competitive situation. Specifically, we investigate how to coordinate the divisible resource allocation among the running tasks in terms of their structures like workload or varied estimated progress.

Based on the composite cloud service model, we implement a distributed prototype that is able to solve/calculate complex matrix problems submitted by users. We also explore the best choice of the involved parameters used in our algorithm, by running our experiments performed on a real-cluster environment with 56 VMs and 10 services with various execution types. Experiments show that for the sequential-mode tasks, the worst-case performance under LWF is higher than that under other policies by at least 16% when overall resource amount requested is about twice as the real resource amount that can be allocated. Another key lesson we learned is that in a competitive situation, short tasks (with the short single-core execution length) are better to be assigned with more powerful resource amounts than the theoretically optimal values derived from the optimization theory. As for the parallel-mode tasks, LWF+LSTF leads to the best result, which is better than other solutions by 3.8% - 51.6%.

In the remainder of the paper, we will use the term *host*, *machine*, and *node* interchangeably. In Section 2, we describe the architecture of our cloud system. In Section 3, 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 4, 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 5. We discuss the related works in Section 6. Finally, we conclude the paper with a vision of the future work in Section 7.

2 System Overview

The system architecture of our composite cloud service system is shown in Fig. 1 (a). A user request (a.k.a., a task) is made up of multiple subtasks connected in parallel or sequentially. Each subtask is an instance of an off-the-shelf service that has a very convenient interface (such API) to be called. Each whole task is expected to be completed as soon as possible under the constraint of its budget. Task scheduling is a key layer to coordinate the task priorities. Resource allocation layer is responsible for calculating the optimal resource fraction for the subtasks. 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 do the computation). Each subtask will be executed on a VM, with an amount of virtual resource fraction tuned by the substrate VM monitor (VMM, a.k.a., hypervisor). Fault tolerance is beyond the scope of the paper, and we discuss this issue in [17] in details.



(a) System Architecture (b) Task Processing Procedure

Fig. 1: System Overview of Cloud Composite Service System

Each task is processed via a scheduling queue, as shown in Fig. 1 (b). Tasks are submitted continually over time, and each task submitted will be analyzed by a *task parser* (in the user interface module), in order to predict the subtask workloads based on input parameters. Subtask's workload can be characterized using

{resource_processing_rate×subtask_execution_length} based on historical traces or workload prediction approaches like polynomial regression method [18]. Our optimization model will compute the optimal resource vector of all subtasks for the task. And then, the unprocessed subtasks with no dependent preceding unprocessed subtasks will be put/registered in a queue, waiting for the scheduling notification. Upon being notified, the hypervisor of the selected physical machine will launch a VM and perform the resource isolation to match optimization demand. The corresponding service on the VM will be called using specific 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 [19] to perform the resource isolation among VMs on the same physical machine. With XEN [20], we can dynamically isolate some key resources (like CPU rate and network bandwidth) to suit the specific usage demands of different tasks. 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 tuned at runtime.

3 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 is made up of multiple subtasks connected in series or in parallel. 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 formulation is generic enough such that any user request (or task) can be constructed by multiple nested composite services (or subtasks).

Task execution time is represented in different ways based on different intra-structure about subtask connection. For the sequential-mode task, 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 $t_{i(j)}$ and the compute resource allocated respectively. The workload here is evaluated by the number of instructions or data to read/write from/to disk, and the compute resource here means workload processing rate like CPU rate and disk I/O bandwidth. As for a parallel mode task (e.g., embarrassingly parallel application), its total execution length is equal to the longest execution time of its subtasks (or makespan). We will use execution time, execution length, response length, and wall-clock time interchangeably in the following text.

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 [21]). 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).

$$\min\ T(t_i) = \begin{cases} \sum_{j=1}^{m_i} \frac{l_{i(j)}}{r_{i(j)}}, & t_i\, is\, in\, sequential\, mode \\ \max\limits_{j=1\cdots m_i} \{\frac{l_{i(j)}}{r_{i(j)}}\}, & t_i\, is\, in\, parallel\, mode \end{cases} \quad (1)$$

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

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

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

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 in reality. A sequential-mode task's *theoretically optimal length* (TOL) is the sum of the theoretical execution time of each subtask based on the optimal resource allocation solution to the above problem (Formula (1) & Formula (2)), while a parallel-mode task's TOL is equal to the largest theoretical subtask execution time. The *response time* here indicates the whole wall-clock time from a task's submission to its final completion. In general, the response time of a task includes subtask's waiting time, overhead before subtask execution (e.g., on resource allocation or data transmission), subtask's productive time, and processing overhead after execution. We try best to minimize the cost for each part.

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 their connection way), task budget, and varied resource availability over time.

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

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

4 Optimization of System Performance

In order to optimize the entire QoS for each task, we need to minimize the time cost at each step in the course of its execution. We study the best-fit solution with respect to three following facets, resource allocation, task scheduling, and minimization of overheads.

4.1 Optimized Resource Allocation with VMs

We first derive an optimal resource vector for each task (including parallel-mode task and sequential-mode task), subject to task structure and budget, in both non-competitive situation and competitive situation. In non-competitive situation, there are always available and adequate resources for task processing. As for an over-committed situation (or competitive situation), the overall resources are over-committed such that the requested resource amounts succeed the de-facto resource amounts in the system. In this situation, we designed an adjustable resource allocation method for maintaining the high performance and fairness.

4.1.1 Optimal Resource Allocation in Non-competitive Situation

In a non-competitive situation (with unlimited available resource amounts), the resource fraction allocated to some task is mainly restricted by its user-set budget. Based on the target function (Formula (1)) and a constraint (Formula (2)), we analyze the two types of tasks (sequential-mode and parallel-mode) respectively.

- *Optimization of Sequential-mode Task*:

  Theorem 1: If task t_i is constructed in sequential mode, t_i's optimal resource vector $\boldsymbol{r}^*(t_i)$ for minimizing $T(t_i)$ subject to the constraint (2) is shown as Equation (5), where j=1, 2, $\cdots$, m_i.

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

  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 Lagrangian multiplier.

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

  We derive Equation (7) via Lagrangian multiplier method.

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

  In order to minimize $T(t_i)$, the optimal resource vector $\boldsymbol{r}^*_{i(j)}$ should use up all budgets (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. In this situation, our resource allocator will allocate the theoretically optimal resource fraction (Formula (5)) to each subtask's resource capacity (such as maximum CPU rate).

- *Optimization of Parallel-mode Task*:

  Theorem 2: If task t_i is constructed in the parallel mode, t_i's optimal resource vector $\boldsymbol{r}^*(t_i)$ for minimizing $T(t_i)$ subject to the constraint (2) is shown as Equation (8), where j=1, 2, $\cdots$, m_i.

$$r^*_{i(j)} = \frac{l_{i(j)}}{\sum_{j=1}^{m_i} p_{i(j)}l_{i(j)}} \cdot B(t_i) \tag{8}$$

  Proof: We just need to prove the optimal situation occurs if and only if all of subtask execution lengths are equal to each other. That is, the entire execution length of a parallel-mode task will be minimized if and only if Equation (9) holds.

$$\frac{l_{i(1)}}{r_{i(1)}} = \frac{l_{i(2)}}{r_{i(2)}} = \cdots = \frac{l_{i(m_i)}}{r_{i(m_i)}} \tag{9}$$

  In this situation, we can easily derive Equation (8) by using up the user-preset budget $B(t_i)$, i.e., letting $\sum_{j=1}^{m_i}[r_{i(j)} \cdot p_{i(j)}] = B(t_i)$ hold.

  As follows, we use proof-by-contradiction method to prove that Equation (9) is a necessary condition of the optimal situation by contradiction. Let us suppose an optimal situation with minimized task wall-clock length occurs while Equation (9) does not hold. Without loss of generality, we denote by $t_{i(k)}$ the subtask that has the longest execution time (i.e., $\frac{l_{i(k)}}{r_{i(k)}}$), that is, $T(t_i)$=$\frac{l_{i(k)}}{r_{i(k)}}$. Since Equation (9) does not hold, there must exist another subtask $t_{i(j)}$ such that $\frac{l_{i(j)}}{r_{i(j)}}$<$\frac{l_{i(k)}}{r_{i(k)}}$. Obviously, we are able to add a small increment $\triangle_k$ to $r_{i(k)}$ and decrease $r_{i(j)}$ by $\triangle_j$ correspondingly, such that the total payment is unchanged and the two subtasks' wall-clock lengths become the same. That is, Equation (10) and Equation (11) hold simultaneously.

$$\begin{aligned} &r_{i(j)}p_{i(j)} + r_{i(j)}p_{i(k)} \\ &= (r_{i(j)} - \Delta_j)p_{i(j)} + (r_{i(j)} + \Delta_k)p_{i(k)} \end{aligned} \tag{10}$$

$$\frac{l_{i(j)}}{r_{i(j)} - \Delta_j} = \frac{l_{i(k)}}{r_{i(k)} + \Delta_k} \tag{11}$$

  It is obvious that the new solution with Δ_j and Δ_k gets the further reduced task wall-clock length, which contradicts to our assumption that the previous allocation is optimal. □

4.1.2 Adjusted Resource Allocation to Adapt to Competitive Situation

If the system runs in short supply, it is likely the total sum of their optimal resources (i.e., $r^*(t_i)$) may succeed 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 model (PSM) [22] with XEN's credit scheduler by further enhancing resource fractions for short tasks.

Under XEN's credit scheduler, each guest VM on the same physical machine will get its CPU rate that is proportional to its weight[1]. 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 fraction (denoted by $r(v_j)$) allocated to the VM v_j can be calculated by Formula (12).

$$r(v_j) = \frac{w(v_j)}{\sum_{k=1}^{n_i} w(v_k)} c_i \tag{12}$$

Now, the key question becomes how to determine the weight value for each running subtask (or VM) on a physical machine, to adapt to the competitive situation. We devise a novel model, namely *Adjusted Proportional-Share Model (APSM)*, which further tunes the credits based on task's workload (or execution length). The design of APSM is based on the definition of RER: a large value of RER tends to appear with a short task. 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 RER's definition, short task's RER is more sensitive to the execution overheads than that of a long one. Hence, we make short tasks tend to get more resource fractions than their theoretically optimal vector (i.e., $r^*_{i(j)}$). There are two alternative ways to realize this effect.

- *Absolute Mode*: For this mode, we use a threshold (denoted as τ) to split running tasks into two categories, short tasks (workload$\leq\tau$) and long tasks (workload$>\tau$). Three values of τ are investigated in our experiments: 500, 1000, or 2000, which corresponds to 5 seconds, 10 seconds or 20 seconds when running a task on a single core. We assign as much resource as possible to short tasks, while keeping the long tasks' resource fractions unchanged. Task length is evaluated in terms of its workload to process. In practice, it can be estimated based on the workload characterization over history or workload prediction method like [18]. In our design based on the absolute mode, short tasks' credits will be set to 800 (i.e., 8 cores), implying the full computational power. For example, if there is only one short running task on a host, it will be assigned with full resources (8 cores) for its computation. If there are more running tasks, they will be allocated according to PSM, while short tasks will be probably assigned with more resource fractions.
- *Relative Mode*: Our intuitive idea is adopting a proportional-share model (PSM) 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 either Formula (13) or Formula (14), based on different proportional-share credits (either task's workload or task's TOL). Hence, the Relative Mode based APSM (abbreviated as RAPSM) has two different types, Workload-based APSM (abbreviated as RAPSM(W)) and TOL-based APSM (abbreviated as RAPSM(T)).

$$w(t_{i(j)}) = \begin{cases} \eta \cdot r^*_{i(j)} & l_i \leq \alpha \\ r^*_{i(j)} & \alpha < l_i \leq \beta \\ \frac{1}{\eta} \cdot r^*_{i(j)} & l_i > \beta \end{cases} \tag{13}$$

$$w(t_{i(j)}) = \begin{cases} \eta \cdot r^*_{i(j)} & T(t_i) \leq \alpha \\ r^*_{i(j)} & \alpha < T(t_i) \leq \beta \\ \frac{1}{\eta} \cdot r^*_{i(j)} & T(t_i) > \beta \end{cases} \tag{14}$$

The weight values in our design (Formula (13)) 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 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.

In practice, one could use either of the above two modes or both of them, to adjust the resource allocation to adapt to the competitive situation.

4.2 Best-suited Task Scheduling Policy

In a competitive situation where over-many tasks are submitted to the system, it is necessary to queue some

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

tasks that cannot find the qualified resources temporarily. The queue will be checked as soon as some new resources are released or new tasks are submitted. 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 the two-fold objective that aims to minimize the RER and maximize the fairness meanwhile, we investigate the best-fit scheduling policy for both sequential-mode tasks and parallel-mode tasks. We propose that (1) the best-fit queuing policy for the sequential-mode tasks is Lightest-Workload-First (LWF) policy, which assigns the highest scheduling priority to the shortest task that has the least workload amount to process; (2) the best-fit policy for parallel-mode tasks is adopting LWF and Longest-SubTask-First (LSTF) together. 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), Shortest-SubTask-First (SSTF), and so on. 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 based 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 because of usually stable memory size consumed over time. 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, for minimizing the discrepancy among tasks' RERs.
- *Slowest-Progress-First (SPF)*. SPF is designed for sequential-mode tasks, based on 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 (15) and Formula (16) 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.

$$WP(t_i) = \frac{\sum_{j=1}^{d} l_{i(d)}}{l_i} \tag{15}$$

$$TP(t_i) = \frac{wall\text{-}clock\ time\ since\ t_i's\ submission}{TOL(t_i)} \tag{16}$$

  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 (SSTF)*. SSTF 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, SSTF 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 SSTF will make the short subtasks be executed as soon as possible. Hence, this policy is

exactly the same as *min-min* policy [15], which has been effective in Grid workflow scheduling. However, our experiments validate that SSTF is not the best-suited scheduling policy in our Cloud system.

- *LWF+LSTF*. We can also combine different individual policies to generate a new scheduling policy. In our system, LWF+LSTF is devised for parallel-mode task, whose total execution length is determined by its longest subtask execution length (i.e., makespan), thus the subtasks with heavier workloads in the same task will have higher priority to schedule. On the other hand, in order to minimize the overall waiting time, all of tasks will be scheduled based on Lightest Workload First (LWF). LWF+LSTF means that the subtasks whose task has the lightest workload will have the highest priority and the subtasks belonging to the same task will be scheduled based on Longest SubTask First (LSTF). In addition, we also implement LWF+SSTF for comparison.

4.3 Minimization of Processing Overheads

In our system, in addition to the waiting time and execution time of subtasks, there are three more overheads which need also to be counted in the whole response time, VM resource isolation cost, data transmission cost between sub-tasks, and VM's default restoring cost. Our cost-minimization strategy is performing the data transmission and VMM operations concurrently, based on the characterization of their costs. We also assign extra amount of resources to super-short tasks (e.g., the tasks with TOL$\leq$2 seconds) in order to mitigate the impact of the overhead to their executions. Specifically, 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. Details can be found in our corresponding conference paper [24].

5 Performance Evaluation

5.1 Experimental Setting

We implement a cloud composite service prototype that can help solve complex matrix problems, each of which is allowed to contain a series of nested or parallel matrix computations. For an example of nested matrix computation, a user may submit a request like *Solve*($(A_{m\times n}\cdot A_{n\times m})^k$,$B_{m\times m}$), which can be split into three steps (or subtasks): (1) matrix-product (a.k.a., matrix-matrix multiply): $C_{m\times m}$=$A_{m\times n}\cdot A_{n\times m}$; (2) matrix-power: $D_{m\times m} = C^k_{m\times m}$; (3) calculating Least squares solution of $D\cdot X$=B: *Solve*($D_{m\times m}$, $B_{m\times m}$).

In our experiment, we make use of *ParallelColt* [25] to perform the math computations, each consisting of a set of matrix computations. ParallelColt [25] is such a library that can effectively calculate complex matrix computations, such as matrix-matrix multiply and matrix decomposition, in parallel (with multiple threads) based on Symmetric Multiple Processor (SMP) model.

There are totally 10 different matrix computations (such as matrix-product, matrix-decomposition, etc.) as shown in Table 1. We carefully characterize the single-core execution length (or workload) for each of them, and find that each matrix computation has its own execution type. For example, matrix-product and matrix-power are typical computation-intensive services, while rank and two-norm computation should be memory-intensive or I/O-bound ones when matrix scale is large. Hence, each sequential-mode task that is made up of multiple different matrix computations in series can be considered complex applications with execution types varied over time.

In each test, we randomly generate a number of user requests, each of which is composed of 5$\sim$15 subtasks (or matrix computation services). Such a simulation is non-trivial since each emulated matrix has to be compatible for each matrix computation (e.g., two matrices in a matrix-product must be in the form of $A_{m\times n}$ and $B_{n\times p}$ respectively). Among the 10 matrix-computation services, three services are implemented as multiple-threaded programs, 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, or less when its theoretically optimal resource to allocate is less than one core.

In our experiment, we are assigned with 8 physical nodes to use from the most powerful cluster at HongKong (namely Gideon-II [23]), 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 [20] serves as the hypervisor on each node and dynamically allocates various CPU rates to the VMs at run-time using the credit scheduler.

We will evaluate different queuing policies and resource allocation schemes under different competitive situations with different numbers (4-24) of tasks simultaneously. Table 2 lists the candidate key parameters we investigated in our evaluation. Note that the measurement unit of η and β for RAPSM(T) is second, while the measurement unit for RAPSM(W) is seconds$\times$100, because a single core's processing ability is represented as 100 according to XEN's credit scheduler [20], [19].

5.2 Experimental Results

5.2.1 Demonstration of Resource Contention Degrees

We first characterize the various contention degrees with different number of tasks submitted. The contention

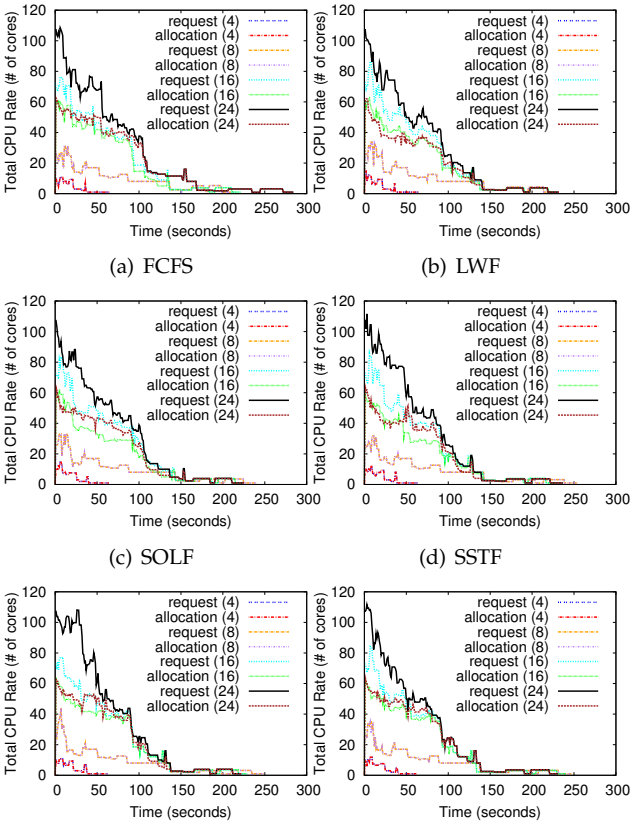
TABLE 1: Workloads (Single-core Execution Length) of 10 Matrix Computations (seconds)

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

TABLE 2: Candidate Key Parameters

Parameter	Value
threshold of short task length (seconds)	5, 10, 20
η	1.25, 1.5, 1.75, 2
α w.r.t. RAPSM(T) (seconds)	5, 10, 20
β w.r.t. RAPSM(T) (seconds)	100, 200, 300
α w.r.t. RAPSM(W) (seconds$\times$100)	500, 1000, 2000
β w.r.t. RAPSM(W) (seconds$\times$100)	10000, 20000, 30000

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.



(a) FCFS (b) LWF

(c) SOLF (d) SSTF

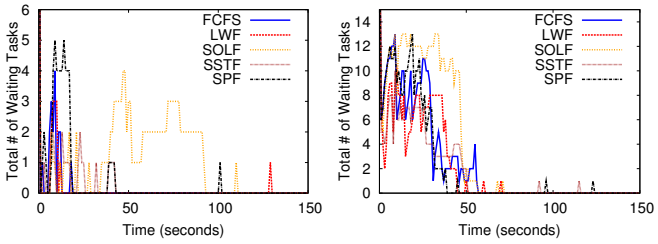
(e) SWPF (f) STPF

Fig. 2: Allocation vs. Request With Different Contention Degrees

Fig. 2 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.

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 Cloud system.

Fig. 3 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 small number of waiting tasks (about 5-6 and 6-7 respectively). By contrast, under SOLF, SWPF, or STPF, QL is much longer (about 10-12 waiting tasks on average), implying a longer waiting time.



(a) # of tasks = 16 (b) # of tasks = 24

Fig. 3: Queue Lengths With Different Scheduling Policies

5.2.2 *Investigation of Best-suited Strategy*

We explore the best-suited scheduling policy and resource allocation scheme, in a competitive situation with 24 tasks (Allocate-Request Ratio (AAR) $\approx \frac{1}{2}$ for the first 50 seconds). The investigation is for sequential-mode tasks and parallel-mode tasks respectively.

A. Best-suited Strategy for Sequential-mode Tasks

Fig. 4 shows the distribution (Cumulative Distribution Function) of Response Extension Ratio (RER) in the competitive situation, with different scheduling policies used for the sequential-mode tasks. For each policy, we

ran the experiments with all the possible combinations of parameters shown in Table 2, and then compute the distribution. It is clearly observed that the RERs under LWF and SSTF are much smaller than those under other policies. By contrast, The two worst policies are SWPF and SOLF, whose maximum RERs are even up to 105 and 55 respectively. The main reason is that LWF and SSTF lead to much shorter waiting time than SWPF and SOLF, according to Fig. 3.

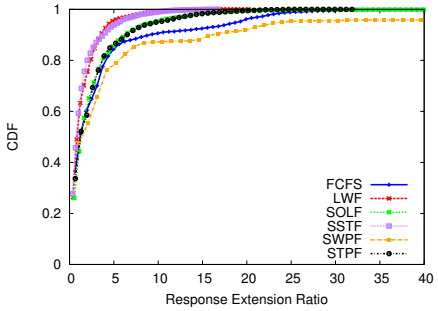


Fig. 4: Distribution of RER in a Competitive Situation

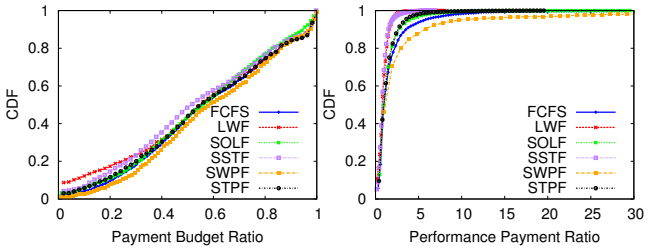
In addition to task scheduling policy, we also investigate the best-fit resource allocation scheme. In Table 3, we show the statistics of RER with various solutions, by combining different scheduling policies and resource allocation schemes. We evaluate each solution with all of different combinations of parameters (including τ, η, α, and β) and compute the statistics (including minimum, average, maximum, and fairness value).

TABLE 3: Statistics of RER in a Competitive Situation with Sequential-mode Tasks

strategy	min.	avg.	max.	fairness
FCFS+PSM	0.712	**3.919**	**22.706**	**0.352**
FCFS+RAPSM(T)	0.718	4.042	23.763	0.351
FCFS+RAPSM(W)	0.720	4.137	24.717	0.348
LWF+PSM	0.720	**2.106**	8.202	0.628
LWF+RAPSM(T)	0.719	2.172	8.659	0.603
LWF+RAPSM(W)	0.723	**2.122**	**7.937**	**0.630**
SOLF+PSM	0.736	2.979	13.473	0.506
SOLF+RAPSM(T)	0.745	3.252	14.625	0.527
SOLF+RAPSM(W)	0.738	3.230	14.380	0.526
SSTF+PSM	0.791	**2.068**	**8.263**	**0.591**
SSTF+RAPSM(T)	0.769	2.169	9.024	0.566
SSTF+RAPSM(W)	0.770	2.126	8.768	0.579
SWPF+PSM	0.713	6.167	58.691	0.209
SWPF+RAPSM(T)	0.726	6.532	62.332	0.208
SWPF+RAPSM(W)	0.718	6.477	61.794	0.208
STPF+PSM	0.723	3.176	16.398	0.465
STPF+RAPSM(T)	0.723	3.208	15.831	0.475
STPF+RAPSM(W)	0.722	3.188	15.399	0.485

Through Table 3, we observe that LWF and SSTF result in the lowest RERs on average and at the worst case, which is consistent with the distribution of RER as shown in Fig. 4. They improve the performance by $\frac{3.919}{2.1} - 1$=86.6%, as compared to First-Come-First-Serve (FCFS) policy. It is also observed that Relative Mode based Adjusted PSM (RAPSM) may not further reduce the RER as expected. This means that RAPSM cannot directly improve the execution performance without Absolute Mode based Adjusted PSM (AAPSM). Later, we will show in Section 5.2.3 that the solutions with Absolute Mode based Adjusted PSM (AAPSM) can significantly reduce RER, in comparison to the RAPSM.

We also show the distributions of Payment Budget Ratio (PBR) and Performance Payment Ratio (PPR) in Fig. 5. Through Fig. 5 (a), we observe that all of tasks' payments are guaranteed below their budgets. This is due to the strict payment-by-reserve model (Formula (2) and Theorem 1) we always followed in our design. Through Fig. 5 (b), it is also observed that PPR exhibits similarly to RER. For example, two best scheduling policies are also LWF and SSTF. Their mean PPRs are 0.874 and 0.883 respectively; their maximum PPRs are 8.176 and 6.92 respectively. Apparently, if we do not take into account the difference of adjusted resource allocation scheme but task scheduling policy, SSTF outperforms others prominently due to its shortest waiting time on average.



(a) Payment Budget Ratio (b) Performance Payment Ratio

Fig. 5: Distribution of PBR and PPR in a Competitive Situation

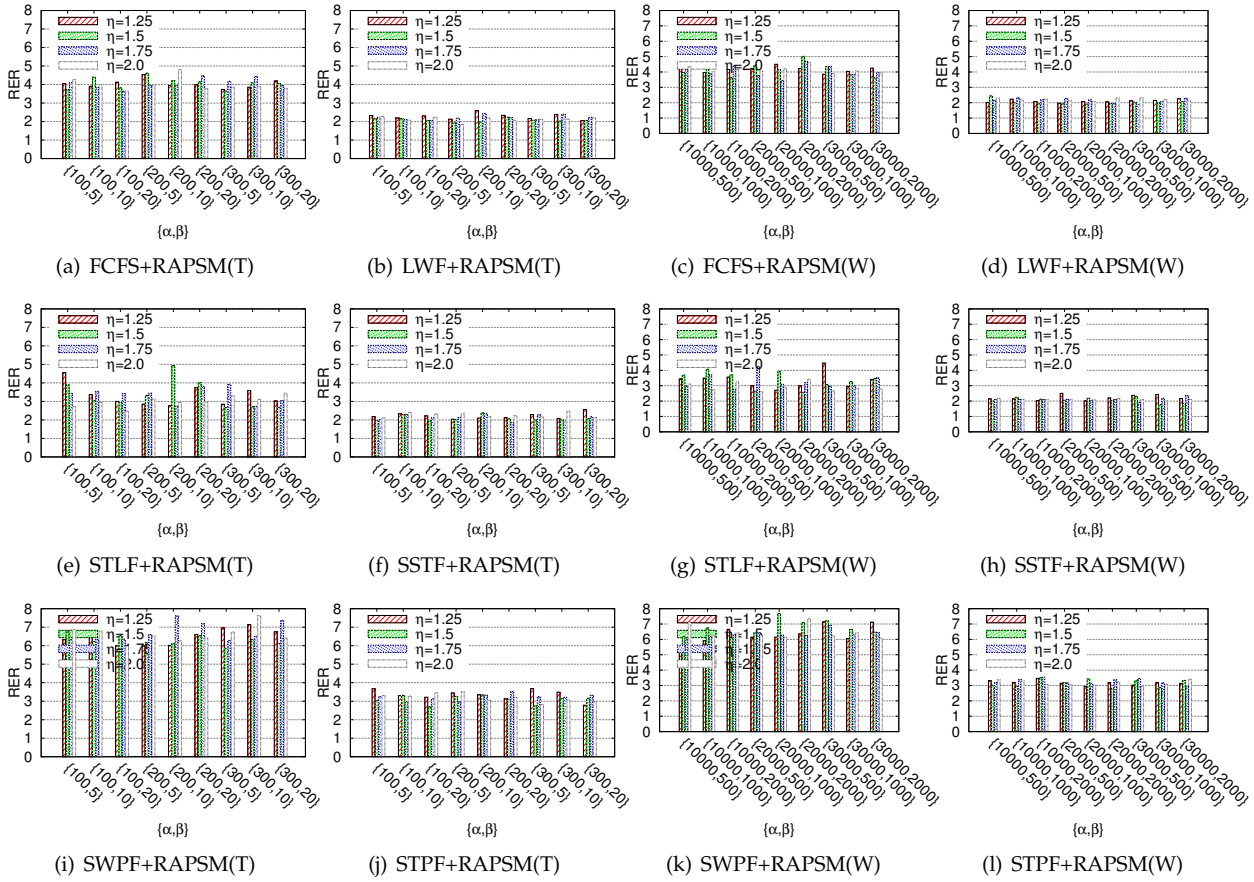
B. Best-suited Strategy for Parallel-mode Tasks

We also explore the best-suited scheduling strategy with respect to the parallel-mode tasks. Table 4 presents different minimum/average/maximum/fairness values of RER when scheduling parallel-mode tasks by different policies, including FCFS, LWF, SSTF, etc. Note that SPF is not included because all of subtasks in a task will be executed in parallel, so that it is meaningless to characterize the processing progress, unlike the situation with sequential-mode tasks.

Each test is conducted with 24 tasks, and each task randomly contains 5-15 parallel matrix-power computation subtasks. Since there are only 8 physical execution nodes, so this is a competitive situation where some tasks have to be queued for scheduling. In this experiment, we also implement Heaviest Workload First (HWF) combined with Longest (Shortest) SubTask First for comparison.

Based on Table 4, we can observe that LWF always leads to a fairly high scheduling performance. For example, when only adopting LWF, the average RER is about 1.30, which is lower than that of FCFS by $\frac{1.5-1.3}{1.5}$=13.3%, and lower than SSTF by $\frac{2.09-1.3}{2.09}$=37.8%. Adopting the LWF+LSTF (i.e., the combination of LWF and SSTF) can minimize the maximum RER to be 3.58, which is lower than other strategies by 3.8% (LWF) - 51.6% (HWF+SSTF).

The key reason why LWF exhibits much better results is that LWF schedules the shortest tasks with highest priority, suffersing the least overall waiting time. In



(a) FCFS+RAPSM(T) (b) LWF+RAPSM(T) (c) FCFS+RAPSM(W) (d) LWF+RAPSM(W)

(e) STLF+RAPSM(T) (f) SSTF+RAPSM(T) (g) STLF+RAPSM(W) (h) SSTF+RAPSM(W)

(i) SWPF+RAPSM(T) (j) STPF+RAPSM(T) (k) SWPF+RAPSM(W) (l) STPF+RAPSM(W)

Fig. 6: Average RER of Various Solutions with Different Parameters

TABLE 4: Statistics of RER in a Competitive Situation with Parallel-mode Tasks

strategy	min.	avg.	max.	fairness
FCFS	2.75	1.50	4.97	0.89
LWF	2.25	1.30	3.72	0.92
HWF	3.04	1.45	6.25	0.86
SSTF	3.37	2.09	4.94	0.96
LWF+LSTF	**2.34**	**1.57**	**3.58**	**0.96**
LWF+SSTF	2.27	1.35	3.79	0.94
HWF+LSTF	3.20	1.43	6.88	0.81
HWF+SSTF	3.20	1.38	7.39	0.82

particular, not only can LWF+LSTF minimize the maximum ERE, but it can also lead to the highest fairness (up to 0.96), which means a fairly high stability of the LWF+LSTF policy. This is because each task is a parallel-mode task, such that Longest SubTask First (SSTF) can effectively minimize the makespan for each task, optimizing the execution performance for a particular task.

5.2.3 Exploration of Best-fit Parameters

In this section, we comprehensively compare various solutions with different scheduling policies and adjusted resource allocation schemes with different parameters shown in Table 2, in a both competitive situation (i.e., AAR is about 1/2) and non-competitive situation (i.e., AAR approaches 1). We find that the adjusted resource allocation scheme could effectively improve the execution performance (RER and PPR), only when combining it with the Absolute Mode based Adjusted PSM (AAPSM).

A. Evaluation in a Competitive Situation

Fig. 6 shows the RER of various solutions with different parameters, including η, α, and β. It is observed that various policies with different choices of the three parameters lead to different results. The smallest RER (best result) is 1.77, when adopting SSTF+RAPSM(W) and η, α, and β being set to 1.75, 30000, and 1000. The largest RER (worst case) is 7.69, when adopting SWPF+RAPSM(W) and η, α, and β being set to 1.5, 20000, and 1000. We also find that different selections of the three parameters may affect the performance prominently for a few solutions like STLF+RAPSM(T) and STLF+RAPSM(W). However, they would not impact RER clearly in most cases. From Fig. 6 (b), (d), (f) and (h), it is observed that with different parameters, the RERs under both LWF and SSTF are within [1.85, 2.31].

In our experiments, the most interesting and valuable finding is that the Absolute Mode based Adjusted PSM (AAPSM) with different short task length thresholds (τ) will result in quite different results, which we show in Table 5, Table 6 and Table 7. These three tables present the three key indicators, average RER, maximum RER, and fairness index of RER, when adopting various solutions with different values of τ and η. In our evaluation, we compute the average value for each of the three indicators, by traversing all of the remaining parameters, including α and β. Accordingly, the values shown in three tables can be deemed relatively stable

mathematical expectation.

Through Table 5, we clearly observe that LWF and SSTF significantly outperforms other solutions, w.r.t. the mean values of RER. The mean values of RER under the two solutions can be restricted down to 1.935 and 1.968 respectively, when short task length threshold is set to 20 seconds. The mean value of RER under FCFS is about 4, which is about twice as large as that of LWF or SSTF. The worst situation occurs when adopting SWPF+RAPSM(W) and setting τ to 20. In this situation, the mean value of RER is even up to 6.784, which is worse than LWF(τ=20) by $\frac{6.784-1.935}{1.935}$=250.6%. The reason why τ=20 is often better than τ=5 is that the former assigns more resources to short tasks at runtime, significantly reducing the waiting cost in the system. However, τ=20 is not always better than τ=5 or τ=10, in that the resource allocation is also related to other parameters like η. That is, if η is set to 2, then τ=20 will lead to an over-adjusted resource allocation situation, which exhibits worst results than τ=10.

TABLE 5: Mean RER under Various Solutions with Different τ&η

strategy	τ	η=1.25	η=1.5	η=1.75	η=2
FCFS	5	4.050	4.142	4.131	4.054
	10	4.122	3.952	3.924	3.845
	20	4.121	4.196	4.139	4.296
LWF	5	2.071	2.090	2.169	2.138
	10	2.268	2.133	2.179	2.152
	20	2.194	**1.935**	2.194	2.218
SOLF	5	3.316	3.321	3.552	3.102
	10	3.241	3.382	2.989	2.783
	20	3.375	3.324	3.305	3.039
SSTF	5	2.111	2.072	2.275	2.147
	10	2.202	2.171	**1.980**	2.172
	20	2.322	**1.968**	2.092	2.205
STPF	5	3.265	3.011	3.271	3.119
	10	3.296	3.024	3.152	3.132
	20	3.200	3.326	3.318	3.244
SWPF	5	6.169	6.371	6.339	6.322
	10	6.271	6.353	6.446	6.659
	20	6.784	6.763	6.730	6.635

Through Table 6, it is observed that that LWF and SSTF significantly outperforms other solutions, w.r.t. the maximum values of RER (i.e., the worst case for each solution). In absolute terms, the expected value of the worst RER when adopting LWF with τ=20 is about 5.539, and SSTF's is about 6.432, which is worse than LWF by $\frac{6.432}{5.539}$ − 1=16.1%. The worst case among all solutions happens when using SWPF+RAPSM(W), and the expected value of the worst RER is even up to 64.887, which is about 11.7 times as large as that of LWF (τ=20). The expected value of the worst RER under FCFS is about 23, which is about 4 times as large as that of LWF.

Table 7 shows the fairness of RER with different solutions. We find that the best result is adopting LWF with τ and η being set to 20 and 1.5, and the expected fairness value is up to 0.709, which is better than the second best solution (SSTF, τ=20, η=1.5) by about $\frac{0.709-0.66}{0.66}$=7.4%. From the three tables, we can conclude that LWF(τ=20,η=1.5) is the best choice in the competitive situation with AAR≈2.

In Fig. 7, we further investigate the performance (on minimum value, average value, and maximum value of RER) with more comprehensive combinations of parameters, by taking into account scheduling policy, AAPSM and RAPSM together. We find that LWF and SSTF result in best results when their short task length thresholds (τ) are set to 20 seconds and 10 seconds respectively. So, we just make the comparison in these two situations. It is observed that the mean values of RER of LWF+AAPSM+RAPSM(T) and LWF+AAPSM+RAPSM(W) (τ=20) can be down to 1.64 and 1.67 respectively, and their values of $\{\alpha,\beta\}$ are {300,20} and {10000,2000} respectively. In addition, the maximum value of RER under LWF+AAPSM+RAPSM(T) (τ=20) is about 3.9, which is the best result as we observed.

TABLE 6: Max. RER under Various Solutions with Different τ&η

strategy	τ	η=1.25	η=1.5	η=1.75	η=2
FCFS	5	23.392	25.733	24.742	24.470
	10	24.184	22.397	22.633	23.192
	20	24.204	25.258	24.391	25.313
LWF	5	9.164	7.826	8.543	8.323
	10	10.323	7.681	9.136	9.104
	20	8.106	**5.539**	6.962	8.812
SOLF	5	13.294	15.885	15.743	13.255
	10	12.735	18.719	13.939	10.325
	20	17.070	15.642	13.379	13.394
SSTF	5	8.091	7.931	9.276	9.803
	10	10.245	8.376	7.224	11.199
	20	11.418	**6.432**	6.706	9.650
STPF	5	16.456	15.545	17.642	15.232
	10	16.925	14.797	15.892	14.710
	20	13.978	15.596	16.250	14.853
SWPF	5	58.467	61.647	60.266	59.044
	10	59.351	60.199	61.241	64.286
	20	65.142	64.887	64.769	63.326

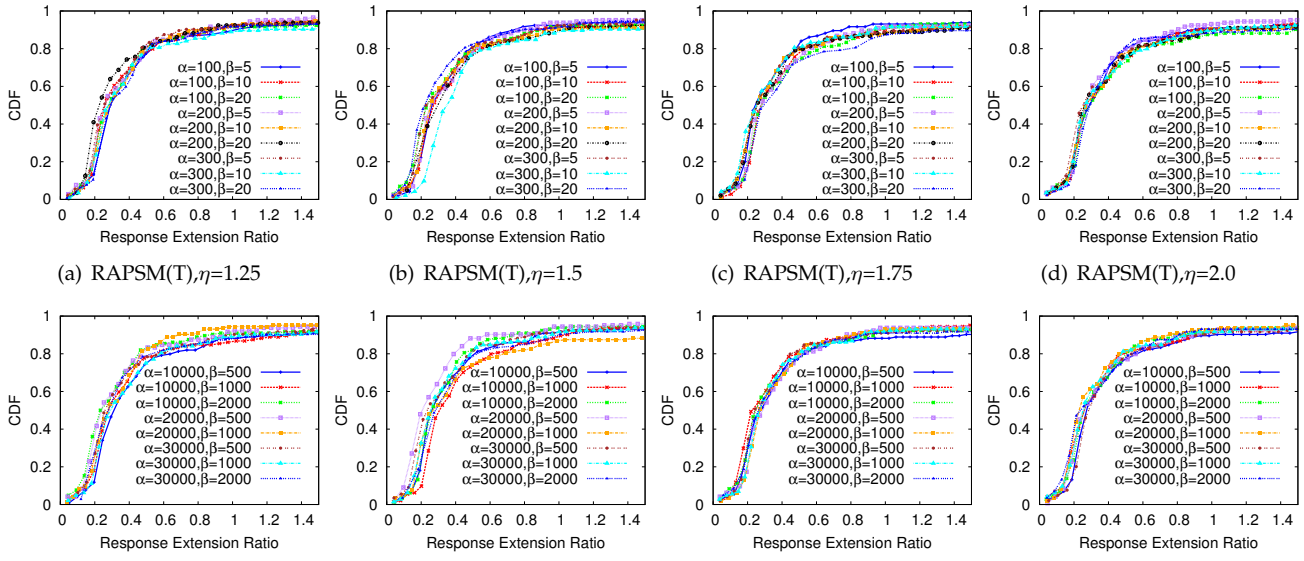
TABLE 7: Fairness of RER under Various Solutions with Different τ&η

strategy	τ	η=1.25	η=1.5	η=1.75	η=2
FCFS	5	0.353	0.338	0.345	0.352
	10	0.347	0.359	0.358	0.348
	20	0.352	0.344	0.354	0.347
LWF	5	0.585	0.628	0.609	0.619
	10	0.552	0.640	0.581	0.579
	20	0.627	**0.709**	0.670	0.610
SOLF	5	0.552	0.506	0.540	0.526
	10	0.566	0.459	0.497	0.573
	20	0.484	0.520	0.538	0.541
SSTF	5	0.594	0.602	0.558	0.549
	10	0.527	0.575	0.623	0.479
	20	0.517	**0.660**	0.650	0.544
STPF	5	0.468	0.472	0.446	0.480
	10	0.457	0.473	0.474	0.494
	20	0.508	0.500	0.479	0.499
SWPF	5	0.210	0.205	0.209	0.215
	10	0.210	0.209	0.209	0.204
	20	0.206	0.207	0.206	0.208

B. Evaluation in a Non-competitive Situation

For the non-competitive situation, there are 8 tasks submitted and AAR is about 1 in the first 50 seconds. We compare the execution performance (i.e., RER) when assigning different values to η, α, and β in a non-competitive situation. For each set of parameters, we perform 144 tests, based on various task scheduling policies and different values of threshold of short task length (i.e., τ), and then compute the CDF for each set of parameters, as shown in Fig. 8.

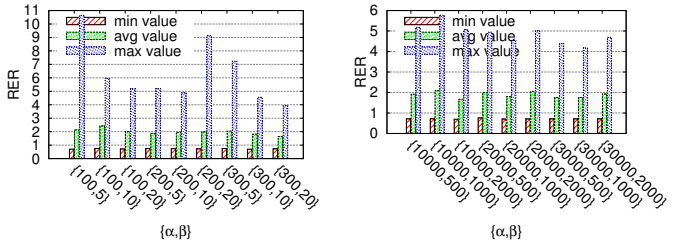
Through Fig. 8, it is observed that various assignments



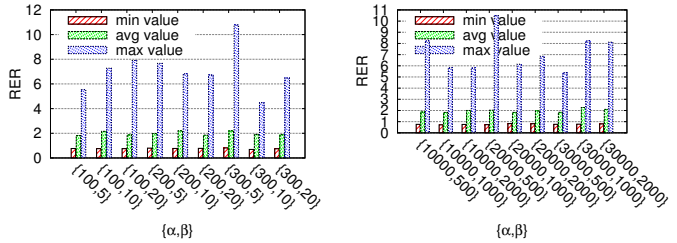
(a) RAPSM(T),η=1.25 (b) RAPSM(T),η=1.5 (c) RAPSM(T),η=1.75 (d) RAPSM(T),η=2.0

(e) RAPSM(W),η=1.25 (f) RAPSM(W),η=1.5 (g) RAPSM(W),η=1.75 (h) RAPSM(W),η=2.0

Fig. 8: Average RER with Different Parameters for the Non-competitive Situation



(a) LWF+AAPSM+RAPSM(T),τ=20 (b) LWF+AAPSM+RAPSM(W),τ=20



(c) SSTF+AAPSM+RAPSM(T),τ=10 (d) SSTF+AAPSM+RAPSM(W),τ=10

Fig. 7: Investigation of Best-fit Parameters for LWF+AAPSM

on the set of parameters result in different execution performance. For RAPSM(T) and RAPSM(W), {α=200 seconds, β=5 seconds} and {α=20000, β=500} often serve as the best assignment of the parameters respectively, regardless of η's value. For example, Fig. 8 (d) shows that when η is set to 2.0, {α=200 seconds, β=5 seconds} is better than other choices by [3.8%, 7.1%]. In addition, we can also observe a relatively high instability in the other assignments of parameters. For instance, with RAPSM(T), {α=100 seconds, β=20 seconds} exhibits good results in Fig. 8 (c) (η=1.75), but bad results in Fig. 8 (η=2.0); Using RAPSM(W) with {α=10000, β=2000}, about 93% of tasks' RERs are below 1 when setting η to 1.75, while the corresponding ratio is only 86% when setting eta to 2.0.

6 Related Work

Although job scheduling problem [26] in Grid computing [27] has been extensively studied for years, most of them (such as [28], [29]) 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 [13]. Hence, task's response time will be more easily degraded by scheduling/execution overheads (such as waiting time and data transmission cost) in Cloud environment than in Grid environment. That is, the overheads in Cloud environment should be treated more carefully.

Recently, many new scheduling methods are proposed for different Cloud systems. Zaharia et al. [30] designed a task scheduling method to improve the performance of Hadoop [31] for a heterogeneous environment (such as a pool of VMs each being customized with different 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. Isard et al. [32] 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. Mao et al. [33], [34] proposed 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. Many of other methods like Genetic algorithms [35] and Simulated Annealing algorithm [36], often overlooked the execution overheads in VM operation or data transmission, and performed the evaluation through simulation.

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. Stillwell et al. [37] 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. Meng et al. [38] analyzed VM-pairs' compatibility in terms of the forecasted workload and estimated VM sizes. SnowFlock [39] 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. Kuribayashi [40] also proposed a resource allocation method for Cloud computing environments especially based on divisible resources. BlobCR [41] aims to optimize the performance of HPC applications on Infrastructure-as-a-Service (IaaS) clouds at system level, by improving the robustness of running virtual machines using virtual disk image snapshots. In comparison, our work focuses on the theoretical optimization of performance when system runs in short supply and corresponding implementation issues at the application level.

7 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 the QoS of each user request and maximize fairness of treatment at runtime. Our contribution is three-fold: (1) we studied the best-suited task scheduling policy with VMs; (2) we explored an optimal resource allocation scheme and an adjusted strategy to suit the competitive situation; (3) the processing overhead is minimized in our design. Based on our experiments, we summarize the following lessons.

- We confirm that the best scheduling policy of scheduling sequential-mode tasks in the competitive situations, is either Lightest-Workload-First (LWF) or Shortest SubTask First (SSTF). Each of them improves the performance by about 86% compared to First-Come-First-Serve (FCFS). As for the parallel-mode tasks, the best-fit policy is combining LWF and Longest SubTask First (LSTF), and the average RER is lower than other solutions by 3.8% - 51.6%.
- For a competitive situation, the best solution is combining Lightest-Workload-First (LWF) with AAPSM and RAPSM (in absolute terms, LWF+AAPSM+RAPSM with short task length threshold and extension coefficient being set to 20 seconds and 1.5 respectively). It outperforms other solutions in the competitive situation, by 16+% w.r.t. the worst-case response time. The fairness under this solution is about 0.709, which is higher than that of the second best solution (SSTF+AAPSM+RAPSM) by 7.4+%.
- For a non-competitive situation, $\{\alpha$=200 seconds, β=5 seconds$\}$ serves as the best assignment of the parameters, regardless of the threshold value of setting the short task length (η).

In the future, We plan to further exploit an adaptive solution that can dynamically optimize the performance in both competitive and non-competitive situations. We also plan to improve the ability of fault tolerance and resilience in our cloud system.

Acknowledgments

This work was made by the ANR project Clouds@home (ANR-09-JCJC-0056-01), also supported by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357, and also in part by HKU 716712E.

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] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: towards a cloud definition," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 1, pp. 50–55, 2009.

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

[4] J. E. Smith and R. Nair, *Virtual Machines: Versatile Platforms For Systems And Processes*. Morgan Kaufmann, 2005.

[5] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat, "Enforcing performance isolation across virtual machines in xen," *Proc. seventh ACM/IFIP/USENIX International Conference on Middleware (Middleware'06)*, 2006, pp. 342–362.

[6] J. N. Matthews, W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, G. Hamilton, M. McCabe, and J. Owens, "Quantifying the performance isolation properties of virtualization systems," *Proc. ACM workshop on Experimental computer science (ExpCS'07)*, 2007.

[7] S. Chinni and R. Hiremane, "Virtual machine device queues," Virtualization Technology White Paper, Tech. Rep., 2007.

[8] T. Cucinotta, D. Giani, D. Faggioli, and F. Checconi, "Providing performance guarantees to virtual machines using real-time scheduling," *Proc. 5th ACM Workshop on Virtualization in High-Performance Cloud Computing (VHPC'10)*, 2010, pp. 657–664.

[9] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds : Managing performance interference effects for qos-aware clouds," *Proc. ACM European Conference on Comp. Sys. (EuroSys'10)*, 2010, pp. 237–250.

[10] R. Ghosh, V.K. Naik, "Biting Off Safely More Than You Can Chew: Predictive Analytics for Resource Over-Commit in IaaS Cloud," *IEEE 5th International Conference on Cloud Computing*, 2012, pp. 25–32.

[11] Google cluster-usage traces: online at http://code.google.com/p/googleclusterdata.

[12] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Towards understanding heterogeneous clouds at scale: Google trace analysis," Intel science and technology center for cloud computing, Carnegie Mellon University, Pittsburgh, PA, USA, Tech. Rep. ISTC–CC–TR–12–101, Apr. 2012.

[13] S. Di, D. Kondo, and W. Cirne, "Characterization and comparison of cloud versus grid workloads," in *Proc. of IEEE International Conference on Cluster Computing (Cluster'12)*, 2012, pp. 230–238.

[14] M. Rahman, S. Venugopal, and R. Buyya, "A dynamic critical path algorithm for scheduling scientific workflow applications on global grids," in *Proc. of 3rd IEEE International Conference on e-Science and Grid Computing*, 2007, pp. 35–42.

[15] 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 *Proc. of 8th Heterogeneous Computing Workshop (HCW '99)*, Washington, DC, USA: IEEE Computer Society, 1999, p. 30.

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

[17] S. Di, Y. Robert, F. Vivien, D. Kondo, C-L. Wang, F. Cappello, "Optimization of cloud task processing with checkpoint-restart mechanism, " in *Proc. of IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC'13)*, 2013, pp. 64:1–64:12.

[18] 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 *Proc. of 24th International Conference on Neural Information Processing Systems (NIPS'10)*. 2010, pp. 1–9.

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

[20] 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 *Proc. of 19th ACM symposium on Operating systems principles (SOSP '03)*. New York, NY, USA: ACM, 2003, pp. 164–177.

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

[22] M. Feldman, K. Lai, and L. Zhang, "The proportional-share allocation market for computational resources," *IEEE Trans. on Parallel and Distributed Systems*, vol. 20, pp. 1075–1088, 2009.

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

[24] S. Di, D. Kondo, and C.L. Wang, "Optimization and stabilization of composite service processing in a cloud system, " *Proc. of IEEE/ACM 21st International Symposium on Quality of Service (IWQoS'13)*, 2013, pp. 1–10.

[25] 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.

[26] C. Jiang, C. Wang, X. Liu, and Y. Zhao, "A survey of job scheduling in grids," in *Proc. 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.

[27] 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.

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

[29] 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.

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

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

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

[33] M. Mao, J. Li, and M. Humphrey, "Cloud auto-scaling with deadline and budget constraints," in *Proc. of 11th IEEE/ACM International Conference on Grid Computing (Grid'10)*, 2010, pp. 41–48.

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

[35] S. Kaur and A. Verma, "An Efficient Approach to Genetic Algorithm for Task Scheduling in Cloud Computing", *International Journal of Information Technology and Computer Science*, Vol. 10, pp. 74–79, 2012.

[36] S. Zhan and H. Huo, "Improved PSO-based Task Scheduling Algorithm in Cloud Computing", *Journal of Information and Computational Science*, vol 9, no. 13, pp. 3821–3829, 2012.

[37] M. Stillwell, F. Vivien and H. Casanova, "Virtual Machine Resource Allocation for Service Hosting on Heterogeneous Distributed Platforms," in *Proc. of the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS'12)*, Shanghai, China, 2012, pp. 786–797.

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

[39] 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 *Proc. of the 4th ACM European Conference on Computer systems (EuroSys'09)*, New York, NY, USA: ACM, 2009, pp. 1–12.

[40] 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.

[41] B. Nicolae and F. Cappello, "BlobCR: efficient checkpoint-restart for HPC applications on IaaS clouds using virtual disk image snapshots, " in *Proc. of IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*, 2011, pp. 34:1–34:12.

Sheng Di Sheng Di received his Master (M.Phil) degree from Huazhong University of Science and Technology in 2007 and Ph.D degree from The University of Hong Kong in Nov. of 2011, both on Computer Science. Dr. Di's research interest involves optimization of distributed resource allocation in large-scale cloud platforms, characterization and prediction of workload at Cloud data centers, fault tolerance on Cloud/HPC, and so on. Contact him at sdi1@anl.gov.

Derrick Kondo Derrick Kondo is a tenured research scientist at INRIA, France. He received his Bachelor's at Stanford University in 1999, and his Master's and Ph.D. at the University of California at San Diego in 2005, all in computer science. His general research interests are in the areas of reliability, fault-tolerance, statistical analysis, job and resource management. In 2009, he received a Young Researcher Award (similar to NSF's CAREER Award), and received Amazon Research Award in 2010, and Google Research Award in 2011. Contact him at derrick.kondo@inria.fr.

Cho-Li Wang Cho-Li Wang received his Ph.D. degree from University of Southern California in 1995. Prof. Wang's research interests include multicore computing, software systems for Cluster and Grid computing, and virtualization techniques for Cloud computing. He serves on the editorial boards of several international journals, including IEEE Transactions on Computers (2006-2010), Journal of Information Science and Engineering, and Multiagent and Grid Systems. Contact him at clwang@cs.hku.hk.