2012 IEEE Eighth World Congress on Services

Just Satisfactory Resource Provisioning for Parallel Applications in the Cloud

Chen Wang
CSIRO ICT Centre
PO Box 76, Epping, NSW 1710, Australia
Email: chen.wang@csiro.au

Junliang Chen, Bing Bing Zhou, and Albert Y. Zomaya
Centre for Distributed and High Performance Computing
The University of Sydney, NSW 2006, Australia
Email: jchen@it.usyd.edu.au, {bing.zhou, albert.zomaya}@sydney.edu.au

***Abstract*—The paper discusses a resource management method at the cloud tenant level. It concerns efficiently running a profitable service on resources leased from a cloud infrastructure provider. Particularly, the paper focuses on services that handle parallelizable client requests, e.g, such a request can be processed using a MPI or a MapReduce program. The objective is to manage resource use to just satisfy the performance requirements of clients and avoid the common over-provisioning or under-provisioning problem. The proposed resource management method makes initial resource leasing plan based on client request profile and performance targets in service level agreements (SLAs). It then dynamically adjusts resource allocation based on monitoring data. Our extensive experiments show that the method is able to provision and efficiently use resources to just satisfy the SLA targets.**

I. Introduction

The cloud computing paradigm promises a cost effective computing platform for a variety of business services. This is mainly achieved through efficient resource management techniques at the *cloud infrastructure provider level*. At this level, physical resources are encapsulated into virtual resources to fit various needs of cloud tenants on demand. Virtualization greatly improves the utilization of physical resources. Virtual resources are priced according to market conditions. Normally, the same type of virtual resources has the same price for fairness. We consider that the price for a type of virtual resources is stable during a certain time frame, like the on-demand instances in Amazon EC2 [1]. The infrastructure provider is responsible for the availability and performance promises of virtual resources. There are many efforts in this area. This paper instead concerns the *tenant level resource management*.

A cloud tenant leases virtual resources to run its applications. For a specific business service, leasing a set of virtual machine (VM) instances from the cloud or buying a dedicated cluster may incur different cost [15]. Similarly, the same application running on different resource types or different amount of resources may result in different cost-effectiveness [11]. How to make leasing decisions and use leased resources efficiently is the main task of tenant level resource management. In this paper, we consider cloud tenants that make profit by providing services to their own clients. Such a service provider has its own service level agreements (SLAs) with its clients regarding how a client request is priced and how a client satisfies or dissatisfies the quality of the service. SLAs also enable the service provider to better understand the request pattern of a particular client, e.g., an SLA may include information such as mean request arrival interval and the range of processing time. As we will discuss later, the information is important for a service provider to efficiently manage resources. An SLA can also help identifying the party that is accountable when problems occur.

Generally, the objective of tenant level resource management for a service provider is to lease just sufficient amount of virtual resources to process its clients' requests so that the profit can be maximized. Apparently, the resource manager needs to meet the satisfaction targets set by clients, meanwhile, it shall maintain the fairness of effort and resource use among those clients who pay similar prices for processing similar requests. Currently there is a gap between service provider's resource use and client satisfaction.

In the *infrastructure level resource management*, it has been noticed that the billing of cloud resource use does not reflect the effort and resource spent by the provider [14]. The problem exists in the *tenant level resource management* as well, particularly for services that handle parallelizable requests. The processing of such a request may involve multiple VM instances with a MPI or a MapReduce program. The speedup of processing the request in parallel normally is not proportional to the number of VM instances used and is affected by many factors. For example, processing a request with 10 VM instances in parallel may reduce the execution time by 80%, but 5 VM instances may lead to a reduction of 70%. Whether the performance gain is worth the cost of running additional five VM instances is a question. Currently there is no systematic ways to involve the affected client into the decision making. Moreover, the workload fluctuation in the leased VM instances and request scheduling policies may further complicate the cost/benefit calculation. For a service provider running in the cloud, how to link the resource use to a client's satisfaction is essential to its profit, however, the problem has not been well investigated.

In this paper, we provide a tenant level resource management mechanism for parallel applications to tackle this problem. Our paper makes the following contributions:

1) We define SLA metrics that can be transformed into

978-0-7695-4756-5/12 $26.00 © 2012 IEEE
DOI 10.1109/SERVICES.2012.38

resource usage by the resource manager. These SLA metrics include the estimation of a client on its request pattern and the agreed performance targets between the client and the service provider;

2) We give a resource allocation scheme that differentiates client requests based on different settings in their SLAs and an algorithm called *JustSAT* that incorporates static capacity planning and dynamic resource allocation to just meet the SLA targets of clients.

The paper is organized as follows: Section II introduces the background and formulates the problem; Section III details the resource management algorithms; Section IV evaluates these algorithms; Section V summarizes the related work and Section VI concludes the paper.

II. Problem Description

In this section, we explain the problem by modelling a simple market scenario. The next section develops an algorithm to handle more complex scenarios.

A. Running a profitable service in the cloud

We consider a *service provider* has a set of *clients* requesting its service. The service provider leases a set of virtual machines (VMs) from a cloud *infrastructure provider* to serve client requests. The service provider pays for the time it runs VM instances and charges its clients for request processing on a per-request basis.

On the demand side, we denote the average hourly query request arrival rate from a client as λ. The client demand, in terms of client request arrival rate is affected by the price set for request processing in the market. We use the following constant elasticity demand curve [13] to model the relationship between mean request arrival rate and the price p for processing a request.

$$\lambda = Ap^{\epsilon} \tag{1}$$

in which, A is an arbitrary positive constant and ϵ is the elasticity. ϵ is normally a negative number. According to this demand curve, the decrease of price means the increase of request arrival rate.

We assume requests arrive following Poisson process and the average hourly request arrival rate from client u at price p is denoted by $\lambda(p)$. We also assume the request sizes follow exponential distribution with mean μ. The mean response time for requests from u is therefore $t = \frac{1}{\mu - \lambda(p)}$ according to M/M/1 queuing model.

By combining Equation 1, we have the following:

$$t = \frac{1}{\mu - Ap^{\epsilon}} \tag{2}$$

A client request can be parallelized and run on multiple VM instances. We model the speedup s using the following equation [6], [12]:

$$s = \ln(n) + 1 \tag{3}$$

in which, n is the number of VM instances used.

We denote the mean request processing rate with a single node as μ_0. At a given price p and a given number of VM instances n, the expected response time for requests from client u is as below based on the speedup model:

$$t = \frac{1}{\mu_0(\ln(n) + 1) - \lambda(p)} \tag{4}$$

1) The service level agreement (SLA) between a client and a service provider: Running a profitable service often involves service level agreements between the service provider and clients. In our scenario, the SLA between a service provider and client u has the following items:

- The client pays p to the service provider if the response time of a request is not greater than T.
- The client is compensated by αp if the request misses the target response time T.
- The client has a target rate q, which is the percentage of requests that are served within T.

The setting of T should be within the processing capacity of the service provider and also satisfy the client's needs. To make q percent of requests finish within time T, T needs to be equal to or greater than the q-percentile processing time of requests from the client, e.g., when q is 95%, T is given as below:

$$T = \frac{1}{\mu - \lambda} \ln(\frac{1}{1 - 0.95}) \approx \frac{3}{\mu - \lambda} \tag{5}$$

2) Client satisfaction and service profit: We use q to define a client's *satisfaction* to the service quality. As client requests can be processed in parallel, the service provider needs to make decision on how many VM instances should be allocated to parallelize the processing of a request in order to satisfy a client. This can be modelled as below by replacing μ in Equation 5 with a function of μ_0 and n in Equation 6:

$$T = \frac{1}{\mu_0(\ln(n) + 1) - \lambda} \ln(\frac{1}{1 - q}) \tag{6}$$

With a given T and request processing price p, increasing n will improve q. However, the increase of n may bring down profit as the service provider needs to pay for the time it runs these VM instances.

Consider that a set of VM instances are dedicated to requests from client u, the hourly revenue is $(q-(1-q)\alpha)p\lambda_u$ with q-percentile satisfaction rate and the profit $\mathcal{P}$ of serving client u during the hour of t is as below:

$$\mathcal{P} = (q - (1 - q)\alpha)Ap^{1+\epsilon} - nw^t \tag{7}$$

in which, w^t is the price paid for leasing an instance during time t (w^t is a constant for a given type of on-demand instances). n can be derived from Equation 6 as below:

$$n = e^{\frac{\ln(\frac{1}{1-q})}{T\mu_0} + \frac{Ap^{\epsilon}}{\mu_0} - 1} \tag{8}$$

Equation 8 also shows the cost of processing requests can be estimated if the target satisfaction rate q, target response time T and mean request size μ_0 are given.

By combining Equation 7 and 8, we derive the profit of the service provider by serving client u as below:

$$\mathcal{P} = (q - (1-q)\alpha)Ap^{1+\epsilon} - e^{\frac{\ln(\frac{1}{1-q})}{T\mu_0} + \frac{Ap^{\epsilon}}{\mu_0} - 1} w^t \quad (9)$$

With Equation 9, we now have a closer look of the relationship between profit $\mathcal{P}$ and q, T and μ_0. Fig. 1(a) shows the change of profit along with the target satisfaction rate and the target response time. We fix the price for request processing at 0.05 each. We also fix the price elasticity at -0.2. As the arrival rate is determined by the demand curve, the request arrival rate is stable. In the scenario shown in Fig. 1(a), all client requests have the same mean processing time on a single node. The profit increases along with q till a certain point and starts to decrease as q continues rising. The profit rises initially because the penalty of missing the target response time is high with a low q. However, the cost of ensuring the target satisfaction rate increases significantly when q increases to a certain point. On the other hand, serving requests with a rigid target response time generates lower profit when the request processing price is fixed. It is therefore difficult to ensure the fairness for clients with different target response time and satisfaction rate requirement even when the request sizes are similar as the service provider always has incentive to choose request that has relaxed target response time to process to reduce the resource leasing cost.

Fig 1(b) shows another scenario where different clients have different mean request size. It is clear that the service provider is likely to take small requests from clients to process. Moreover, requests with different speedup profiles may result in profit difference and therefore affect the service provider's decision making as well.

B. A semi-static resource allocation scheme

When multiplexing requests from different clients onto shared resources, a fixed pricing mechanism easily results in unfairness for clients if they have different target satisfaction rates, target response time or mean request sizes, as discussed above. On the other hand, a dynamic and fair pricing mechanism is often difficult to design and implement for practical use. Such a mechanism is likely to cause confusion on the client side. To address this problem, we use a semi-static mechanism to differentiate different types of requests in this paper. The mechanism prices client requests based on tuple (μ_0, T, q, sp), in which sp is the speedup profile of client requests. The mechanism works as follows:

1) A client provides an initial value of (μ_0, T, q, sp) in its SLA.
2) The service provider groups client requests together based on similar tuple values. In runtime, requests with different values are processed using different sets of resources.
3) The service provider monitors the changes of values provided by the client and adjusts these values to reflect the actual characteristics of client requests based on the monitoring data. Adjusting these values results in adjustment of resource allocation.

1) The Objectives for Resource Management: According to the description above, a client specifies the characteristics of its requests and performance requirement using the following tuple:

$$(A, \epsilon, \mu_0, T, q, sp)$$

The first two elements in the tuple determine the request arrival rate from u and can be obtained through market study or from historical data. The service provider therefore deals with a set of such tuples from various clients. It divides these tuples into groups based on the similarity of the last four elements in the tuple. The process is done offline at this stage of our work. The resource manager handles each group separately in runtime.

For each group, the resource manager has the following goals when handling requests from the group of clients:

1) Optimizing the resource use to meet the targets on response time and satisfaction rate of different clients and avoid resource over- or under-provisioning;
2) Ensuring the fairness of the service among clients.

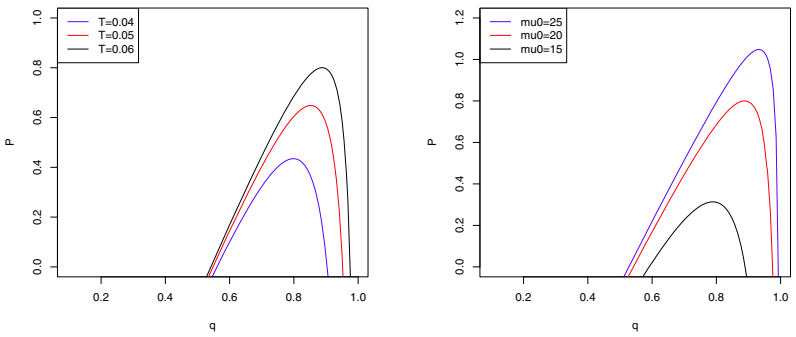
The fairness includes two aspects: firstly, a group of clients that are charged the same price for their request processing should have similar SLA satisfaction rates; secondly, the group of clients should have similar speedups for their request processing if their requests share a speedup profile, in another word, the resource consumption of each client should be similar. Formally, we define the fairness as the standard deviation of q for the first aspect and standard deviation of speedups for the second aspect among a group of clients.

We give a resource management algorithm called *JustSAT* to achieve this goal in the following section.

III. Resource Allocation for SLA Satisfaction

Based on the resource allocation scheme described in Section II, the resource manager consists of two components. The first component deals with capacity planning, which estimates the number of VM instances required to serve incoming requests to achieve given targets set in SLAs. The second component handles runtime monitoring and dynamic resource pool adjustment. The resource manager tracks the satisfaction of each individual user and adjusts the resource allocation and request scheduling strategies accordingly.

The resource allocation model is illustrated in Fig. 2. The resource manager maintains a queue for requests from various clients. In runtime, it dynamically makes resource



(a) (b)

Figure 1. The change of client hourly profit along with satisfaction rate: $\alpha = 1.0$, $\epsilon = -0.2$, $w^t = 0.05$, $p = 0.06$, $A = 15$. (a) $\mu_0 = 20$; (b)$T = 0.06$.

leasing and request scheduling decisions. VM instances are grouped to parallelize request processing. Upon arrival, a request is dispatched to an instance group (or cluster) that is most likely to satisfy the client according to its SLA.

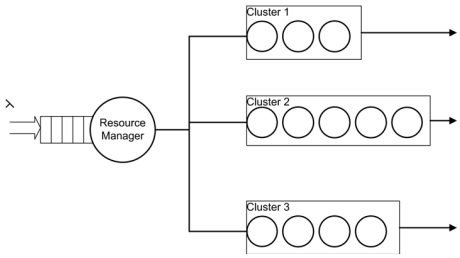


Figure 2. The resource allocation model.

A. Example

As the increase of resource use is not proportional to the speedup for most parallel applications, partitioning resources into groups can often improve the performance. Here we use a simple example to explain the difference.

We consider the resource pool leased by the service provider consists of N VM instances. We examine two methods of using these instances to serve client requests. The first method processes each request in parallel with all N nodes. The second method partitions the N instances into two groups with each consists of $N/2$ instances. We assume that the average service rate (number of requests processed in a time unit) by using all N nodes to process each request in parallel is 1.8μ while the average service rate of using $N/2$ is μ.

With the assumption that these requests arrive in a Poisson process at rate λ and request processing time follow exponential distribution, the mean response time is $\frac{1}{1.8\mu-\lambda}$ with the first method.

In the second method, requests are dispatched to the two groups of instances to run. This forms a $M/M/2$ queue. The mean response time of requests is as below:

$$\frac{1}{\mu} + \frac{P_Q}{2\mu - \lambda} \tag{10}$$

in which, P_Q is the probability that a request finds that the two clusters of nodes are all busy.

$$P_Q = \frac{1}{2}(\frac{\lambda}{\mu})^2 \cdot \frac{p_0}{1 - \frac{\lambda}{2\mu}} \tag{11}$$

p_0 is as below:

$$p_0 = (1 + \frac{\lambda}{\mu} + \frac{1}{2}(\frac{\lambda}{\mu})^2 \frac{1}{1 - \frac{\lambda}{2\mu}})^{-1} \tag{12}$$

When $\mu = 13$ and $\lambda = 20$, the mean response time of method 1 is 0.29 while that of method 2 is 0.19. The performance achieved by the service provider is very different even though the two methods incur the same resource leasing cost. In the following, we give a method to estimate the resource needs to achieve given SLA targets and efficiently partition leased resources to serve parallelizable requests.

B. Capacity planning

Our resource estimation method is based on the following observation: *a service provider always intends to keep high utilization of the VM instances it leases from the infrastructure provider.*

When the utilization of VM instances is high, the load of the system, i.e., $\frac{\lambda}{m\mu}$ (m is the number of groups and μ is is service rate for one group) is likely to be high. Under a high load, the average response time of serving requests with a $M/M/m$ queue is close to that with a $M/M/1$ queue. Hence we can make initial estimation and partitioning of leased resources using $M/M/1$ queuing model. Given a response time target T and client satisfaction rate q, the number of VM instances can be calculated by Equation 8.

We consider that sp stores the map between instance number and speedup of parallel request processing. sp_i is

the average speedup of running requests using i instances. We have the following estimation of q-percentile response time with i instances:

$$T = \ln(\frac{1}{1-q})\frac{1}{m\mu_0 sp_i - \lambda} \quad (13)$$

in which, m is the number of instance groups. The total instance number required to achieve target response time T at a satisfaction rate q is therefore $m \cdot i$. The condition is as below:

$$\frac{\lambda}{m(\mu_0 sp_i)} < 1 \quad (14)$$

Algorithm 1 details the process of selecting the minimal number of VM instances to achieve given performance target.

Note, even though the actual request processing time does not necessarily follow exponential distribution, our method can still maintain certain accuracy. This is because that our semi-static resource allocation scheme contains an implicit variation reduction phase. The resource manager can use the information in SLAs to group clients based on the similarity of their mean processing time, target client satisfaction rate and target response time, as a result, the variation of request processing in each client group is not significant. Our monitoring mechanism can further enforce low variation by moving a client from one client group to another if her request pattern changes.

Algorithm 1: JustSAT – Estimation and partitioning

Input:
sp: a vector of speedups;
λ: the request arrival rate;
μ_0: mean service rate with single instance;
T: target response time in SLA;
q: target SLA satisfaction rate.
Output:
(m, n): m is the number of groups and n is the minimal number of VM instances in a group.

$(m, n)^{MIN}$ = (MAX, MAX) ;
obtain q-percentile request processing time with single instance, denoted by $\frac{1}{\mu_q}$ based on historical data and speedup profile ;
foreach sp_i **do**
  $t = \frac{1}{\mu_q sp_i}$;
  if $t \leq T$ **then**
    $m' = \frac{\ln(\frac{1}{1-q})\frac{1}{T} + \lambda}{\mu_0 * sp_i}$;
    if $m' \cdot i < m \cdot n$ **then**
      $(m, n)^{MIN} = (m', i)$;
    end
  end
end
return $(m, n)^{MIN}$;

C. Monitoring and dynamically adjusting resource allocation

As mentioned above, the request processing time in real world may not follow exponential distribution. We have a dynamic resource allocation component that works with Algorithm 1 to adjust resource allocation according to monitoring data. A monitor tracks the following metrics:

- q-percentile response time in the past time interval. q is the target satisfaction rate in the SLA between the service provider and a group of clients. The time is normalized to the response time on single instance by using the speedup profile;
- mean request processing time in the past time interval;
- average SLA satisfaction rate of all clients;
- individual SLA satisfaction rate of each client;
- mean request arrival rate.

Algorithm 2 is activated in fixed time intervals. It first calculates all possible ways of partitioning instance pool that can achieve the client targets of satisfaction rate and response time based on the monitoring data of the most recent time interval. It then sorts these combinations (stored in I) based on the number of instances needed to achieve these goals. When the average satisfaction rate is above the target, the algorithm returns the combination with the least number of instances. When the average satisfaction rate is below the target, the algorithm chooses a combination with instance number just above the combination used in the previous time interval. The dynamic resource allocation algorithm adopts the following heuristics: increasing the instance number can eventually improve the system performance. Note, the improvement does not necessarily happen in the immediate next time interval.

Algorithm 2 intends to satisfy the overall targets of the whole client group. Our resource manager has another dynamical adjustment mechanism to optimize the fairness for each individual client in runtime. The mechanism tracks the SLA satisfaction rate for each individual client. When a client with the lowest satisfaction rate is behind her target with a margin, the resource manager will use a *preemptive request scheduling policy* that moves the requests of the client to the head of the queue whenever there are requests from other clients ahead of them. The policy continues till either no client misses her target or all clients are in a similar distance to their targets. As the resource partitioning applies to requests from the same client group, the fairness on speedup of each client is ensured by our *JustSAT* Algorithm.

IV. Evaluation

A. Experiment Settings

To evaluate the performance of our algorithm, we implement a resource manager and feed it with synthetic workload. The sizes (processing time on single instance) of

Algorithm 2: JustSAT – Dynamic resource allocation

Input:
sp: a vector of speedups;
λ': the updated request arrival rate;
μ'_q: q-percentile response time;
μ'_0: updated mean service rate with single instance;
sat': the updated average satisfaction rate;
(m, n): existing instance allocation;
T: target response time in SLA ;
q: target SLA satisfaction rate.
Output:
(m', n'): new instance allocation.

$I = \phi$;
foreach sp_i **do**
  $t = \frac{1}{\mu'_q sp_i}$;
  if $t \leq T$ **then**
    $m' = \frac{\ln(\frac{1}{1-q})\frac{1}{T}+\lambda'}{\mu'_0 \cdot sp_i}$;
    $I = I \cup (m', i)$;
  end
end
sort I according to $(m \cdot n)$ in ascending order ;
if $sat' < q$ **then**
  foreach $i \in I$ **do**
    if $m \cdot n \leq i_m \cdot i_n$ **then**
      $(m', n') = i$;
      return (m', n');
    end
  end
end
return (I_{0_m}, I_{0_n});

requests submitted to the resource manager follow Bounded Pareto distribution with $\alpha = 1.1$. These requests may be submitted by different clients while each client provides a profile of its requests. The profile includes the estimated mean request sizes (processing times), as well as the lower and upper bound of these sizes. In our experiment, we use the lower and upper bound in the Bounded Pareto random number generator. A client also specifies the target response time and target satisfaction rate in the SLA.

The resource manager leases homogeneous VM instances for request processing. We assume the hourly price of a VM instance is a constant in the experiment (as an example, the on-demand instance type in Amazon EC2). The price for processing each request that uses the same SLA is also a constant. The revenue of the service provider can therefore be expressed by the SLA satisfaction rate. We also assume that the time used to start a new VM instance is trivial compared with the request processing time.

In our experiment, we use two speedup profiles as shown in Table I to characterize the parallelism of client requests.

Table I
SPEEDUP PROFILES

instance#	1	2	3	4	5	6	7
profile 1	1.0	1.69	2.1	2.39	2.61	2.79	2.95
profile 2	1.0	1.80	2.99	3.97	4.29	5.41	6.53

Profile 1 characterizes the speedup with n instances as $(\ln(n) + 1)$. Profile 2 is collected through running *mpiBlast* [5] in Amazon EC2 with c1.medium instances [1]. The genome sequence database used is *swissprot*.

In the following, we compare *JustSAT* with a randomized parallelization algorithm (*RandomParallel*).

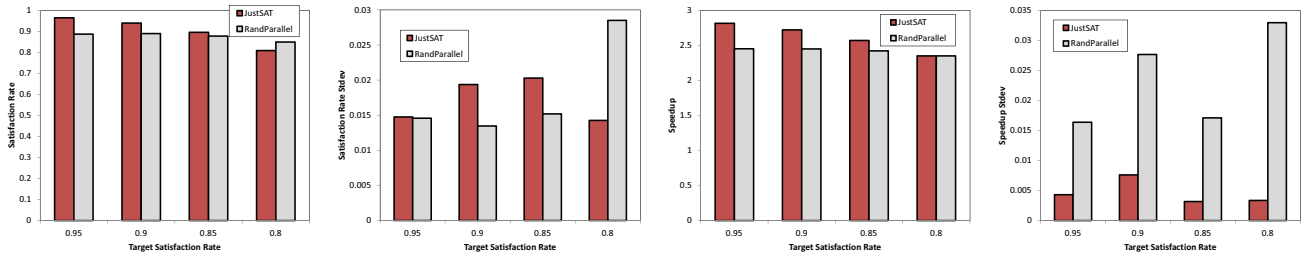
B. Comparison with randomized algorithm

Upon the arrival of a request, *RandomParallel* selects a random number of VM instances to parallelize the processing of the request. The random number is between 3 and 7 in the results shown in Fig.3. We set the lower bound of the random number to a value that is sufficient to meet the target response time according to a given speedup profile. The request sizes vary from 500 to 1500 in this experiment. The total number of clients is set to 10, each with different request arrivals generated from a uniform random distribution with a mean inter-arrival time between 50 and 1,100. The target response time in the SLA of the group of clients is 500. We run *RandomParallel* on the same number of VM instances used by *JustSAT* and compare its performance with that achieved by *JustSAT*.

Fig. 3(a) shows that *RandomParallel* fails to reach the target satisfaction rate at 95% and 90% level for client requests, while the actual satisfaction rate achieved by *JustSAT* is 96.5% and 94% for target rate 95% and 90% respectively. Both algorithms meet the target satisfaction rate at 85% and 80% level. This can be explained by Fig. 3(c). The speedup of *JustSAT* decreases as the target rate drops, but the speedup of *RandomParallel* does not change much at different target satisfaction rates. It is easy to see that *JustSAT* can fully use the capacity of the leased resources to satisfy rigid targets while *RandomParallel* can only use the same number of resources to achieve less rigid targets. As further shown in Fig. 3(c), the speedup of *JustSAT* is getting closer to that of *RandomParallel* as the target satisfaction rate decreases.

On the other hand, the standard deviation of the actual satisfaction rates is small for both *JustSAT* and *RandomParallel* due to that both algorithms do not assign priority to clients. However, the standard deviation of speedups of all clients shows difference. *JustSAT* is fairer than *RandomParallel* as *JustSAT* uses the same number of instances to parallelize request processing within a monitoring interval.

Fig. 4 compares *JustSAT* with *RandomParallel* under a relaxed target response time 650, in which we expect the randomized parallelization algorithm performs better as there is more flexibility in selecting the number of instances to process a request without missing the deadline. The



(a) (b) (c) (d)

Figure 3. Comparison with randomized parallelization (Request profile – profile 1; target response time: 500): (a) SLA satisfaction rate; (b) Standard deviation of SLA satisfaction rates; (c) Average speedup; (d) Standard deviation of speedups of different clients.

actual SLA satisfaction rates of *RandomParallel* indeed improves for profile 1. As shown in Fig. 4(a), the actual SLA satisfaction rate at 90%, 85% and 80% are all above the corresponding target rate with *RandomParallel*. The actual satisfaction rate at 95% target rate is slightly lower than the target at 93.7%. In comparison, the actual satisfaction rates for *RandomParallel* miss their targets at 95% and 90% level in Fig. 3(a) while the actual satisfaction rate is only 88.7% at 95% target rate.

Note that Fig. 4(b) and Fig. 4(c) reveal the different strategies used by *JustSAT* and *RandomParallel* in achieving the target satisfaction rate on profile 1. *JustSAT* has a higher average queue length (the execution time used for computing slowdown is the parallel processing time) as well as a higher average speedup than *RandomParallel*, which indicates *JustSAT* uses more instances on average to process each request in parallel. We explained early that the strategy increases the processing capacity of the resource pool. This is indicated by the higher SLA satisfaction rate *JustSAT* achieves at the 95% target level.

For profile 2, *RandomParallel* that randomly chooses instances with number between 2 and 7 performs poorly at 85% and 80% target satisfaction rate. It is mainly due to two reasons: firstly, the parallel processing time of requests that have a speedup model according to profile 2 is sensitive to the instance number used and a small change in instance number may result in big difference in capacity of resources; secondly, the lack of adaptivity to the workload is likely to create a backlog that dramatically increases the time a request stays in the system. The slowdown at 85% and 80% target satisfaction rate is 6.21 and 6.85 respectively, as shown in Fig. 4(b). In comparison, the slowdown at 85% and 80% target satisfaction rate is only 1.75 and 2.30 for *JustSAT*.

Another algorithm that can be used for comparison uses a fixed number of instances to process each client request in parallel. However, how to set the fixed number is a non-trivial task for this type of algorithms. In general, their performance is comparable to *JustSAT* only when the fixed number matches the workload.

V. Related Work

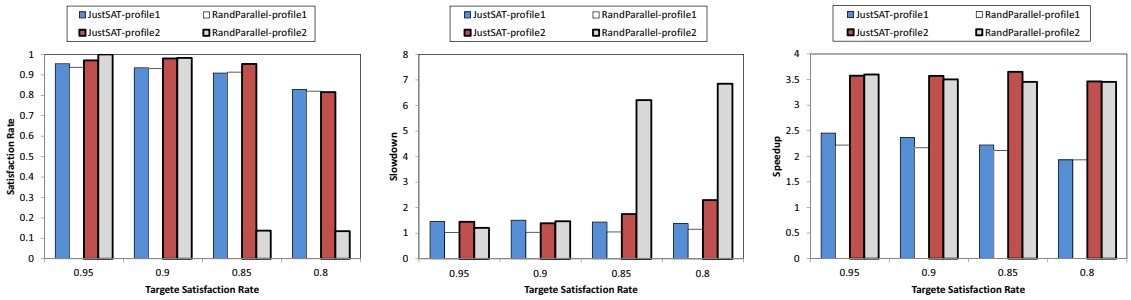
Our work is mainly related to market-based and SLA-based resource management. Some market-based resource allocation methods are non-pricing-based [7], [17], [9], but pricing-based methods can reveal the true needs of users who compete for shared resources [10]. Pricing plays an increasingly important role in cloud computing. Our work assume that client demand is affected by request processing price.

The pricing mechanisms can be further divided into non-time-varying and time-varying. In the former, the price is the value associated with each request and it does not change with the response time [3]. In the latter, the price reflects the value change of a job along with the increase of response time [4]. The price change can be triggered by deadline of response time [16], a decay rate associated with the response time[4], or a combination of deadline and decay rate [8]. Our work adopt a deadline based request pricing mechanism.

Recently, K. Tsakalozos et.al. [12] investigate pricing and profit maximization in resource management for master-worker based parallel applications. It intends to maximize profit by computing optimal number of VMs for a parallel job. The optimal number is at the point where the marginal cost equals to the marginal revenue. The main technique targets on profit maximization for a single job. Our work deals with client satisfaction with a number of requests, which affects a service provider's long term profit.

P. Xiong et.al. [16] give an SLA based resource management method for databases. The main performance metrics in an SLA is based on the deadline of single query response time. Clients are classified into different classes based on different deadlines. Parallelism is associated with the number of data replicas. Its resource allocation is driven by meeting profit target and minimizing penalty. Differing to [16], our method considers the parallelism in the processing of a request.

Other SLA based resource provisioning work include [18] and [2]. [18] uses a step-wise time decay function in SLAs and client requests are divided into different classes with each class has its own SLA. [2] proposes a data structure



(a) (b) (c)

Figure 4. Comparison with randomized parallelization (the instance number uses the one produced by JustSAT, target response time 650): (a) SLA satisfaction rate; (b) Average slowdown; (c) Average speedup.

called SLA-tree for a service provider to decide whether to postpone or expedite a query in a queue. These work do not apply to parallel workload.

VI. Conclusion

Tenant level resource management becomes increasingly important as more and more profitable services are running in the cloud. For those services that are capable of parallel processing their client requests, there is a need for achieving cost-effectiveness. In this paper, we gave a resource management method that is capable of provisioning "just sufficient" resources to satisfy client requests for these services. Our method defined a realistic SLA that helps to group clients with similar characteristics. The method reduced the request processing time variation in each group. We further developed a dynamic resource allocation method to track the actual client satisfaction rates and adjust resource allocation accordingly. Our experiments showed the effectiveness of our method.

References

[1] Amazon Inc. Amazon EC2. http://aws.amazon.com/ec2/.

[2] Y. Chi, H. J. Moon, H. Hacigümüs, and J. Tatemura. SLA-tree: a framework for efficiently supporting sla-based decisions in cloud computing. In *EDBT*, pages 129–140, 2011.

[3] B. N. Chun and D. Culler. Market-based proportional resource sharing for clusters. *Technical Report CSD-1092, University of California at Berkeley*, January 2000.

[4] B. N. Chun and D. E. Culler. User-centric performance analysis of market-based cluster batch schedulers. In *CCGRID*, pages 30–38, 2002.

[5] A. E. Darling, L. Carey, and W. chun Feng. The design, implementation, and evaluation of mpiBLAST. In *ClusterWorld Conference & Expo and the 4th International Conference on Linux Clusters: The HPC Revolution*, 2003.

[6] D. Eager, J. Zahorjan, and E. Lazowska. Speedup versus efficiency in parallel systems. *Computers, IEEE Transactions on*, 38(3):408 –423, mar 1989.

[7] D. F. Ferguson, C. Nikolaou, J. Sairamesh, and Y. Yemini. Economic models for allocating resources in computer systems. In S. H. Clearwater, editor, *Market-Based Control: A Paradigm for Distributed Resource Allocation*. World Scientific, 1996.

[8] D. E. Irwin, L. E. Grit, and J. S. Chase. Balancing risk and reward in a market-based task service. In *HPDC*, pages 160–169, 2004.

[9] J. F. Kurose and R. Simha. A microeconomic approach to optimal resource allocation in distributed computer systems. *IEEE Trans. on Computers*, 38(5), 1989.

[10] K. Lai. Markets are dead, long live markets. *Sigecom Exchanges*, 5(4):1 – 10, July 2005.

[11] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: comparing public cloud providers. In *Internet Measurement Conference*, pages 1–14, 2010.

[12] K. Tsakalozos, H. Kllapi, E. Sitaridi, M. Roussopoulos, D. Paparas, and A. Delis. Flexible use of cloud resources through profit maximization and price discrimination. In *ICDE*, pages 75–86, 2011.

[13] H. R. Varian. *Intermediate Microeconomics: A Modern Approach*. W. W. Norton & Company, 2009.

[14] M. Wachs, L. Xu, A. Kanevsky, and G. R. Ganger. Exertion-based billing for cloud storage access. In *3rd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2011.

[15] E. Walker. The real cost of a CPU hour. *Computer*, 42(4):35 –41, april 2009.

[16] P. Xiong, Y. Chi, S. Zhu, H. J. Moon, C. Pu, and H. Hacigümüs. Intelligent management of virtualized resources for database systems in cloud environment. In *ICDE*, pages 87–98, 2011.

[17] Y. Yemini. Selfish optimization in computer networks. In *Proc. the 20th IEEE Conf. Decision and Control*, pages 281 – 285, 1981.

[18] L. Zhang and D. Ardagna. Sla based profit optimization in autonomic computing systems. In *ICSOC*, pages 173–182, 2004.