Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Profit-driven scheduling for cloud services with data access awareness

Young Choon Lee[a,*], Chen Wang[b], Albert Y. Zomaya[a], Bing Bing Zhou[a]

[a] *Centre for Distributed and High Performance Computing, School of Information Technologies, University of Sydney, NSW 2006, Australia*

[b] *CSIRO ICT Center, PO Box 76, Epping, NSW 1710, Australia*

ARTICLE INFO

Article history:
Received 25 July 2011
Received in revised form
1 November 2011
Accepted 11 December 2011
Available online 29 December 2011

Keywords:
Scheduling
Cloud computing
Market-based resource allocation
Processor-sharing
Mashup services
Data service

ABSTRACT

Resource sharing between multiple tenants is a key rationale behind the cost effectiveness in the cloud. While this resource sharing greatly helps service providers improve resource utilization and increase profit, it impacts on the service quality (e.g., the performance of consumer applications). In this paper, we address the reconciliation of these conflicting objectives by scheduling service requests with the dynamic creation of service instances. Specifically, our scheduling algorithms attempt to maximize profit within the satisfactory level of service quality specified by the service consumer. Our contributions include (1) the development of a pricing model using processor-sharing for clouds (i.e., queuing delay is embedded in processing time), (2) the application of this pricing model to composite services with dependency consideration, (3) the development of two sets of service request scheduling algorithms, and (4) the development of a prioritization policy for data service aiming to maximize the profit of data service.

1. Introduction

Cloud computing as the strategic marriage between traditional distributed computing and utility computing greatly facilitates IT outsourcing (e.g., workload offloading). Main advantages of this paradigm shifting technology include cost-effectiveness, flexibility and scalability [1,24]. Resource sharing between multiple tenants is the key rationale behind the cloud. Resources in a cloud are not restricted to hardware, such as processors and storage devices, but can also be software stacks and Web service instances. Although the notion of a cloud has existed in one form or another for some time now (its roots can be traced back to the mainframe era [18]), recent advances in virtualization technologies and the business trend of reducing the total cost of ownership (TCO) in particular have made it much more appealing compared to when it was first introduced.

While resource sharing greatly helps service providers improve resource utilization and increase profit, it impacts on the service quality (e.g., the performance of consumer applications). In other words, the service provider aims to accommodate/process as many requests as possible with a main objective to maximize profit, and this may conflict with consumer's performance requirements (e.g., response time). In today's cloud computing platforms, the service quality (performance guarantee) that consumers expect for their applications/requests is left open and pricing is simply based on service usage; together, this is a great burden on the consumer's side.

There have been a number of studies exploiting market-based resource allocation to resolve those conflicting objectives. Noticeable scheduling mechanisms include FirstPrice [3], FirstProfit [19], proportional-share [4,14,21], and recently a flexible resource allocation strategy is proposed, which is probably the closest work to ours in this paper. As the cloud plays as a market for computing and data storage/processing services (or simply IT services), these works are of great importance and relevance to our study. However, most of them are limited to job scheduling in conventional supercomputing settings. Specifically, they are only applicable to scheduling batch jobs in systems with a fixed number of resources. User applications that require the processing of mashup services, which are quite common in the cloud, are not considered by these mechanisms. The scenario addressed in this study is different in terms of application type and the organization of the cloud.

In clouds, particularly that offer infrastructure services (e.g., Amazon EC2 instances), there are primarily two types of consumers: bare users and business service providers (or simply service providers). Service providers further deal with their service users or consumers. This study focuses on the cloud market between service providers and their service consumers. All these three parties (infrastructure providers, service providers and consumers) strive for cost efficiency. However, the interpretation of this cost efficiency differs between them. While service consumers seek for best value for money (better performance) service

* Corresponding author.
E-mail addresses: young.lee@sydney.edu.au, yclee@it.usyd.edu.au (Y.C. Lee), chen.wang@csiro.au (C. Wang), albert.zomaya@sydney.edu.au (A.Y. Zomaya), bing.zhou@sydney.edu.au (B.B. Zhou).

0743-7315/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2011.12.002

providers primarily aim to maximize revenue/profit (improved resource utilization), and thus, their objectives conflict with each other. The reconciliation of these conflicting objectives is the key to prevalence of the cloud.

The specific problem addressed in this paper is the scheduling of consumers' service requests (or applications) on service instances made available by service providers taking into account costs—incurred by both consumers and infrastructure providers—as the most important factor. This scheduling problem that we previously dealt with in [15] is further extended with the introduction of data service and the association of data accesses with services. One might be interested in another work of ours in [16] if energy efficiency in the cloud is the main concern rather than cost efficiency dealt with in this work. Our contributions include (1) the development of a pricing model using processor-sharing (PS) for clouds, (2) the application of this pricing model to composite services (to the best of our knowledge, the work reported in this study is the first published in the literature), (3) the development of two sets of profit-driven scheduling algorithms exploiting key characteristics of (composite) service requests including precedence constraints, and (4) the development of a prioritization policy for data service aiming to maximize profit. Note that the data service is often First-Come-First-Serve based. It does not fit PS model and therefore needs special treatment.

In addition to introduction of data service, our experiments take into account mainly services that are dynamically composed by a variety of applications compared with [15].

Our service request scheduling algorithms are primarily different in their calculation of profitability. The first set of algorithms takes into account not only the profit achievable from the current service, but also the profit from other services being processed on the same service instance. The second set of algorithms attempts to maximize service-instance utilization without incurring loss/deficit; this implies the minimization of costs to rent resources from infrastructure providers.

The prioritization of data access requests in our policy is performed using the time-varying utility function we adopt in this paper.

The rest of the paper is organized as follows. Section 2 describes the cloud, application, scheduling and data service models used in this paper. Our pricing model with the incorporation of PS is discussed in Section 3 leading to the formulation of our objective function. We present our profit-driven service request scheduling algorithms and data access prioritization policy in Section 4. Section 5 describes experimental settings and performance metrics used in our evaluation study followed by results of our performance evaluation in Section 6. Related work is discussed in Section 7. We then summarize our work and draw conclusions in Section 8.

2. Models

In this section, we begin by describing the system and application models focusing on the specific characteristics of the scheduling scenario considered in this work; that is, the three-tier cloud structure and the inter-related composite service request models. The scheduling model for service requests and the incorporation of their data access are detailed.

2.1. Cloud system model

We consider a three-tier cloud structure (Fig. 1), which consists of infrastructure providers, service providers and consumers. Here, again the latter two parties are of our particular interest. This three-tier structure can be commonly found in the current cloud computing environments. For example, there are many service

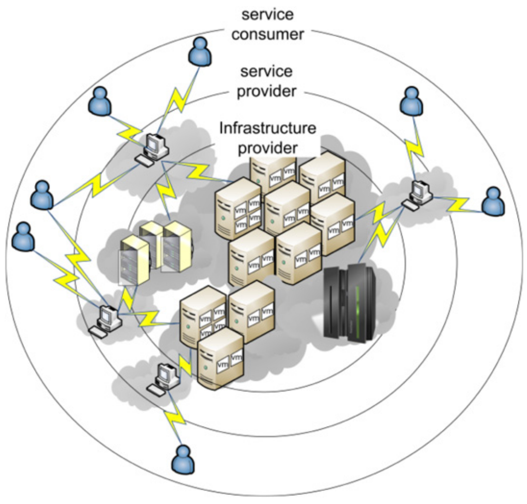

Fig. 1. A three-tier cloud structure.

providers who deploy their services in Amazon EC2 and consumers (or end-users) access those services.[1]

A cloud computing system – owned and operated by an infrastructure service provider – consists of a set of physical resources (server computers) in each of which there are one or more processing elements/cores; these resources are fully interconnected in the sense that a route exists between any two individual resources. We assume resources are homogeneous in terms of their computing capability and capacity; this can be justified using virtualization technologies. Nowadays, as many-core processors and virtualization tools (e.g., VMware Workstation and Xen) are commonplace, the number of concurrent tasks on a single physical resource is loosely bounded. Although a cloud can span across multiple geographical locations (i.e., distributed), the cloud model in our study is assumed to be confined to a particular physical location.

A service provider rents resources from cloud infrastructure providers and prepares a set of services in the form of virtual machine (VM) images; the provider is then able to dynamically create instances from these VM images. The underlying cloud computing infrastructure service is responsible for dispatching these instances to run on physical resources. A running instance is charged by the time it runs at a flat rate per time unit. It is in the service provider's interests to minimize the cost of using the resources offered by the cloud infrastructure provider (i.e., resource rental costs) and maximize the revenue (specifically, net profit) generated through serving consumers' applications.

From the service consumer's viewpoint, a service request for an application consisting of one or more services is sent to a provider specifying two main constraints, time and cost. Although the processing (response) time of a service request can be assumed to be accurately estimated, it is most likely that its actual processing time is longer than its original estimate due primarily to delays (e.g., queuing and/or processing) occurring on the provider's side. This time discrepancy issue is typically dealt with using service level agreements (SLAs).

2.2. Application model

Services offered in the cloud system can be classified into software as a service (SaaS), platform as a service (PaaS) and infrastructure as a service (IaaS). Since this study's main interest

[1] http://aws.amazon.com/solutions/case-studies/.

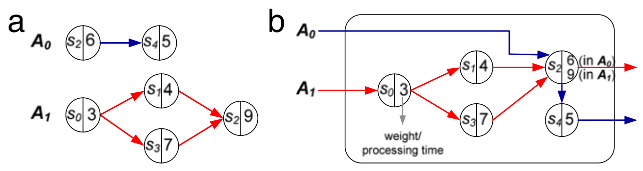

Fig. 2. Two service requests with an overlapping service. (a) Consumers' applications. (b) Actual service request processing on provider's side.

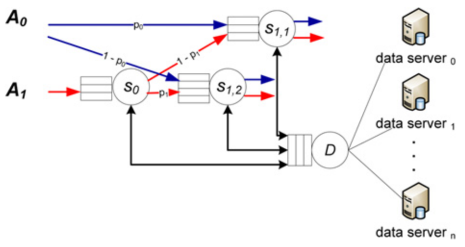

Fig. 3. The scenario for services with data accesses: D is the data service with n data servers.

is in the economic relationship between service providers and consumers, services in this work can be understood as either of the first two types.

An application A outsources several functionalities to a set of services S, $S = \{s_0, s_1, \ldots, s_n\}$ and these services are interdependent on each other in processing requests arriving at the application, i.e., precedence-constrained (A_0 and A_1 in Fig. 2). More formally, consumer applications in this work can be represented by a directed acyclic graph (DAG). A DAG, $A = (S, E)$, consists of a set S of n nodes and a set E of e edges. A DAG is also called a task graph or macro-dataflow graph. The nodes represent services comprising an application; the edges represent precedence constraints. An edge $(i, j) \in E$ between service s_i and service s_j also represents inter-service dependency. In other words, service s_j can only start its processing once service s_i completes its processing. A service with no predecessors is called an entry service, s_{entry}, whereas an exit service, s_{exit}, is one that does not have any successors. Among the predecessors of a service s_i, the predecessor which completes the processing at the latest time is called the most influential parent (MIP) of the service denoted as MIP_i. The longest path of a task graph is the critical path (CP).

The weight on a service s_i denoted as w_i represents the processing time of the service. Since the performance of multiple instances of the same service is assumed to be identical, for a given service we do not differentiate its weights on different service instances. However, the weight of a service may differ between applications (s_2 in Fig. 2).

The earliest start time of, and the earliest finish time of, a service s_i are defined as:

$$est_i = \begin{cases} \text{arrival time of } s_i & \text{if } s_i = s_{entry} \\ eft_{MIP_i} & \text{otherwise} \end{cases}, \tag{1}$$

$$eft_i = est_i + w_i. \tag{2}$$

Note that, since PS is used in this study, the actual start and finish times of a service s_i on a service instance $s_{i,k}$, are denoted as $ast_{i,k}$ and $aft_{i,k}$ can be different from its earliest start and finish times, est_i and eft_i, if one or more service requests are being processed on $s_{i,k}$ and/or if any predecessor of s_i is still being processed at est_i.

Services comprising a particular application are not entirely dedicated or used for that application. Therefore, there might be multiple service requests in which some services are the same (s_2 in Fig. 2).

Services are often dynamically composed together to serve the needs of various applications [2,27]. Cloud computing platforms such as Windows Azure [17] provide convenient tools for service composition. It is therefore important for independent services running in the cloud to be adaptive to the frequent requirement changes from applications due to their dynamic service composition.

Requirements from consumer applications can be characterized by using priorities they expect from the scheduler of a service. For example, as shown in Fig. 2, two applications A_0 and A_1 constitute a set of services, namely s_0, s_1, s_2, s_3, and s_4 to achieve certain functionalities. To eliminate the bottleneck and avoid delays, application A_1 may require service s_2 to give its request high priority. When there is no contention, i.e., application A_0 can tolerate the delay due to low priority, s_2 has no problem handling this. However, when A_0 also requests high priority, a mechanism should be introduced to resolve the contention.

2.3. *Service request scheduling*

The service request scheduling problem addressed in this study is how to assign interdependent services, part of one or more consumer applications, to service instances – that may dynamically be created on demand by providers – aiming to maximize the (net) profit for providers without violating time constraints associated with consumer applications. The scheduling model in this study focuses on multiple service requests from a number of consumers and a set of services offered by a particular provider. The revenue of a provider for a particular time frame is defined as the total of values charged to consumers for processing their applications (service requests) during that time frame. The net profit is then defined as the remainder of the revenue after paying the rental costs—which are associated with the resources on which the service instances of the provider have run and which the provider has had to pay to its cloud infrastructure provider(s). Since a service instance runs constantly once it is created, the provider needs to strike a balance between the number of service instances it creates (and provides), and the service request volume/pattern to ensure its net profit is maximized. The effectiveness of this balancing should be reflected in the average utilization of those service instances.

2.4. *Data service*

The scenario above does not consider the data access of service instances. In a typical datacenter setup, a set of service instances running on a set of hosts accesses data from a set of shared data stores. These data stores can be treated as managed by a data service. Naturally, the workload of the data service may affect the data access time of a service instance, and therefore affect the response of the application served by the service instance. The scenario is shown in Fig. 3.

One may note that we assume a FCFS (First-Come-First-Serve) queue for the data service, which reflects the practice in a storage system design. According to [9], the response time of the data service to a request is roughly linear to the number of outstanding requests in the queue. The performance is also related to the data size, operation type (read or write) as well as the randomness of data access.

We assume that the data service charges the applications it serves separately based on the same utility function defined in Eq. (6). Apparently it is of each application's interest to achieve satisfactory performance while minimizing the cost. The data service intends to maximize its profit. The strategy here is to associate the I/O requests from an application with the value and time decay rate of processing the requests. The data service can then prioritize the requests in the queue based on the value and decay rate. These requests might be processed by different data servers present and running at the data service.

The assignment of values and decay rates differs among applications as well as different services an application's requests flow through, e.g., in Fig. 2, service s_3 may have higher value or decay rate to its I/O requests in attempt to catch up with the processing speed of the corresponding requests flowing through s_1. Naturally, the constraint for the value is that it has to be smaller than the total value of processing the whole request.

3. Pricing model using PS

In this section, we describe the PS scheduling policy and discuss its incorporation into the request-based pricing model for our service request scheduling scenario.

3.1. PS scheduling policy

We consider that a service uses the PS scheduling policy. Under PS, when there are n requests at the service, each of them receives $1/n$ of the service's capacity as shown in Fig. 4. It is assumed that requests from application A_0 and A_1 arrive as a Poisson process with rates λ_0 and λ_1, respectively. For simplicity, these requests come from distribution D and have mean μ_0 if processed by service s_0, and have mean μ_1 if processed by service s_1.

The mean response time for requests served by $s0$ is therefore defined as:

$$t_0 = \frac{1}{\mu_0 - \lambda_0}. \tag{3}$$

Similarly, the mean response time for requests served by $s1$ is defined as:

$$t_1 = \frac{1}{\mu_1 - \lambda_1 - \lambda_0}. \tag{4}$$

Note that, an $M/M/1/PS$ server has the same service rate as its corresponding $M/M/1/FCFS$ server. According to Burke's Theorem, the departure of an $M/M/1$ server is a Poisson process with a rate equal to its arrival rate. As a result, we have an arrival rate λ_0 to service s_1 from service s_0 in Eq. (4). The result also applies to an $M/G/1/PS$ server.

In order to maintain a satisfactory response time, a consumer (for his/her service requests) may pay the service provider to reduce the rate of incoming requests. However, the service provider needs to maintain a certain level of system load to cover the cost of renting resources and gain sufficient profit. We assume that a service provider is charged a flat rate at c per time unit for each instance it runs on the cloud infrastructure. If a request is charged at a rate m, the total revenue during a time unit, denoted by r should be greater than the cost when $\lambda < \mu$, i.e.,

$$r = m\lambda > c. \tag{5}$$

There are incentives for a service provider to add running instances when $\lambda \geq \mu$ and Eq. (5) holds for a new instance.

Furthermore, we assume that each consumer application has an expectation of service response time, i.e., application A expects $t < TMAX$, in which $TMAX$ represents the maximum acceptable mean response time of application A. We denote the average value of finishing processing a request from application A in time t as $v(t)$; obviously, this average value should be greater than the (minimum) price the consumer has to pay. v is negatively related to t. We define a time-varying utility function as below:

$$v(t) = \begin{cases} V, & t \leq TMIN \\ V - \alpha t, & TMIN \leq t \leq TMAX \\ 0, & t > TMAX, \end{cases} \tag{6}$$

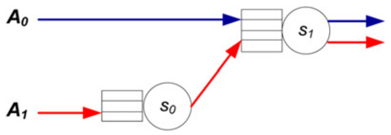

Fig. 4. Scheduling requests using the PS policy.

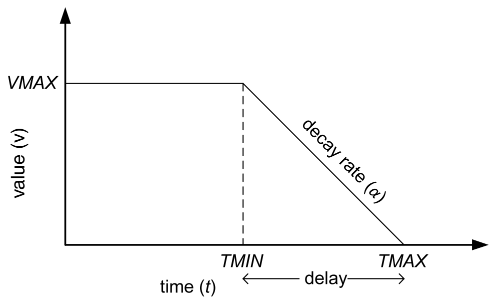

Fig. 5. Time-varying utility function.

where V is the maximum value obtainable from serving a request of consumer application A, $TMIN$ is the minimal time required for serving a request of A, and α is the value decay rate for the mean response time. The V value of a request processing shall be proportional to the service time of that application. The function has similarity to that in [3], however, the way we treat $TMIN$ is different. $TMIN$ is not the mean service time of a request as used in [3], instead it is a *dynamic* value that takes into account of the dependency of processing the request. For example, as shown in Fig. 2, when a request of A_1 is processed by two services (s_1 and s_3) in parallel before converging to another service (s_2), the $TMIN$ values requested by A_1 in s_1 and s_3 are interdependent. The one with shorter service time, say $TMIN_{s1}$can be overridden by the one with longer service time, i.e., $TMIN_{s1} = TMIN_{s2}$ without losing any value as s_2 cannot start processing a request of A_1 before s_1 and s_3 finish processing it. The pictorial description of our utility function is presented in Fig. 5.

Considering λ_{TMAX} is the request arrival rate at the maximal acceptable mean response time, application A requires the following upper bound for λ in order to obtain positive value from serving a request,

$$\lambda \leq \lambda_{TMAX} \leq \mu - \frac{1}{TMAX} \leq \mu - \frac{\alpha}{V}. \tag{7}$$

Combining Eqs. (3), (5) and (6), and assuming $TMIN \leq t \leq TMAX$, we further have the following:

$$V - \frac{\alpha}{\mu - \lambda} \geq m > \frac{c}{\lambda} \tag{8}$$

which gives another constraint of λ as below:

$$\frac{\mu V + c - \alpha - \sqrt{(\alpha - c - \mu V)^2 - 4\mu c V}}{2V} < \lambda < \frac{\mu V + c - \alpha + \sqrt{(\alpha - c - \mu V)^2 - 4\mu c V}}{2V} \tag{9}$$

under the condition:

$$(\alpha - c - \mu V)^2 \geq 4\mu c V. \tag{10}$$

Fig. 6 shows three different services represented by different mean request processing time (μ varies among 30, 40, and 50). When sending requests to these services, different consumers may set different values of V and α in Eq. (8). Fig. 6 shows profitable λ ranges changing with α. Profitable λ ranges with α equals to 5, 10, and 20 are shown as dashed lines for three different services.

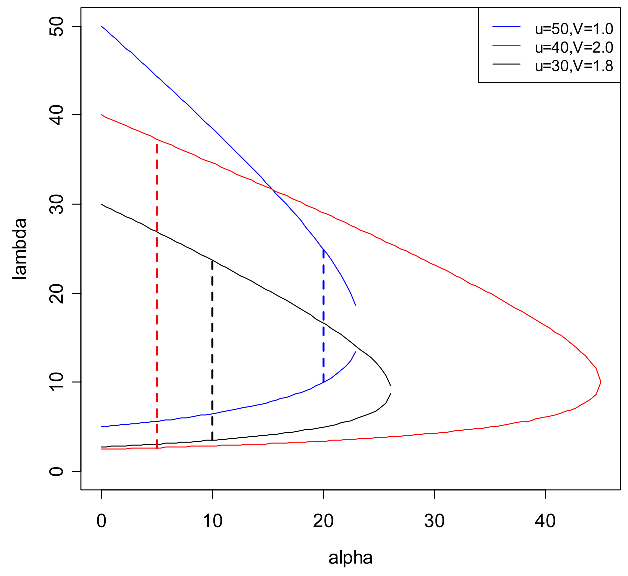

Fig. 6. The upper and lower bounds of profitable request arrival rate ranges for three different consumer applications represented by three different (μ, V) value pairs ($c = 5$). The dashed lines represent the profitable λ ranges of applications with given α values.

As shown in Fig. 6, when $V = 1.8$, $\mu = 30$ and $\alpha = 10$, request arrival rate outside of the black dashed line cannot simultaneously satisfy both the mean response time requirement of the consumer application and the profit needs of the service provider. In this case, the service provider may negotiate with the infrastructure provider to lower c, or raise the cost for request processing. Specifically, in the latter case, V and α associated with the consumer application may be increased and/or reduced, respectively.

3.2. Incorporation of PS into the pricing model

In our approach, we consider that a consumer reaches an SLA with a service provider for each of the applications the consumer outsources. In the SLA for a given application, the consumer specifies V, α and its estimation of λ. The service provider makes a schedule plan according to these pieces of information. Under the PS scheduling scheme, the scheduler mainly plays the role of an admission controller, which controls the request incoming rate of a particular instance. As shown in Fig. 7, the scheduler determines best possible request-instance matches on the basis of performance criteria, such as profit and response time. This match-making process can be programmed with a probability for a given service (Fig. 7(a)) to statically select an appropriate service instance for each request; the probability is determined based on application characteristics (information on composition and service time) and consumer supplied application-specific parameters (α and λ). However, this "static" service-dispatch scheme is not suitable for our dynamic cloud scenario in which new consumers may randomly join and place requests, and some existing consumers may suspend or stop their requests. Thus, scheduling decisions in our algorithms are made dynamically, focusing primarily on the maximization of (net) profit (Fig. 7(b) and (c)).

Under the above mentioned constraints, the optimization goal of the scheduler for a particular time frame is to dispatch requests to a set of service instances in order for the service provider to maximize its profit. More formally, the net profit of a service provider for a given time frame τ is defined as:

$$p^{net} = \sum_{i=1}^{N} v_i - \sum_{j=1}^{L} c\tau_j, \tag{11}$$

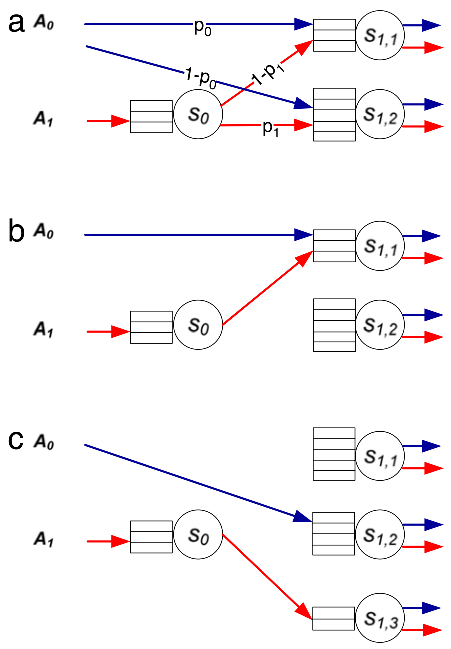

Fig. 7. Scheduling examples based on different criteria. (a) probabilities. (b) utilization and/or profit. (c) net profit.

where N is the total number of requests served, v_i is the value obtained through serving request i, L is the number of service instances run, and τ_i is the actual amount of time s_j has run, if s_j started later than the beginning of time frame τ.

4. Profit-driven service request scheduling

In this section, we begin by characterizing allowable delay for composite service requests and present two profit-driven service request scheduling algorithms with a variant for each of these algorithms. The incorporation of data service into our scheduling scenario is then articulated with a novel prioritization policy for data service requests that we have developed.

4.1. Characterization of allowable delay

A consumer application in this study is associated with two types of allowable delay in its processing, i.e., application-wise allowable delay and service-wise allowable delay (Fig. 8). For a given consumer application A_i, there is a certain additional amount of time that the service provider can afford when processing the application; this application-wise allowable delay is possible due to the fact that the provider will gain some profit as long as A_i is processed within the maximum acceptable mean response time $TMAX_i$. Note that, response time and processing time in our work are interchangeable since the PS scheduling policy adopted in our pricing model does not incur any queuing delay. The minimum/base processing time $TMIN_i$ of application A_i is the summation of processing times of services along the CP of that application. The application-wise allowable delay aad_i of application A_i is therefore:

$$aad_i = TMAX_i - TMIN_i. \tag{12}$$

We denote the actual processing time of A_i as t_i. For a given service s_j, a service-wise allowable delay for s_j may occur when the processing of its earliest start successor service element is bounded by another service; that is, this service-wise delay occurs (see s_1 in Fig. 2) since a service request in this study consists of one or more precedence-constrained services and these services may differ in

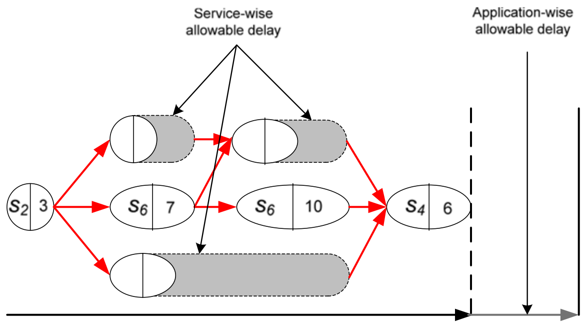
Service-wise allowable delay
Application-wise allowable delay
S2 3
S6 7
S6 10
S4 6

Fig. 8. Allowable delays.

their processing time. For each service in an application, its service-wise allowable delay time is calculated based on its actual latest start and finish times. The actual latest start and finish times of a service s_j are defined as:

$$alst_j = alft_j - w_j \tag{13}$$

$$alft_j = \begin{cases} aft_j & \text{if } s_j = s_{exit} \\ \min\limits_{s_k \in succ(s_j)} (alst_k) & \text{otherwise,} \end{cases} \tag{14}$$

where $succ(s_j)$ is the set of immediate successor services of s_j.

The service-wise allowable delay time sad_j of service s_j is defined as:

$$sad_j = alft_j - aft_j. \tag{15}$$

Based on Eqs. (12) and (14), we can derive two more service-wise allowable delay metrics, aggregative and cumulative, and for a given service s_j in application A_i, they are defined as:

$$asad_j = sad_j + aad_i \frac{(sad_j + w_j)}{TMIN_i} \tag{16}$$

$$csad_j = asad_j + w_{MIP_j} + asad_{MIP_j} - t_{MIP_j}, \tag{17}$$

where w_{MIP_j} is the processing time of MIP service of s_j, t_{MIP_j} is the actual processing time used for MIP service of s_j. These two metrics directly correlate with profit. Particularly, the cumulative service-wise allowable delay time $csad_j$ of service s_j in addition to its original processing time (w_j) indicates the upper bound processing time of s_j without loss/deficit.

4.2. Maximum profit algorithm

MaxProfit (Fig. 9) takes into account not only the profit achievable from the current service, but also the profit from other services being processed on the same service instance. Specifically, the service is assigned only if the assignment of that service onto the service instance yields some additional profit. Note that, each service is associated with a decay rate and therefore, the assignment of a service to a service instance on which some other services of the same kind are being processed may result in a certain degree of profit loss.

The *MaxProfit* algorithm is activated by a service when any one of the following events occurs:

1. A new application requests to use the service.
2. An existing application notifies that it will no longer use this service.
3. Any requirement change of existing applications, such as the change in mean request arrival rate, decay rate or maximum request value.

As the changes happen frequently in a dynamic environment, *MaxProfit* does not intend to produce optimal schedule for a particular set of parameters, but to run quickly to allocate VM instances for a service to optimize profit in a greedy manner.

```
1. Let $max\_pi = \emptyset$
2. Let $s_{j,*} = \emptyset$
3. Let $s_j^* =$ the first service to be scheduled
4. for $\forall s_{j,k} \in I_j$ do
5.    Let $pi_k = \emptyset$
6.    Let $pi_k^* = \emptyset$
7.    for $\forall s_{j,k}^l$ running on $s_{j,k}$ do
8.       Let $aft_{j,k}^l = aft$ of $s_{j,k}^l$ without considering $s_j^*$
9.       Let $aft_{j,k}^{l*} = aft$ of $s_{j,k}^l$ with considering $s_j^*$
10.      Let $clft_{j,k}^l = aft_{j,k}^l + asad_{j,k}^l$
11.      if $aft_{j,k}^{l*} > clft_{j,k}^l$ then // possible loss
12.         Go to Step 4
13.      end if
14.      Let $pi_k = pi_k + clft_{j,k}^l - aft_{j,k}^l$
15.      Let $pi_k^* = pi_k^* + clft_{j,k}^l - aft_{j,k}^{l*}$
16.   end for
17.   Let $pi_k^* = pi_k^* + clft_{j,k}^* - aft_{j,k}^*$      // include $s_j^*$
18.   if $pi_k^* > pi_k$ then
19.      Let $\Delta pi_k = pi_k^* - pi_k$
20.      if $\Delta pi_k > max\_pi$ then
21.         Let $max\_pi = \Delta pi_k$
22.         Let $s_{j,*} = s_{j,k}$
23.      end if
24.   end if
25. end for
26. if $s_{j,*} = \emptyset$ then
27.    Create a new service instance $s_{j,new}$
28.    Let $s_{j,*} = s_{j,new}$
29. end if
30. Assign $s_j^*$ to $s_{j,*}$
```

Fig. 9. The *MaxProfit* algorithm.

MaxProfit maintains a service queue containing yet-to-be-dispatched services according to their precedence constraints; that is, when a new application arrives, its entry service is the only one to be processed. *MaxProfit* checks all service instances (the outer *for* loop; Steps 4–25) of service $s_j(I_j)$—for the current ready service (s_j^*)—and selects the best instance based on additional profit incurred by s_j^* (Step 19). At the core, for each service running the current service instance (Step 7), the profit difference between the two schedules (one with considering s_j^* and the other one without considering s_j^*) is computed, and this is denoted as profit index (or *pi*). The statement "considering s_j^*" (in Step 9) should be interpreted as the consideration of the assignment of s_j^* to the current service instance. In Steps 14 and 15, profit indices for those two schedules are computed using the current latest finish time $clft_{j,k}^l$ of each service (Step 10). Note that, the current latest finish time $(clft_x)$ of a running service s_x may be different from the actual latest finish time in that $clft_x$ is computed based on the current schedule and there might be new services assigned to the current instance before s_x completes its processing. $clft_x$ can be seen as an intermediate measure of $alft_x$. The current service instance is disregarded (Step 11) if the actual finish time of any of the running services considering s_j^* ($aft_{j,k}^{l*}$) is greater than its current latest finish time ($clft_{j,k}^l$), since this implies a possible loss in profit. The profit index with considering s_j^* should include the profit index value of s_j^* (Step 17). After each iteration of the inner *for* loop (from Step 7 to Step 16), *MaxProfit* checks if the current instance delivers the largest profit increase (Step 20) and keeps track of the best instance (Steps 21 and 22). If none of the current instances is selected (i.e., no profit gain is possible with s_j^*), a new instance is created (Step 27). The final assignment of s_j^* is then carried out in Step 30 onto the best instance$s_{j,*}$.

While the "asad metric" attempts to ensure maximum profit gain by focusing more on each individual service in an application, the "csad metric" tries to maximize the net profit by balancing revenue and utilization. In other words, ensuring high service-instance utilization tends to avoid the creation of new instances resulting in minimizing resource rental costs. Based on this balancing fact, we have devised a variant of *MaxProfit* using the *csad* metric (i.e., $MaxProfit^{csad}$) instead of the *asad* metric used in Step 10.

4.3. Maximum utilization algorithm

The main focus of *MaxUtil* (Fig. 10) is on the maximization of service-instance utilization. This approach is an indirect way of reducing costs to rent resources – or to increase net profit – and it also implies a decrease in the number of instances the service provider creates/prepares. For a given service, *MaxUtil* selects the instance with the lowest utilization. Although scheduling decisions are made based primarily on utilization, *MaxUtil* also takes into account profit incorporating *alft* into its scheduling. Specifically, *alft* compliance would ensure the avoidance of deficit.

MaxUtil is triggered by the same conditions that activate *MaxProfit*. As in *MaxProfit*, the service with the earliest start time, in the service queue maintained by *MaxUtil*, is selected for scheduling (Step 3). A similar scheduling routine to that of *MaxProfit* can also be identified—from Step 4 to Step 11 where each instance for the selected service s_j^* is checked if it can accommodate s_j^* without incurring loss; hence, that instance is a candidate. For each candidate instance, its utilization is calculated and the instance with the minimum utilization is identified (Steps 13–17). The utilization $util_{j,k}$ of a service instance $s_{j,k}$ is defined as:

$$util_{j,k} = \frac{\tau^{used}}{(\tau^{cur} - \tau^{start})}, \tag{18}$$

where τ^{used}, τ^{cur} and τ^{start} are the amount of time used for processing services, the current time, and the start/creation time of $s_{j,k}$, respectively. The *if* statement (Steps 19–22) ensures the processing of s_j^* creating a new instance in the case of absence of profitable service-instance assignment. A variant of *MaxUtil* (i.e., $MaxUtil^{csad}$) is also devised by incorporating the *csad* metric.

4.4. Prioritization policy for data access requests

Data access requests are queued up in a 'global' queue (gQ) and dispatched to different data servers for actual processing. In essence, the request with the highest priority in gQ is selected and dispatched to the first available data server at the data service. In our scenario values (and decay rates) are attached to data accesses, and thus, the prioritization should be carried out in the way the profit of data service is maximized. The same utility function (Eq. (6)) used for service requests is adopted to calculate the profit since data accesses also exhibit similar time-varying characteristics; that is, the value and decay rate of a data access request are primarily determined by the urgency of processing that request.

```
1.  Let $min\_util = 1.0$
2.  Let $s_{j,*} = \emptyset$
3.  Let $s_j^*$ = the first service to be scheduled
4.  for $\forall s_{j,k} \in I_j$ do
5.    for $\forall s^l_{j,k}$ running on $s_{j,k}$ do
6.      Let $aft^l_{j,k}$ = $aft$ of $s^l_{j,k}$ without considering $s_j^*$
7.      Let $aft^{l*}_{j,k}$ = $aft$ of $s^l_{j,k}$ with considering $s_j^*$
8.      Let $clft^l_{j,k} = aft^l_{j,k} + asad^l_{j,k}$
9.      if $aft^{l*}_{j,k} > clft^l_{j,k}$ then // possible loss
10.       Go to Step 4
11.     end if
12.   end for
13.   Let $util_{j,k}$ = utilization of $s_{j,k}$
14.   if $util_{j,k} < min\_util$ then
15.     Let $min\_util = util_{j,k}$
16.     Let $s_{j,*} = s_{j,k}$
17.   end if
18. end for
19. if $s_{j,*} = \emptyset$ then
20.   Create a new service instance $s_{j,new}$
21.   Let $s_{j,*} = s_{j,new}$
22. end if
23. Assign $s_j^*$ to $s_{j,*}$
```

Fig. 10. The *MaxUtil* algorithm.

We present a novel profit-driven prioritization policy (*MaxLoss*), which effectively identifies the most profitable data access request in gQ. This identification is determined based on the balance between the throughput of data servers and the urgency (decay rate) of data access requests. *MaxLoss* calculates the loss per unit-time delay (for each data access request in gQ) is calculated taking into account its data access time (or size). For a given data access, the unit-time profit is defined as its profit (estimated at the time of a prioritization event using our utility function) divided by data access time. Then, the loss per unit-time delay is defined as the loss incurring from delaying the dispatch of that data access by one time unit; i.e., at a given time t, the loss per unit-time delay is defined as $profit_t/size - profit_{t+1}/size$. Since the loss per unit-time delay indicates the criticality (or urgency) of that request in terms of profitability, *MaxLoss* selects the request with the highest loss per unit-time delay.

The policy may be overridden if a request in gQ is unable to be processed by its latest completion time in other data servers than the currently available server. This situation may occur because its priority has been lowered one or more times resulting from the low (possibly improper) valuation of the request.

5. Experimental evaluation

Our algorithms are evaluated on the basis of comparisons with reference algorithms using three performance metrics (profit rate, utilization and response rate). Separate comparative studies were conducted for service request scheduling algorithms and data access prioritization policies, respectively. Specifically, two sets of our algorithms (*MaxProfit* and $MaxProfit^{csad}$, and *MaxUtil* and $MaxUtil^{csad}$) and a non PS-based algorithm (called earliest finish time with profit taken into account or EFT^{profit}) are used in the former comparative study. EFT^{profit} that we have implemented as a reference algorithm is used for our evaluation study, since none of the existing algorithms can be directly comparable to our algorithms. As the name (*EFT*^{profit}) implies, for a given service, the algorithm selects the service instance that can finish the processing of that service the earliest with a certain amount of profit. If none of the current service instances is able to process that service without loss, a new instance of that service is created.

For the latter comparative study, our data access prioritization policy (*MaxLoss*) is compared with three reference algorithms, *FCFS*, *MinSize* and *MaxSize*. These reference algorithms are self-explanatory as their dispatch decision is made based on simply arriving order, minimum and maximum request size, respectively. FCFS is commonly used in practice. The main indicators we use for comparison are the profit realized via different policies and the distribution of response time over request values.

5.1. Experimental settings

The performance of our algorithms was thoroughly evaluated using our discrete-event cloud simulator developed in C/C++. Simulations were carried out with a diverse set of applications (i.e., various application characteristics) and settings. Due to the large scale constitution of cloud computing environments, the number of service instances (that can be created) is assumed to be unbounded.

The total number of experiments carried out to evaluate service request scheduling algorithms alone is 105,000. Specifically, 21,000 distinctive experiments were prepared and each of these applied to five algorithms. Those 21,000 experimental settings were generated randomly using parameters as follows:

- maximum width: {2, 4, 8, 16, 32, 64}
- number of services per application: U(10, 80), and
- simulation duration {2000, 4000, 8000, 12,000, 16,000, 20,000, 30,000}.

Now we describe and discuss the generation of parameters associated with service requests including maximum values, maximum acceptable mean response times, decay rates and arrival rates. The lower bound of maximum value V_i^{lower} of an application A_i consisting of n services in our evaluation study is generated based on the following:

$$V_i^{lower} = \sum_{j=1}^{n} w_j u, \tag{19}$$

where u is a unit charge set by the provider. u is defined as ce where e is an extra charge rate set again by the provider. Due to constant resource rental costs for service instances regardless of usage state (whether they are processing services or not), the provider needs to identify/calculate (extra) charges applied to consumer applications. In our experiments, e is set to 2.0, i.e., twice as much as the resource rental cost c. The maximum value of an application and/or u in practice may be negotiated between the consumer and provider. While V_i^{lower} computed using Eq. (19) is an accurate estimate, it should be more realistically modelled incorporating a certain degree of extra value; this is mainly because the absolute deadline of an application (*TMIN*) cannot be practically met for various reasons including the use of PS in our study. The extra value V_i^{extra} of A_i is defined as:

$$V_i^{extra} = TMIN_i d_i u, \tag{20}$$

where d_i is an extra delay rate the consumer/application can afford for A_i. Now the actual maximum value of A_i is defined as:

$$V_i^{act} = V_i^{lower} + V_i^{extra}. \tag{21}$$

The decay rate α_i of an application A_i is negatively related to t_i (more specifically, $t_i - TMIN_i$) and defined as:

$$\alpha_i = \frac{V_i^{extra} + V_i^{lower}\,(1 - 1/(1+e))}{TMIN_i d_i}. \tag{22}$$

Arrival times—for repeatedly requested applications (λ) and newly requested applications—are generated on a Poisson process with mean values randomly generated from a uniform distribution.

Since a data access request is part of a service request in a given application and the same utility function (Eq. (6)) is adopted, most of experimental settings described above remain the same. Additional parameters used due to the incorporation of data service are the number of data servers, and the data access time, allowable delay, maximum value and decay rate for each data access request. Similar to those for service requests, data access request parameters (except the number of data servers) are randomly generated from a uniform distribution. For a given data access request the upper bound of its access time plus allowable delay is the processing time of the service that the data service is associated with. On the other hand, the lower bound is set to zero; that is, some services in an application may not require data access. Three different numbers of data servers are used (i.e., 20, 30 and 40). The initial state of data servers in our simulations is set to busy serving data access requests; that is, each simulation starts with gQ being saturated with requests. Those earlier 105,000 experiments were repeated for three more times with these three numbers of data servers.

5.2. Performance metrics

Experimental results are plotted, analysed and discussed using three performance metrics, the average net profit rate, the average utilization, and the average response rate. The average net profit rate pr^{net} is computed by the actual net profit p^{net} divided by the maximum possible net profit p^{max}. More formally,

$$p^{\max} = \sum_{i=1}^{N}\left(V_i^{act} - c\sum_{j=1}^{n} w_j\right) \tag{23}$$

$$pr^{net} = \frac{p^{net}}{p^{\max}}. \tag{24}$$

The average utilization is defined as:

$$\overline{util} = \frac{\sum_{j=1}^{L} util_j}{L}. \tag{25}$$

The response rate rr_i of an application A_i is defined as:

$$rr_i = \frac{TMIN_i}{t_i}. \tag{26}$$

Then the average response rate for all applications serviced by the provider is defined as:

$$\overline{rr} = \frac{\sum_{i=1}^{N} rr_i}{N}. \tag{27}$$

6. Results

In this section, we present and discuss performance evaluation results obtained from an extensive set of simulations.

6.1. Profit-driven service request scheduling

The results obtained from our simulations for profit-driven service request scheduling are summarized in Table 1, followed by results (Fig. 11) based on each of those three performance metrics. Clearly, the significance of our explicit and intuitive incorporation of profit into scheduling decisions is verified in these results. That is, all our algorithms achieved utilization above 50% with compelling average net profit rates reaching up to 57%. Note that

Table 1
Overall comparative results.

Algorithm	Avg. net profit (%)	Avg. utilization (%)	Avg response rate (%)
EFT^{profit}	36	32	100
MaxUtil	41	56	68
MaxUtil^{csad}	32	58	61
MaxProfit	57	52	84
MaxProfit^{csad}	44	55	72

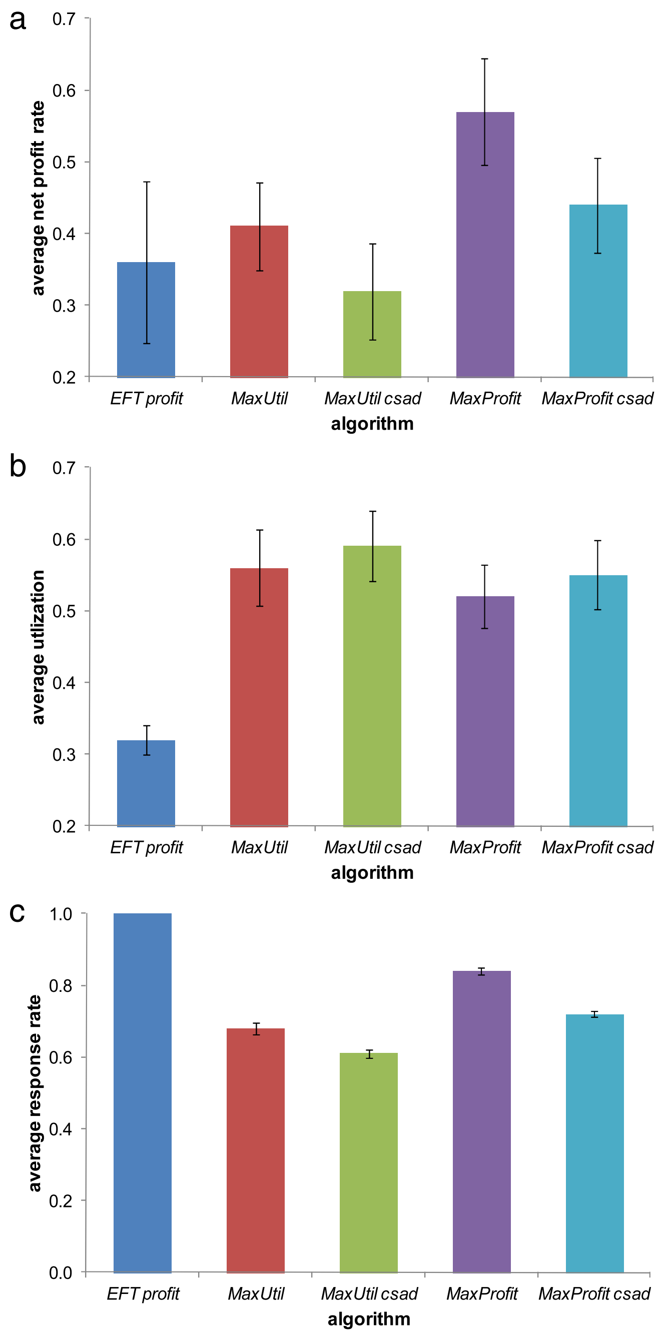

Fig. 11. Performance with respect to different metrics. (a) avg. profit rate. (b) avg. utilization. (c) avg. response rate.

in spite of using the same algorithms of ours, these results are different from those presented in our previous paper [15] due to introduction of dynamic composition of applications. Although the incorporation of the *csad* metric improves utilization by 3% on average, the profit—that those variants with the *csad* metric gained—is not very appealing, 11% lower on average compared with the profit gained by *MaxProfit* and *MaxUtil*. This lower profit gain can be explained by the fact that increases in utilization are enabled by the allowance of additional delay times (i.e., *csad* times) accommodating more services resulting in lower response rate.

While utilization has a significant impact on profit, response rate is equally influential to profitability particularly in our market-based scheduling scenario. Fig. 11 shows that our algorithms effectively deal with response rate balancing those two performance goals (utilization and response rate). Therefore, profit is strongly influenced by utilization and response time collectively, not independently. That is, the minimal response rate that *EFT*^{profit} delivers alone is no good impact on the improvement of profit since that compelling performance (response rate of 1.0) is only achieved by creating more service instances resulting in poor utilization. Thus, the utilization is the primary determinant on profit in case of *EFT*^{profit}; however, it fluctuates significantly by service diversity and the parallelism of applications. This characteristic can be clearly identified with those error bars in Fig. 11.

On average, the *MaxProfit* suite and the *MaxUtil* suite outperformed *EFT*^{profit} by 40% and 10% in terms of net profit rate, respectively, and 80% and 67% in terms of utilization. Since *EFT*^{profit} does not adopt PS and it tends to create a larger number of instances (compared with those created by our algorithms) to avoid deficit, the average response rate is constant (i.e., 1.0). This creation of (often) an excessive number of instances results in low utilization and in turn low net profit rate.

It is identified that our utilization-centric algorithms performance compared with those of our profit-centric algorithms (*MaxProfit* and *MaxProfit*^{csad}), because the particular focus on utilization in the former set has an adverse effect on response rate.

6.2. Data access prioritization

Results shown in Figs. 12–14 clearly demonstrate the advantage of profit-driven data access prioritization. Specifically, the effectiveness of incorporating values and data access sizes into the prioritization of data accesses in our algorithms is well reflected on response rates (Figs. 12 and 13), and this in turn results in good profit rates (Fig. 14). On average, the average profit rate of *MaxLoss* is 67%, 13% and 40% points (or 515%, 19% and 100%) higher than *FCFS*, *MinSize* and *MaxSize* respectively.

Since *FCFS* dispatches data accesses without considering either value or data access size most requests suffer from long delay (waiting time) in their processing (Figs. 12(a) and 13(a)); the result of this simple dispatching leads to the poor profit rate. In the meantime, *MinSize* and *MaxSize* consider data access size alone in their prioritization. This incorporation of data access size improves response rate to a certain degree (Figs. 12(b), (c), 13(b) and (c)). Due to their profit indifferent nature, many high-value data accesses are assigned low priority resulting in excessive queuing delay (low response rates). Apparently, *MinSize* with its prioritization of small data accesses tends to avoid some excessive queuing delay compared to *MaxSize*.

Unlike these profit-indifferent policies, our profit-driven algorithm differentiates high-value data accesses from relatively low-value ones in its prioritization. As a result, response rates they deliver have a linear relationship with values (Figs. 12(d) and 13(d)). Clearly, profit rates are closely related to the degree of slope of this linearity.

We have conducted a one-off experiment with a sample set of data accesses filled in gQ and results of this experiment are presented in Fig. 13 in order to highlight that linearity. Specifically,

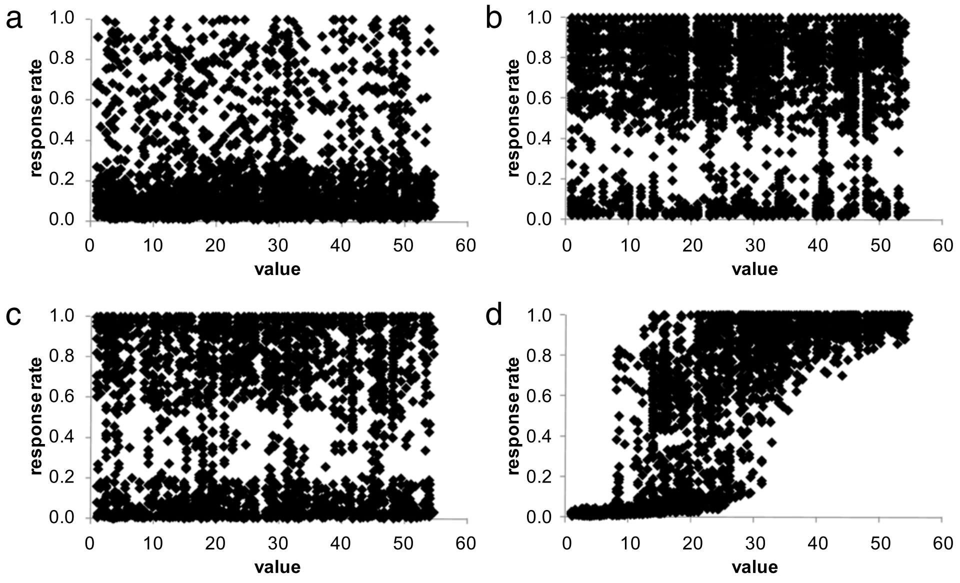

Fig. 12. Correlation between response rates and values. (a) *FCFS*. (b) *MinSize*. (c) *MaxSize*. (d) *MaxLoss*.

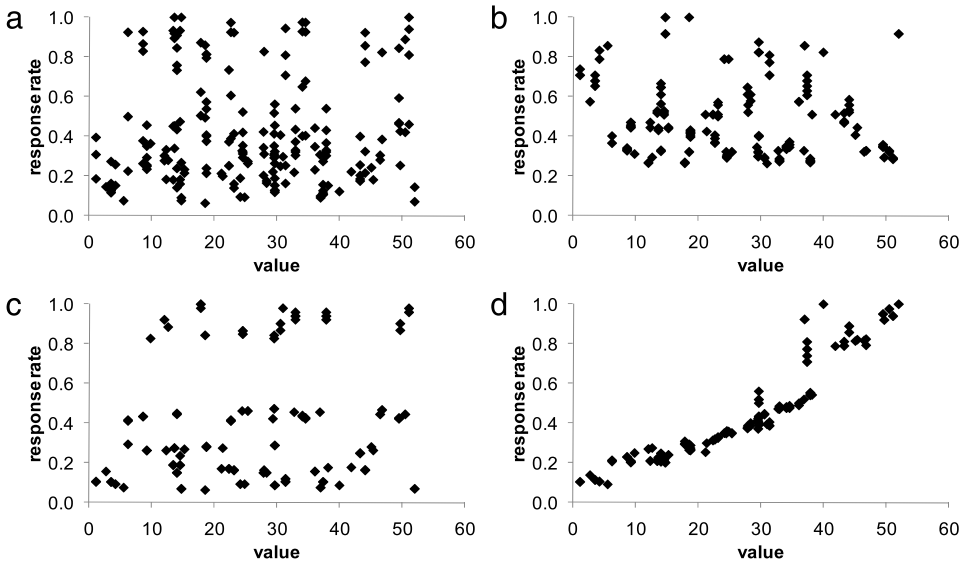

Fig. 13. Correlation between response rates and values with a sample set of data accesses. (a) *FCFS*. (b) *MinSize*. (c) *MaxSize*. (d) *MaxLoss*.

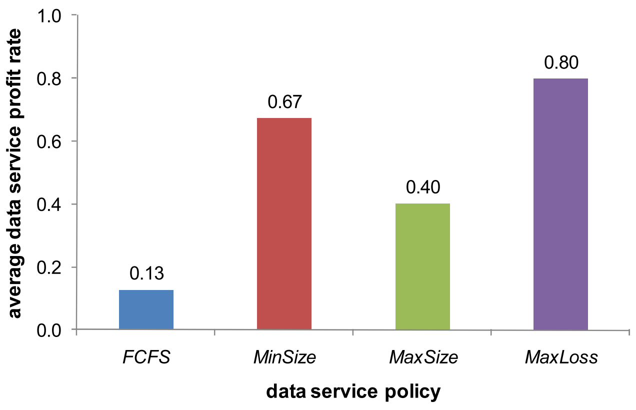

Fig. 14. Overall data service profit rate.

a gQ with a size of 200 is packed with data access requests arrived at the same time and dispatched by 25 data servers using four different prioritization policies. The advantage of our profit-driven policy is much clearly seen in Fig. 13(d).

7. Related work

Research on market-based resource allocation started from 1981 [26], and possibly earlier. The tools offered by microeconomics for addressing decentralization, competition and pricing are thought useful in handling computing resource allocation problem [7]. Even though some market-based resource allocation methods are non-pricing-based [8,26,12], pricing-based methods can reveal the true needs of users who compete for shared resources and allocate resources more efficiently [13]. The application of market-based resource allocation ranges from computer networking [26], distributed file systems [12], distributed database [22] to computational job scheduling problems [4,5,25]. Our work is related to pricing-based computational job scheduling, or utility computing [20].

Chun and Culler [4] build a prototype cluster that provides a market for time-shared CPU usage for various jobs. Coleman et al. [5] use the market-based method to address flash crowds and traffic spikes for clusters hosting Internet applications. Libra [21] is a scheduler built on proportional resource share clusters. One thing in common for [4,5,21] is that the value of a job does not change with the processing time. The work in [3] introduced time-varying resource valuation for jobs submitted to a cluster. The changing values were used for prioritizing and scheduling batch sequential and parallel jobs. The job with the highest value per CPU time unit is put ahead of the queue to run. Irwin et al. [10] extended the time-varying resource valuation function to take account of penalty when the value was not realized. The optimization therefore also minimizes the loss due to penalty. In our model, the penalty is not directly reflected in our time-varying valuation function, but it is implicitly reflected in the cost of physical resource usage in the cloud. The cost incurs even when no revenue is generated.

Popovici and Wilkes [19] consider a service provider rents resources at a price, which is similar to the scenario we deal with in cloud computing. The difference is that resource availability is uncertain in [19] while resource availability is often guaranteed by the infrastructure provider in the cloud. The scheduling algorithm (FirstProfit) proposed in [19] uses a priority queue to maximize the per-profit for each job independently. Our work differs from [19] in the queuing model. As a consequence, profit maximization is not based on a single job, but on all the concurrent jobs in the same PS queue of a service instance.

The proportional-share allocation is commonly used in market-based resource allocation [14,6], in which a user submits bids for different resources and receives a fraction of each resource equal to his bid divided by the sum of all the bids submitted for that resource. It is different to our processor-sharing (PS) allocator, which allows each admitted request to have an equal share of the service capacity. Proportional-share requires all jobs to have pre-defined weights in order to calculate the fraction of resource to allocate. This works well for batch jobs in a small cluster, but has limitations for a service running in the cloud. The elastic resource pool and requests from dynamic client applications make weight-assignment a non-trivial task. Our processor-sharing admission control is capable of accommodating dynamic applications and ensures that random requests do not interfere with the processing of requests that carry certain values for an application and a service provider.

Tsakalozos et al. [23] adopt microeconomics to deal with the balance between provider's profit and user satisfaction in terms of application performance (more specifically, value for money). Since in [23] data-intensive applications using the MapReduce framework are the focus and the performance of these applications is heavily dependent on the number of worker nodes, the mechanism in [23] dynamically adjusts the number of worker nodes (VMs). Adjustment decisions are made taking into account the user's budget and the performance variation primarily caused by resource sharing with other VMs.

Khan and Ahmad et al. [11] investigate three game theoretic approaches for resource allocation in computational grids. While these game-based approaches provide an effective means to deal with deadline-constrained tasks using multiple resources their application is not directly realized in our study with precedence-constrained workflow applications and explicit data services.

Our method is unique in the following aspects:

1. The utility function is time-varying and dependency aware. The latter is important for the cloud where mashup services are common.
2. Our model allows new service instances to be added dynamically and consistently evaluates the profit of adding a new instance, while most previous work deal with resource pools of fixed size.
3. Data service takes into account profitability by prioritizing data access requests using our utility function.

8. Conclusion

In this paper, we presented algorithms for scheduling service requests and prioritizing their data accesses in the cloud with the main objective of maximizing profit. We adopted a three-tier cloud structure, which is a representative cloud system model. The cost effectiveness is interpreted differently between those three parities in this cloud structure; hence, their objectives conflict with each other. To this end, we have taken into account specific characteristics of virtualized cloud system and composite-service applications. In other words, the dynamic creation of service instances, processor-sharing, and time-varying/dependency-aware utility function are incorporated into our algorithms. In addition, service request scheduling algorithms exploit two allowable delay metrics (*asad* and *csad*). We have demonstrated the efficacy of such an approach for consumer applications with interdependent services. It is identified that those two allowable delay metrics enable effective exploitation of characteristics of precedence-constrained applications. Furthermore, we have introduced data service and associated data accesses with services for more realistic settings. Clearly, the prioritization of data accesses considering the profitability of data service plays a key role in maximizing profit. Our evaluation results confidently confirm these claims together with the promising performance of our algorithms.

Acknowledgement

This work is supported in part by an Australian Research Council Grant DP1097111.

References

[1] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R.H. Katz, A. Konwinski, G. Lee, D.A. Patterson, A. Rabkin, I. Stoica, M. Zahariam, Above the clouds: a berkeley view of cloud computing, Technical report UCB/EECS-2009-28, Electrical Engineering and Computer Sciences, University of California at Berkeley, USA, February 2009.
[2] D. Benslimane, S. Dustdar, A. Sheth, Services mashups: the new generation of web applications, IEEE Internet Computing 12 (5) (2008) 13–15.
[3] B.N. Chun, D.E. Culler, Market-based proportional resource sharing for clusters, Technical Report CSD-1092, University of California at Berkeley, January 2000.
[4] B.N. Chun, D.E. Culler, User-centric performance analysis of market-based cluster batch schedulers, Proc. IEEE/ACM Int'l Symp. Cluster Computing and the Grid, pp. 30–38, May 2002.
[5] K. Coleman, J. Norris, G. Candea, A. Fox, OnCall: defeating spikes with a free-market application cluster, Proc. the IEEE Conf. Autonomic Computing, 2004.
[6] M. Feldman, K. Lai, L. Zhang, The proportional-share allocation market for computational resources, IEEE Transactions on Parallel and Distributed Systems 20 (8) (2009).
[7] D.F. Ferguson, The application of microeconomics to the design of resource allocation and control algorithms, Ph.D. Thesis, Columbia University, 1989.
[8] D.F. Ferguson, C. Nikolaou, J. Sairamesh, Y. Yemini, Economic models for allocating resources in computer systems, in: S.H. Clearwater (Ed.), Market-Based Control: A Paradigm for Distributed Resource Allocation, World Scientific, 1996.
[9] A. Gulati, C. Kumar, I. Ahmad, K. Kumar, BASIL: automated IO load balancing across storage devices, Proc. 8th USENIX Conference on File and Storage Technologie (FAST 2010), 2010.
[10] D.E. Irwin, L.E. Grit, J.S. Chase, Balancing risk and reward in a market-based task service, Proc. IEEE Symp. High Performance Distributed Computing, pp. 160–169, 2004.
[11] S.U. Khan, I. Ahmad, Non-cooperative, semi-cooperative, and cooperative games-based grid resource allocation, Proc. International Parallel and Distributed Processing Symposium, 2006.
[12] J.F. Kurose, R. Simha, A microeconomic approach to optimal resource allocation in distributed computer systems, IEEE Transactions on Computers 38 (5) (1989).
[13] K. Lai, Markets are dead, long live markets, Sigecom Exchanges 5 (4) (2005) 1–10.
[14] K. Lai, L. Rasmusson, E. Adar, S. Sorkin, L. Zhang, B.A. Huberman, Tycoon: an implementation of a distributed, market-based resource allocation system, Multiagent and Grid Systems 1 (3) (2005) 169–182.
[15] Y.C. Lee, C. Wang, A.Y. Zomaya, B.B. Zhou, Profit-driven service request scheduling in clouds, Proc. IEEE/ACM Conf. Cluster, Cloud and Grid Computing (CCGrid 2010), pp. 15–24, 2010.
[16] Y.C. Lee, A.Y. Zomaya, Energy efficient utilization of resources in cloud computing systems, Journal of Supercomputing, (in press).

[17] Microsoft, Windows Azure AppFabric, available at: http://www.microsoft.com/windowsazure/appfabric/teched/, 2011.
[18] D. Parkhill, The Challenge of The Computer Utility, Addison-Wesley Educational Publishers Inc, US, 1966.
[19] F.I. Popovici, J. Wilkes, Profitable services in an uncertain world, Proc. the ACM/IEEE SC2005 Conf. High Performance Networking and Computing (SC 2005), 2005.
[20] M.A. Rappa, The utility business model and the future of computing services, IBM Systems Journal 43 (1) (2004) 32–42.
[21] J. Sherwani, N. Ali, N. Lotia, Z. Hayat, R. Buyya, Libra: a computational economy-based job scheduling system for clusters, Software Practice and Experience 34 (2004) 573–590.
[22] M. Stonebraker, P.M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, A. Yu, Mariposa: a wide-area distributed database system, The VLDB Journal 5 (1996) 48–63.
[23] K. Tsakalozos, H. Kllapi, E. Sitaridi, M. Roussopoulos, D. Paparas, A. Delis, Flexible use of cloud resources through profit maximization and price discrimination, in: Proc. IEEE Int. Conf. on Data Engineering, ICDE 11, April 2011.
[24] L.M. Vaquero, L. Rodero-Merino, J. Caceres, M. Lindner, A break in the clouds: towards a cloud definition, ACM SIGCOMM Computer Communication Review 39 (1) (2009) 50–55.
[25] C.A. Waldspurger, T. Hogg, B.A. Huberman, J.O. Kephart, W.S. Stornetta, Spawn: a distributed computational economy, Software Engineering 18 (2) (1992) 103–117.
[26] Y. Yemini, Selfish optimization in computer networks, Proc. the 20th IEEE Conf. Decision and Control, pp. 281–285, 1981.
[27] Q. Zhao, G. Huang, J. Huang, X. Liu, H. Mei, A web-based mashup environment for on-the-fly service composition, Proc. IEEE International Symposium on Service-Oriented System Engineering, pp. 32–37, 2008.

Young Choon Lee received the B.Sc. (hons) degree in 2003 and the Ph.D. degree from the School of Information Technologies at the University of Sydney in 2008. He is currently a post doctoral research fellow in the Centre for Distributed and High Performance Computing, School of Information Technologies. His current research interests include scheduling and resource allocation for distributed computing systems, nature-inspired techniques, and parallel and distributed algorithms. He is a member of the IEEE and the IEEE Computer Society.

Chen Wang received his Ph.D. degree from Nanjing University. He is a senior research scientist at CSIRO ICT Centre, Australia. His research interests are primarily in distributed, parallel and trustworthy systems. His current work focuses on accountable distributed systems, resource management in cloud computing and the smart grid.

Albert Y. Zomaya is currently the *Chair Professor of High Performance Computing & Networking* and *Australian Research Council Professorial Fellow* in the School of Information Technologies, The University of Sydney. He is also the *Director of the Centre for Distributed and High Performance Computing* which was established in late 2009. Prof. Zomaya is the author/co-author of seven books, more than 400 papers, and the editor of nine books and 11 conference proceedings. He is the Editor in Chief of the *IEEE Transactions on Computers* and serves as an associate editor for 19 leading journals. Prof. Zomaya is the recipient of the*Meritorious Service Award* (in 2000) and the *Golden Core Recognition* (in 2006), both from the *IEEE Computer Society*. Also, he received the *IEEE TCPP Outstanding Service Award* and the *IEEE TCSC Medal for Excellence in Scalable Computing*, both in 2011. Prof. Zomaya is a Chartered Engineer, a Fellow of AAAS, IEEE, IET (UK), and a Distinguished Engineer of the ACM.

Bing Bing Zhou received the B.S. degree from Nanjing Institute of Technology, China and the Ph.D. degree in Computer Science from Australian National University. He is currently an associate professor at the University of Sydney. His research interests include parallel/distributed computing, Grid and cloud computing, peer-to-peer systems, parallel algorithms, and bioinformatics. He has a number of publications in leading international journals and conference proceedings. His research has been funded by the Australian Research Council through several Discovery Project grants.