<table>
<thead>
<tr>
<th><strong>Title</strong></th>
<th>Optimization and stabilization of composite service processing in a cloud system</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Author(s)</strong></td>
<td>Di, S; Kondo, D; Wang, CL</td>
</tr>
<tr>
<td><strong>Citation</strong></td>
<td>The IEEE/ACM 21st International Symposium on Quality of Service (IWQoS 2013), Montreal, QC., 3-4 June 2013. In International Workshop on Quality of Service, 2013, p. 1-10</td>
</tr>
<tr>
<td><strong>Issued Date</strong></td>
<td>2013</td>
</tr>
<tr>
<td><strong>URL</strong></td>
<td><a href="http://hdl.handle.net/10722/189640">http://hdl.handle.net/10722/189640</a></td>
</tr>
<tr>
<td><strong>Rights</strong></td>
<td>International Workshop on Quality of Service. Copyright © Institute of Electrical and Electronics Engineers.</td>
</tr>
</tbody>
</table>
Optimization and Stabilization of Composite Service Processing in a 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 virtual machines (VM), we design a cloud system aiming to optimize the overall performance, in processing user requests made up of composite services. We address three contributions. (1) We optimize VM resource allocation with a minimized processing overhead subject to task’s payment budget. (2) For maximizing the fairness of treatment in a competitive situation, we investigate the best-suited scheduling policy. (3) We devise a resource sharing scheme adjusted based on Proportional-Share model, further mitigating the resource contention. Experiments confirm two points: (1) mean task response time approaches the theoretically optimal value in non-competitive situation; (2) as system runs in short supply, each request could still be processed efficiently as compared to their ideal results. Combining Lightest Workload First (LWF) policy with Adjusted Proportional-Share Model (LWF+APSM) exhibits the best performance. It outperforms others in a competitive situation, by 38% w.r.t. worst-case response time and by 12% w.r.t. fairness of treatment.

I. INTRODUCTION

Cloud computing [1] has emerged as a compelling paradigm for the deployment of ease-of-use virtual execution environment on the Internet. It allows users to customize their own services based on specific purposes. Platform as a Service (PaaS) is one of the 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 model is based on the PaaS paradigm, in which users can compose complex requests by combining a set of off-the-shelf web services. Each service is associated with a price, which is assigned by its maker. When a user submits a request calling other services, he/she needs to pay the resource consumption. On the other hand, we leverage the VM resource isolation technology [3] to refine the resource allocation. The key question is how to split the physical resources according to users’ requirements, how to minimize the overhead in data transmission and the operations of virtual machine monitor (VMM) (i.e., cost of hypervisor), and how to queue user requests when necessary. This problem is quite challenging in that each user request is determined by a different structure that is made up of various services, and also associated with a varied budget to restrict its payment.

Our objective is to optimize and stabilize the Quality of Service (QoS) of each request (or task) with virtual machines, especially for competitive situations. Each task is made up of a set of subtasks (instance of service), and the whole response time (or wall-clock time) of each task is expected to be minimized. We will show that virtual machine technology can not only provide elastic and isolated execution environment on demand, but it can also be used to improve the system-wide performance and stability.

In this paper, we tackle three key issues listed below.

- **Minimizing Processing Overhead**: 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 use VM isolation to refine the resource allocation, the cost of VMM operations (such as the time cost in changing CPU-capacity 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.

- **Investigating Best-suited Scheduling Policy (or Queuing Policy)**: A competitive situation with low availability will easily delay the responses of some requests with unfairness of the overall treatment. We investigate the best-suited queuing policy, including 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 length, which is similar to Heterogeneous Earliest Finish Time (HEFT) [4]. LWF and SSTF can be considered Shortest Job First (SJF) and min-min algorithm [5] respectively. The principle of SPF is similar to Earliest Deadline First (EDF) [6].

- **Optimizing Divisible-Resource Allocation**: how to adjust the divisible resource allocation among running tasks to adapt to the competitive situation. We devise an adjusted resource allocation in terms of task structures like workload or varied estimated progress, which can further mitigate the influence of resource contention.

Over a real-cluster environment deployed with XEN’s hypervisor (a.k.a., VMM) [7], we implement a distributed prototype that is able to solve/calculate complex matrix problems. Experiments show that the worst-case performance under LWF 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. 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 about 10-30%, and the fairness can be kept over 0.8. 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 derived from the convex-optimization theory [8].

The remainder of the paper is organized as follows. In Section II, we present the overview of our composite cloud service system. In Section III, we formulate the research problem, 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 OVERVIEW

The system architecture of our composite cloud 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 compose a computing request (i.e., a task) by a set of individual steps (involved with various services) and each step could be represented as a matrix calculation (e.g., matrix-product and matrix-decomposition). Each such matrix calculation can be considered a subtask, and the whole task is expected to be completed as soon as possible with a 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 and each subtask is executed on a VM, with various types of resources assigned via VMM (a.k.a., hypervisor). Our work is 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 process the task execution with minimized overhead.

Each task is processed according to the pseudo-code shown in Algorithm 1, also as shown in Figure 1 (b). At 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 $r^*(t)$=$r^*(t(1))$, $r^*(t(2))$, $\cdots$, $r^*(t(m))$. 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 $r^*(t(i))$), waiting for the task scheduling notification with a selected qualified host and an idle VM running atop it. 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 [9] to isolate CPU rate among VMs on the same hardware. 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 target VMs are running 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 $t_i$.

Since each task is composed of a series of subtasks, its total execution time can be denoted as $T(t_i)=\sum_{j=1}^{m_i} \frac{t_i(j)}{r_i(j)}$. 

![System Architecture](image-url)

**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
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 transmit between subtasks, and data to read from disk) and the resource fractions 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 types of 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 [10]. 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, since they are the ones who pay monthly resource leases to Infrastructure-as-a-Service (IaaS) providers (e.g., Amazon EC2 [11]). 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).

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

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

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

$$RER(t_i) = \frac{t_i’s\ real\ response\ time}{t_i’s\ theoretically\ optimal\ length}$$  \hspace{1cm} (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 task’s theoretically optimal length (TOL) is the sum of the theoretical execution length of each subtask under the theoretically optimal resource allocation, which is the solution to the above convex-optimization problem (Formula (1) and Formula (2)). The real 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 (e.g., on resource allocation and data transmission), subtask’s execution time, processing 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 (or stability) of the treatment in the system. Its value ranges in [0, 1], and the bigger, the higher fairness of the treatment. Based on Formula (3), the fairness is also related to different types of execution overheads. How to effectively coordinate the overheads among tasks is very challenging, because of heterogeneous task structures, task budget, and dynamically varied resource availability over time.

$$fairness(t_i) = \frac{\left(\sum_{j=1}^{m_i} RER(t_{i(j)})\right)^2}{n \sum_{j=1}^{m_i} RER^2(t_{i(j)})}$$  \hspace{1cm} (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

We need to minimize the overheads raised at each step in the course of task execution. In general, there are two major reasons for over-large RER and unfairness of treatment, especially in a competitive situation: (1) the remarkable waiting time cost in task scheduling; (2) the possible over-heaviness of task execution. We design an adjusted scheme to dynamically allocate isolated resources for running tasks. We first derive optimal resource fractions for a task subject to its workload and budget, in a non-competitive situation. Then we explore 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. We first derive optimal resource fractions for a task subject to its workload and budget, in a non-competitive situation. Then we explore the best-fit solution to the above problem on the three facets, resource allocation, task scheduling, and minimization of overheads.

We design an adjusted scheme to dynamically allocate isolated resources for running tasks. We first derive optimal resource fractions for a task subject to its workload and budget, in a non-competitive situation. Then we explore the best-fit solution to the above problem on the three facets, resource allocation, task scheduling, and minimization of overheads.

We design an adjusted scheme to dynamically allocate isolated resources for running tasks. We first derive optimal resource fractions for a task subject to its workload and budget, in a non-competitive situation. Then we explore the best-fit solution to the above problem on the three facets, resource allocation, task scheduling, and minimization of overheads.
In order to minimize \( T(t_i) \), the optimal resource vector \( r^*_i(j) \) should use up all the budget (i.e., let the total payment be equal to \( B(t_i) \)). Then, we can get Equation (5).

According to Theorem 1, we can easily compute the optimal resource vector for any task based on its budget constraint. Specifically, \( r^*_i(j) \) is the theoretically optimal resource fraction (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 resource fractions compared to the value \( r^*_i(j) \), it would be useless for the task \( t_i \) due to its limited budget. Thus, the resource allocator should set each subtask’s CPU capacity \(^1\) (i.e., the maximum CPU rate) as its theoretically optimal resource fraction, Formula (5).

If the system runs in short supply, it is likely that the total sum of their optimal resource fractions (i.e., \( r^*_i(t) \)) may exceed hosts’ resource capacities. To this end, it is necessary to coordinate tasks’ priorities, such that none of tasks’ real execution lengths would be extended noticeably compared to its theoretically optimal execution length (i.e., minimizing RER\((t_i)\) for each task \( t_i \)). In our system, we improve the proportional-share mechanism (PSM) with XEN’s credit scheduler to manage subtask’s resource utilization.

With 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).

\[
\begin{align*}
    r(v_j) & = \frac{w(v_j)}{\sum_{k=1}^{n_i} w(v_k)} c_i \\
\end{align*}
\]

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 \( x_1 = 1, 2, \cdots, m_i \), then, \( w(t_i(j)) \) will be determined by Formula (9). We call it Adjusted Proportional-Share Model (APSM).

\[
\begin{align*}
    w(t_i(j)) & = \begin{cases} \\
    \eta \cdot r^*_i(j) & l_i \leq \alpha \\
    r^*_i(j) & \alpha < l_i \leq \beta \\
    \frac{1}{\eta} \cdot r^*_i(j) & l_i > \beta 
    \end{cases} \\
\end{align*}
\]

The weight values in our design (Formula (9)) are determined by four parts, the extension coefficient \( \eta \), theoretically optimal resource fraction \( r^*_i(j) \), the threshold value \( \alpha \) to determine short tasks, and the threshold value \( \beta \) to determine long tasks. Apparently, \( \eta \)'s value is supposed to be always greater than 1. In reality, tuning \( \eta \)'s value could adjust the extension degree for short/long tasks. Changing the values of \( \alpha \) and \( \beta \) 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 some new resources are released due to 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 task 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 (SSTF). 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. 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.

---

\(^1\)the capacity-setting command is “\texttt{xm sched-credit -d VM -c r^*_i(j)}”.
\(^2\)the weight-setting command is “\texttt{xm sched-credit -d VM -w weight}.”
• **Lightest-Workload-First (LWF)**. LWF schedules the subtasks based on the predicted workload of their corresponding tasks. 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 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 its real execution length as its theoretically optimal length (TOL) is a fixed constant. Since tasks’ TOLs are different due to their heterogeneous structures, workloads, and budgets, the overhead on data transmission and VM operations is relatively fixed. That is, the tasks with smaller TOL are more sensitive to the overheads, thus these tasks are supposed to be scheduled with higher priorities.

• **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 Progress (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 $TOL(t_i)=\sum_{j=1}^{m_i} \frac{l_i(j)}{p_i(j)}$. 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.

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

$$TP(t_i) = \frac{wall\text{-}clock\ text{time} \text{ \ since } t_i^i\text{’s submission} }{TOL(t_i)}$$

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 [5], 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.

### C. Minimization of Processing Overheads

In this section, we intensively study how to minimize the impact of processing overheads to the task’s whole response length at runtime. Our idea is illustrated in Figure 2.

![Figure 2. Illustration of Task Response Time](image)

Figure 2 shows a subtask’s wall-clock time. At the beginning (Step 1), the subtask A will be put in the scheduling queue until there are qualified available resource matching its demand. Then, the scheduler will notify the VMM 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 [7], 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 VM’s memory size and working state). This cost cannot be overlooked especially for the short tasks whose TOLs (i.e., theoretically optimal wall-clock time) are short (say a few 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 aforementioned.

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. 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 composite cloud service prototype that can help solving any dense-matrix based problems. Dense-matrix computation is a very fundamental domain in mathematics, which is widely used in linear-algebra research. Quite a few Grid, cloud, and web services [12], [13], [14], [15] have been developed to suit ease-of-use demand.

In our prototype, each matrix problem may consist of a series of nested matrix computations. For example, a user could submit a request like solving the matrix equation \( (A_{m \times n} \cdot A_{n \times m})^k x = B_{m \times m} \). 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: \( D_{m \times m} = C_{m \times m}^k \); (3) Least squares solution of \( D \cdot X = B \) based on QR-Decomposition: \( \text{Solve}\{ D_{m \times m} \cdot X = B_{m \times m} \} \).

In our experiment, we are assigned with 8 physical nodes to use from the most powerful supercomputer at HongKong (called Gideon-II [16]), and each node owns 2 quad-core Xeon CPU E5540 (i.e., totally 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 [7] 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 [17] to perform the math computations, each consisting of a set of matrix operations. ParallelColt [17] 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 shown in Table I). Each generated matrix must be compatible for each matrix operation (e.g., two matrices in a matrix product must be in the form of \( A_{m \times n} \) and \( B_{n \times p} \) respectively). We also characterize the single-core execution length (or workload) for each service, as shown in Table I. Among the 10 matrix-computation services, three services are coded via single threads, including matrix-matrix multiply, QR-decomposition, and matrix-power, thus their computation can get an approximate-linear speedup when being allocated with 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.

B. Experimental Results

1) Demonstration of Competitive Degrees: We first characterize the various competitive degrees, to confirm the competitive situations in our experiment. The competitive degree is evaluated via two metrics, \( \text{Allocate-Request Ratio} \) (abbreviated as ARR) and \( \text{Queue Length} \) (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 competitive 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.

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 does impact the performance a lot in the cloud system.

Figure 4 presents the queue length (QL) increases with the number of tasks submitted, confirming the competitive situation in our experiment. QL differs a lot under different scheduling policies. For example, when there are 24 tasks submitted, SSTF and LWF both lead to small number of waiting tasks (about 5-6 and 6-7 respectively). By contrast, the QLs under SPF and SOLF are about 8-10 and 10-12 on average, implying a higher waiting cost.

Table I

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>11.7</td>
<td>2.6</td>
<td>m=10</td>
<td>2.21</td>
<td>0.001</td>
<td>0.010</td>
<td>1.6</td>
<td>0.375</td>
<td>0.94</td>
<td>0.014</td>
</tr>
<tr>
<td>1000</td>
<td>11</td>
<td>2.7</td>
<td>m=20</td>
<td>55</td>
<td>0.003</td>
<td>0.011</td>
<td>8.9</td>
<td>1.25</td>
<td>7.25</td>
<td>0.021</td>
</tr>
<tr>
<td>1500</td>
<td>138</td>
<td>35.7</td>
<td>m=20</td>
<td>193.3</td>
<td>0.005</td>
<td>0.03</td>
<td>29.9</td>
<td>4.43</td>
<td>24.6</td>
<td>0.047</td>
</tr>
<tr>
<td>2000</td>
<td>199.3</td>
<td>78.8</td>
<td>m=10</td>
<td>396</td>
<td>0.006</td>
<td>0.043</td>
<td>67.8</td>
<td>10.2</td>
<td>57.2</td>
<td>0.097</td>
</tr>
<tr>
<td>2500</td>
<td>210.1</td>
<td>99.5</td>
<td>m=20</td>
<td>1015</td>
<td>0.017</td>
<td>0.111</td>
<td>132.6</td>
<td>18.7</td>
<td>109</td>
<td>0.141</td>
</tr>
</tbody>
</table>

Figure 3. Allocation vs. Request With Different Competitive Degrees

Figure 4. Queue Lengths With Different Scheduling Policies

2) Investigation of Best-suited Scheduling Policy: 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 Cartesian product between \( \{5 \text{ sec.}, 10 \text{ sec.}, 20 \text{ sec.}\} \) and \( \{100 \text{ sec.}, 200 \text{ sec.}, 300 \text{ sec.}\} \), and set \( \eta \) 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, our designed APSM outperforms the traditional 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 7.230% w.r.t. the max. value of RER, and by at least 2.74% 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>
<thead>
<tr>
<th>strategy</th>
<th>min.</th>
<th>avg.</th>
<th>max.</th>
<th>fairness</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCFS+PSM</td>
<td>0.732</td>
<td>3.665</td>
<td>21.097</td>
<td>0.345</td>
</tr>
<tr>
<td>FCFS+APSM</td>
<td>0.644</td>
<td>3.779</td>
<td>21.755</td>
<td>0.388</td>
</tr>
<tr>
<td>LWF+PSM</td>
<td>0.712</td>
<td>1.809</td>
<td>5.974</td>
<td>0.703</td>
</tr>
<tr>
<td>LWF+APSM</td>
<td>0.666</td>
<td>1.790</td>
<td>5.239</td>
<td>0.714</td>
</tr>
<tr>
<td>SOLF+PSM</td>
<td>0.720</td>
<td>3.331</td>
<td>17.004</td>
<td>0.482</td>
</tr>
<tr>
<td>SOLF+APSM</td>
<td>0.750</td>
<td>2.780</td>
<td>10.803</td>
<td>0.575</td>
</tr>
<tr>
<td>SSTF+PSM</td>
<td>0.746</td>
<td>2.108</td>
<td>8.706</td>
<td>0.573</td>
</tr>
<tr>
<td>SSTF+APSM</td>
<td>0.769</td>
<td>2.119</td>
<td>7.230</td>
<td>0.618</td>
</tr>
<tr>
<td>SWPF+PSM</td>
<td>0.708</td>
<td>6.106</td>
<td>57.925</td>
<td>0.209</td>
</tr>
<tr>
<td>SWPF+APSM</td>
<td>0.649</td>
<td>6.233</td>
<td>59.627</td>
<td>0.206</td>
</tr>
<tr>
<td>STPF+PSM</td>
<td>0.707</td>
<td>2.830</td>
<td>14.867</td>
<td>0.476</td>
</tr>
<tr>
<td>STPF+APSM</td>
<td>0.713</td>
<td>3.147</td>
<td>15.853</td>
<td>0.474</td>
</tr>
</tbody>
</table>

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, 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 an 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 also 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.

We also evaluate the effectiveness of our design in a non-competitive situation (when there are only 8 tasks submitted), as shown in Table III. In such a situation, all tasks can always be allocated with theoretically optimal resource fractions due to the non-competitive state in the system. Thus, the performance does not differ a lot with different scheduling policies. Specifically, the mean value of the task execution length is only slightly higher than its theoretically optimal value by about 10-30%. The maximum RERs and the fairness indexes of all strategies are always lower than 3 and greater than 0.8 respectively. There are two reasons why RER may not be always kept close to 1 in the non-competitive situation: (1) In computing RER (Formula 3), task’s TOL does not take into account transmission cost while task’s real response time contains. (2) The workload prediction may have more or less errors compared to their real execution. In our characterization, we find that the workload prediction errors of the execution lengths of some matrix calculations like matrix-decomposition could be even up to about 15%.

<table>
<thead>
<tr>
<th>strategy</th>
<th>min.</th>
<th>avg.</th>
<th>max.</th>
<th>fairness</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCFS+PSM</td>
<td>0.906</td>
<td>1.300</td>
<td>2.025</td>
<td>0.891</td>
</tr>
<tr>
<td>FCFS+APSM</td>
<td>0.878</td>
<td>1.243</td>
<td>2.041</td>
<td>0.878</td>
</tr>
<tr>
<td>LWF+PSM</td>
<td>0.933</td>
<td>1.308</td>
<td>2.092</td>
<td>0.876</td>
</tr>
<tr>
<td>LWF+APSM</td>
<td>0.863</td>
<td>1.320</td>
<td>2.331</td>
<td>0.840</td>
</tr>
<tr>
<td>SOLF+PSM</td>
<td>0.901</td>
<td>1.376</td>
<td>2.723</td>
<td>0.811</td>
</tr>
<tr>
<td>SOLF+APSM</td>
<td>0.871</td>
<td>1.324</td>
<td>2.205</td>
<td>0.863</td>
</tr>
<tr>
<td>SSTF+PSM</td>
<td>0.911</td>
<td>1.270</td>
<td>2.000</td>
<td>0.893</td>
</tr>
<tr>
<td>SSTF+APSM</td>
<td>0.891</td>
<td>1.327</td>
<td>2.318</td>
<td>0.859</td>
</tr>
<tr>
<td>SWPF+PSM</td>
<td>0.882</td>
<td>1.725</td>
<td>1.581</td>
<td>0.925</td>
</tr>
<tr>
<td>SWPF+APSM</td>
<td>0.860</td>
<td>1.413</td>
<td>2.183</td>
<td>0.845</td>
</tr>
<tr>
<td>STPF+PSM</td>
<td>0.941</td>
<td>1.262</td>
<td>2.044</td>
<td>0.881</td>
</tr>
<tr>
<td>STPF+APSM</td>
<td>0.883</td>
<td>1.369</td>
<td>2.440</td>
<td>0.829</td>
</tr>
</tbody>
</table>

Through Table III, another key finding is that the good solutions in competitive situations may perform not very well in non-competitive situations. The best solution in the non-competitive situation is not LWF+APSM but STPS+PSM. In addition, it is also observed that PSM usually outperforms APSM slightly in the non-competitive situation. That is, APSM will have resource allocation be a little over-adjusted against the optimal solution. Such a lesson inspires us to dynamically optimize the performance for both situations via an adaptive solution, which will be our future work.

3) Speedup of Task’s Execution: Finally, we evaluate the speedup of task’s execution in our cloud system, as compared to the task’s single-core execution length. Figure 5 shows the mean values of task execution speedup, with different scheduling policies in both competitive situation (when 24 tasks are submitted, and AAR≈\(\frac{1}{2}\)) and non-competitive situation (when 8 tasks are submitted).

Through Figure 5, it is clearly observe that in a competitive situation, LWF performs the best among all scheduling policies. The mean speedup under the LWF policy with APSM is higher than that under FCFS+PSM by \(\frac{3.112}{2.57} = 1 = 21.1\%\). For the non-competitive situation, we find that LWF+APSM still performs the best. It gets the highest average speedup with 4.539, which is higher than FCFS+PSM by \(\frac{4.539}{3.57} = 1 = 24.8\%\). Hence, from the perspective of speedup, LWF+APSM is the most recommended solution in both competitive and non-competitive situations.

Figure 5 also shows that the Adjusted PSM (APSM)
in most of cases performs better than the standard PSM. For instance, LWF+APSM outperforms LWF+PSM by about 15%. This is because APSM tends to assign more resource fractions to short tasks at runtime, such that the overall waiting cost can be further mitigated for all tasks. In addition, Figure 5 also shows the mean speedup of tasks running in a non-competitive situation is always higher than in a competitive situation by about 19-62%, which is reasonable as the tasks are supposed to be assigned with more resource fractions in a non-competitive situation.

VI. RELATED WORK

Although job scheduling problem in Grid computing [18] has been extensively studied for years, most of them (such as [19], [20]) are not suited for our composite cloud service processing environment. Grid jobs are often with long execution length, while cloud tasks are often short based on [21]. 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. Zaharia et al. [22] designed a task scheduling method to improve the performance of Hadoop [23] for a heterogeneous environment (such as a pool of VMs each customized with different processing abilities). Unlike the FIFO policy and speculative execution model originally used in Hadoop, they designed a so-called Longest Approximate Time to End (LATE) policy. Such a policy assigns higher priorities to the jobs with longer remaining execution lengths, which is similar to the SPF policy evaluated in our experiment. Their intuitive idea is maximizing the opportunity for a speculative copy to overtake the original and reduce job’s response time. Isard et al. [24] proposed a fair scheduling policy (namely Quincy) in order to maximize the scheduling fairness and minimize the data transmission cost meanwhile. Compared to these works, our cloud systems work with a strict pay-as-you-go model, under which the optimal resource allocation for each task can be computed based on convex optimization. Mao et al. [25] 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 short-supply situation by assuming the resource pool is relatively sufficient and users have unlimited budgets.

In addition to the task scheduling model, much cloud management research focuses on the optimization of resource assignment. Unlike Grid systems whose resources are often exclusively consumed by jobs, cloud resource allocation is able to be refined based on elastic demand by leveraging VM resource isolation technology. Stillwell et al. [26] explored how to optimize the resource allocation for service hosting on a heterogeneous distributed platform. Their work 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 explored the best-suited scheduling policy and resource allocation scheme for the competitive situation. We also take into account user payment, and evaluate our solution on a real-VM-deployment environment which tackles more practical technical issues like minimization of various execution overheads. Meng et al. [27] analyzed VM-pairs’ compatibility in terms of the forecasted workload and estimated VM sizes. SnowFlock [28] 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 [29] also proposed a resource allocation method for cloud computing environments especially based on divisible resources. In comparison, the key advantage of our design is to guarantee each task’s QoS at a satisfactory level with an overall fair treatment, even in 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. We investigated the best-suited task scheduling policy under a composite service processing model and explored an adjusted strategy with convex-optimization theory and minimized processing overhead. We address four key lessons below.

• Our convex-optimization model with VM resource isolation technology leads to near-optimal performance in a non-competitive situation.
• In a competitive situation, short tasks (with short single-core execution length) are better to be assigned more powerful resource fractions than their theoretical values derived from the convex-optimization theory.
• Experiments confirm that applying the Lightest-Workload-First (LWF) policy with a Proportional-Share resource allocation with the credit being set to the adjusted task workload delivers the best results. It outperforms other strategies in the competitive situation,

![Figure 5. Average Task Execution Speedup](image-url)
by 38% w.r.t. the worst-case response time and by 12% w.r.t. the fairness of the treatment.

- Experiments show that LWF policy with Adjusted PSM lead to the highest task execution speedup in both competitive and non-competitive situations. It outperforms FCFS+PSM by 21.1% and 24.8% respectively.

In the future, we plan to explore the most accurate coefficients (e.g., $\eta$) 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