Workload Consolidation for Cloud Data Centers with Guaranteed QoS Using Request Reneging

Soamar Homsi, *Member, IEEE,* Shuo Liu, *Member, IEEE,* Gustavo A. Chaparro-Baquero, *Member, IEEE,* Ou Bai, Shaolei Ren, and Gang Quan *Senior Member, IEEE*

Abstract—Cloud data centers are widely employed to offer reliable cloud services. However, low resource utilization and high power consumption have been great challenges for cloud providers. Moreover, the rapid increase in demand for affordable cloud services magnifies the obstacles for proficient resource management policies. In this paper, we investigate how to improve resource utilization and power consumption in cloud data centers when delivering services with statistically guaranteed Quality of Service (QoS). We assume that the service provider hosts different types of services, each of which has request classes with different QoS requirements. Different from the traditional approaches that distribute workloads with different QoS levels on different Virtual Machines (VMs), we introduce an approach to pack requests of the same service type, even with different QoS requirements, into the same VM, and to remove potential failure requests in time to improve resource usage and energy cost. We formally prove that our algorithm can statistically guarantee QoS conditions in terms of deadline miss ratios. We develop a cloud prototype to empirically validate our proposed methods and algorithm. Our experimental results demonstrate that our approach can significantly outperform other traditional approaches in terms of QoS guarantees, power consumption, resource demand and electricity cost.

Index Terms—Cloud computing, virtualization, workload consolidation, power efficiency, utilization, reneging, guaranteed QoS.

1 Introduction

Cloud computing [1] has recently become the dominant trend for the *continuous delivery* (CD) of online services over the internet using large-scale data centers. In the meantime, the relentless increase in demand for different services [2], [3], in both personal and professional life sectors with high Service Levels (SL), has posed crucial challenges on cloud service providers. Maintaining excessive computing resources won't effectively address this problem, as it can lead to tremendously high power consumption rates and energy costs.

Exorbitant power consumption rates and energy costs are among the main concerns in cloud infrastructure facilities. Cloud service providers strive to enrich competing markets with more reliable, yet less costly, services to a modern world handles Everything as a Service (EaaS) [1]. Whereas the price for online services decreases and the performance of computing systems increases at almost the same rate as Moore predicted five decades ago, the performance-per-watt of computing components increases at a much slower rate than what Moore has predicted [4]. As an example, in 2013, the annual electricity consumption of data centers only in the United States was close to 91 billion KiloWatt Hour (KWH), which is larger than the annual amount of electrical power required by most countries [5]. Thereupon, service providers are taxed by intimidating energy bills as they try to provide satisfactory Quality-of-Service (QoS) guarantees. Such a consumption rate of electricity is not only a cost-and-profit problem, but also a serious threat to the environment as a result of the large amounts of carbon dioxide emissions during powering and cooling those data centers [5]. As a result, proficient power-aware resource management policies become a necessity and a key infrastructure component for any agile, consolidated and dynamically scalable cloud's data center that provides affordable and reliable high-quality services.

On the other hand, power-saving techniques tend to, if not always, cause a degraded computing performance. The QoS is the key to clients' satisfaction, and service providers normally provide multiple Service Level Agreements (SLAs) regarding different QoS kinds. For example, a database may be queried internally by a company's employees or externally by a company's customers, who may have a higher or a lower priority than that of the employees [6]. Cloud service providers need to provide a competing and guaranteed QoS, but with less energy costs. Whereas over-provisioning is a common and simple solution to avoid SLA violations, resource over-provisioning is an expensive method by which resources are drastically underutilized, particularly under the unpredictably fluctuating cloud workloads [7].

Low resource utilization [8] is a prevailing problem in virtualized data centers, and is a major leading factor to their high power consumption and increased operational costs [4]. For example, while Google service provider makes its data centers greener by benefiting from wind and solar energy sources, and operating recycling cooling systems, the utilization of Google's servers is less than 50% on an average [3]. Maximizing resource utilization becomes more crucial when performance must meet a defined satisfactory level of service given by QoS conditions. The challenge is then how to allocate the cloud's workloads in a way that maximizes resource usage and guarantees the QoS requirements.

In this paper, we propose and investigate new power-aware cloud workload allocation policies to minimize the processing power demand of cloud's services in cloud's data centers, and to reduce energy consumption, with statistically guaranteed QoS for users. We assume that clustered

data centers are able to accommodate different types of services, each of which can have request classes with different QoS requirements. Our main contributions in this paper are: (1) Different from previous studies that employ separate Virtual Machines (VMs) for requests with different QoS requirements, we develop a workload multiplexing method that enables requests of the same service type, but with different QoS constraints, to share the same VM. To our best knowledge, this is the first approach by which different requests with different QoS guarantees can be hosted on a single node in order to increase resource utilization. (2) We also devise a novel methodology that properly discards potential failure requests as soon as possible to minimize processing rate demands, and to reduce total power consumption with statistically guaranteed QoS. We introduce a packing and consolidation algorithm, called *Green Workload Packing and Consolidation algorithm* (GWPC), that statistically guarantees the QoS requirements of service requests in terms of deadline miss ratios. (3) We design a fully-automated private Green Cloud Computing Prototype (GCCP) that conforms to the industrial standards, generates, and runs different benchmarks under well-controlled conditions. We further extend it to incorporate new workload scheduling, resource provisioning and performance monitoring schemes proposed in this paper. (4) In addition to the analytical validation of our proposed methods, we experimentally verify them, under general and cloud-specific workloads, by implementing our algorithm, along with three other common algorithms in GCCP. Extensive experimentation results show that GWPC outperforms existing approaches widely in terms of the QoS satisfaction, power consumption efficacy, resource demand minimization, and electricity cost saving. We use the words VM, node and server interchangeably throughout the paper.

The rest of this paper is organized as follows: Section 2 discusses related work; Section 3 introduces the system model; Section 4 details the preliminary study; Section 5 describes the GWPC Algorithm; Section 6 describes the validation planform GCCP and its functional modules. Section 7 validates our analytical findings through intensive experimentation and different benchmark types, and shows the results; Section 8 discusses the conclusions and potential future extensions of this work.

2 Related Works

Numerous efforts have been made to reduce power and energy consumption in service-oriented computing systems. We can categorize those researches into different abstraction levels and/or according to different criteria. For example, according to the scale/type of the computing systems, Cai et al. in [8] categorized the energy-aware techniques applicable for servers [9], [10], clusters [11], [12], [13], data centers [4], [14] and clouds [15]. Power/energy aware approaches can also be classified according to the different resource types, such as CPUs [4], [16], memory [7], storage devices [17], and/or network [18]. However, since CPUs usually acquire the highest power consumption among all resource types [14], we focus on improving power/energy efficiency of CPUs in cloud data centers using techniques such as, virtualization, workload consolidation and scheduling [19].

Dynamic Voltage Scaling and Dynamic Frequency Scaling (DVS/DFS) have been powerful conventional methods for adaptive performance and power dissipation adjustment to achieve power efficiency [11], [12]. Hwang et al. showed that the maximum energy savings in virtualized multi-core servers can be achieved when combining the DVS/DFS methods and the consolidation algorithms [9]. Beloglazov et al. introduced in [15] a global-and-local layer approach to make virtualized servers more power-efficient by adjusting the frequency and voltage of processors according to VMs' utilization. Likewise, Kim et al. [11] proposed DVFS-enabled, with both time-shared and space-shared, cluster scheduling policies for a bag of tasks to reduce power consumption and to meet end-users' deadline requirements. Although many researchers and engineers acknowledge that DVFS scheduling algorithms are powerful energy-saving solutions on the server's level, there are many challenges when they are applied in the current virtualized data centers; for instance, they are architecture dependent, hence they may not achieve their best power/energy-saving when applied to the current heterogeneous cloud data centers.

As virtualization technology evolved as a norm in today's data centers to amplify resource usage through running multiple VMs on a single server, VM migration has been widely employed to optimize server utilization and to reduce power consumption [20]. In [21], Mastroianni et al. statistically modeled and analyzed the effects of VMs allocation and migration on minimizing the number of powered-on servers, and on reducing power consumption in data centers. Zhen et al. [20] introduced the concept of "skewness" to measure the unevenness in the servers' multidimensional-resource utilization. However, VM live migration requires a delay that can degrade the overall system performance and availability, and consequently leads to SLA violations [22].

In conjunction with VM migration, server consolidation is of special interest among efficient resource allocation policies [23]. Server consolidation, comparatively to DVFS, improves resource utilization without demanding excessive hardware resources, and it is easy to implement and to deploy [21]. Now that, server power consumption is not exactly proportional to its utilization, and a server may consume a significant amount of power even when it is idle [21], server consolidation methods pack running VMs on a smaller number of physical servers and/or turn off the rest, to minimize the total power consumed by those servers [10], [21], [24]. In [10], Verma et al. presented a two-dimensional, i.e., memory-based and CPU-based, consolidation strategy in which decisions are based on the correlation among different workloads. In [10], Pinheiro presented an algorithm to dynamically turn servers on and off according to the imposed load in computing clusters. Chase et al. [25] reduced the energy consumption of server clusters by degrading services according to their SLAs, when power consumption or thermal dissipation exceeds certain limits.

Whereas saving power/energy is important, service providers must also ensure that their services can satisfy users' QoS requirements, such as response time and/or deadline miss ratios. For instance, a recent report has found that a 100ms extra delay costs Amazon 1% of sales revenue [26]. The problem becomes more challenging with interactive workload types [27], such as online gaming [3] and

multimedia streaming services [28], whose response times are crucial. These online interactive services are commonly described by soft timely constraints [27], in that service providers must guarantee that a predefined percentage of them meet their deadlines. Otherwise, service providers are regarded as failing to keep up with their SLA agreement [2].

VM placement methods with performance-interference awareness were introduced in [29] to improve the performance of VMs and the utilization of physical machines. Resource overbooking, i.e. allocating more resources than the actual available capacity in order to raise service provider profit, with different real-time constraints is presented in [27]. Energy-aware resource allocations with response time and end-to-end time guarantees were introduced in [24] and [13], respectively. Greenberg et al. [18] studied the costs of cloud data centers, and recommended to developing new management systems within and across geographically distributed data centers with the focus on network agility to improve their efficiency and end-to-end performance.

SLA-aware workload consolidation had been proposed to achieve higher dynamic power efficiency, and to overcome the under-utilization problems resulting from applying the over-provisioning policies [30]. Lee et al. [31] developed a pricing model, based on the queue model M/M/1/PS, and used it to develop a profit-driven scheduling algorithms with SLA for clouds' dependent services.

To guarantee service requests with different classes of QoS requirements, it has been a common approach (e.g., [13]) to serve requests with the same QoS requirements on the same VM. Now that, all requests on the same machine have the same QoS requirement, different types of QoS can be captured by a single variable, such as provisioned resources (e.g., [10] [21] [15]), required processing speed (e.g., [12]) or latency (e.g., [24] [13]). Although this approach simplifies the resource management problem to guarantee one specified QoS criteria, it excludes requests that can share resources, and the overall resource usage can be rather inefficient, as illustrated later in this paper. Additionally, almost all previous works implicitly assumed that all accepted requests must be served, even if they do not meet their QoS conditions. We show in this paper that if we can judiciously discard the requests that are likely to miss their deadlines, we can significantly improve resource usages without compromising QoS conditions.

3 System Model

In this section, we introduce our system model and formulate the problem.

3.1 Service Model

We assume that a cloud data center consists of several cloud computer clusters, each of which consists of two or more physical machines, and has its own service manager that is analogous to the cluster schedulers in Google's clustered data centers [6]. Each cloud's cluster has a cloud orchestrator (such as, OpenStack) and a virtualization hypervisor (such as, Xen [32]) that work together to create VMs with different types and capacities for hosted services, and to make them available online for customers who submitted requests with different QoS requirements. We assume that a service provider provides n different types of services based on their application purpose $S = \{S_1, ..., S_n\}$. Each type of service (e.g., S_i) can accommodate different classes of service requests $\Gamma_i = \{\tau_{i,j}, j = 1, \ldots, r_i\}$, i.e. requests under different SLAs. Each class of requests (e.g., $\tau_{i,j}$) has its own QoS requirements (e.g., $Q_{i,j}$). We assume that different types of services must be hosted on different VMs, but different classes of requests of the same type can be potentially hosted in the same VM. We assume that there are n types of VMs $\{VM_1, ..., VM_n\}$ with capacities of $\{C_1, ..., C_n\}$ supporting n different types of services $\{S_1, ..., S_n\}$, respectively. VMs with the same service type S_i are logically grouped together into a single server pool SP_i. A server pool SP_i may contain up to m_i VMs. Each VM $\{VM_{i,k}, k = 1, \ldots, m_i\}$ within a server pool SP_i can require a different processing rate $\{U_{i,k}, k = 1, \ldots, m_i\}$. Our service model is illustrated in Fig. 1.

As long as both waiting and average response times distributions of industrial workloads' requests have variances with small coefficients, we assume that request arrival patterns follow the Poisson distribution, and their response times follow the exponential distribution [33], as they approximate the actual corresponding distributions with acceptable precision [34]. Different request classes of the same service type may have different arrival times, deadlines, and completion ratios. Specifically, a request is modeled with a 3-tuple, i.e. $\tau_{ij} = \{\lambda_{i,j}, D_{i,j}, R_{i,j}\}$; where $\lambda_{i,j}$ is the arrival rate of j-class requests in service S_i, $D_{i,j}$ is the deadline of j-class requests in S_i, and $R_{i,j}$ is the required completion ratio of j-class requests in S_i. The QoS requirement $Q_{i,j}$ of a request $\tau_{i,j}$ is defined by $\{D_{i,j}, R_{i,j}\}$, meaning that at least $R_{i,j}$ percent of $\tau_{i,j}$ requests have to be served no later than $D_{i,j}$ as in [35]; for example, CloudSuite [36], a benchmark suite for cloud services, describes the QoS constraints of web search requests by a latency of $D = 500$ ms and completion ratio of $R = 90\%$ [37].

3.2 Power Model

Considering that the allocation of processing units in cloud data centers generally occurs at levels of whole core(s) [35], [38], we assume that each VM is allocated to an individual processing core on a physical server, and thus we adopt a power model similar to that in [21] to model the power consumption of a VM, as shown in (1):

$$P = P^d \varphi + P^s \tag{1}$$

where; P^s is the static power, P^d is the dynamic power, and φ is the utilization of a processing core. φ is defined as $\varphi = \frac{U}{C}$; where U is the processing rate for a VM, and C is the capacity limit of the core allocated to it (i.e. the maximum processing rate available). We calculate P^d and P^s empirically, as explained later on in section 6, and we assume that they are constant and the same for each set of m_i cores hosting the same service type S_i. This model can be easily extended to the scenarios wherein a VM is mapped to multiple cores.

3.3 Problem Definition

With the system model defined above, the problem we are to address can be formulated as follows.

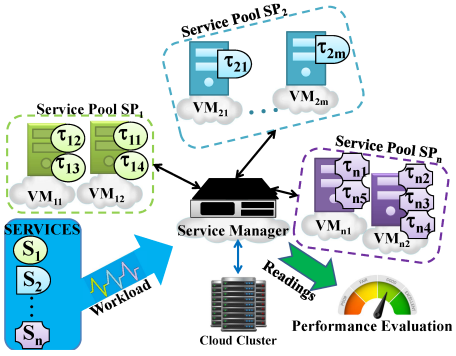


Figure 1. Service Model. A cloud cluster with n server pools, each pool SP_icontains $m_i \geq 1$ VMs, e.g., SP_1 contains 2 VM_1-type VMs $\{VM_{11}$,$VM_{12}\}$, and SP_2 contains m VM_2-type VMs $\{VM_{21}$, ..., $VM_{2m}\}$. There are n types of services $\{S_1, S_2, ..., S_n\}$. Different request classes of the same service type can share the same VM (e.g., τ_{11} and τ_{14} share VM_{12}, and τ_{12} and τ_{13} share VM_{11}). Contrarily, $\{\tau_{21}, \tau_{2m}\}$ are hosted separately on $\{VM_{21}, ..., VM_{2m}\}$, resp.

Problem 1. *Given service requests* $\Gamma = \{\tau_{i,j} : j = 1, ..., r_i; i = 1, \ldots, n\}$, *determine the server pools* $SP = \{SP_i : i = 1, \ldots, n\}$; *where* $SP_i = \{VM_{i,k} : k = 1, ..., m_i\}$, *the corresponding processing rate* $\{U_{i,j} : i = 1, ..., n; j = 1, ..., m_i\}$ *for each virtual machine (e.g.,* $VM_{i,j}$*), and the allocation of* Γ *to the VMs within each server pool in* SP, *such that the QoS requirements of the requests* $\{Q_{i,j}, i = 1, ..., n; j = 1, ..., r_i\}$ *are guaranteed, and the power consumption of each server pool is minimized.*

This problem involves two intertwined problems: (a) how to judiciously pack service requests to a VM, (b) and how to determine the proper service rate to minimize the power consumption while guaranteeing the QoS for all classes of request types. Next, we discuss our analytical results for this problem, and then present our algorithm in details.

4 Preliminaries

This section presents several key analysis results with regard to QoS guarantees, requests multiplexing, and requests packing. These results form the basis of our approach.

4.1 Processing Rate Minimization for QoS Guarantee Using Request Reneging

Traditionally, M/M/1 queue [33] has commonly been adopted to represent the request processing procedure [2], as shown in Fig. 2a. Service requests arrive with a rate λ, wait in a queue with an infinite size, and are processed with a rate μ. Accordingly, the probability density function (PDF), and the cumulative distribution function (CDF) of the response time can be formulated as:

$$f(t) = (\mu - \lambda)e^{-(\mu-\lambda)t} \tag{2}$$

$$F(t) = 1 - e^{-t\mu(1-\frac{\lambda}{\mu})} \tag{3}$$

with a mean response time:

$$E[t] = \frac{1}{\mu - \lambda} \tag{4}$$

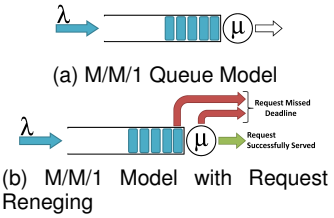


Figure 2. Processing Model.

The q-percentile of the response time t_q (i.e., t_q is larger than $q\%$ of all response times) has the following relationship:

$$1 - e^{-t_q\mu(1-\frac{\lambda}{\mu})} = \frac{q}{100} \tag{5}$$

To this end, given a request $\tau_{i,j}$'s arrival rate $\lambda_{i,j}$, deadline $D_{i,j}$, and completion ratio requirements $R_{i,j}$. In order to guarantee $Q_{i,j}$, the required service rate $\mu_{i,j}$ (when $\tau_{i,j}$ is hosted alone) is:

$$\mu_{i,j} = \frac{ln[\frac{1}{1-R_{i,j}}]}{D_{i,j}} + \lambda_{i,j} \tag{6}$$

The $\mu_{i,j}$ defined above can guarantee $Q_{i,j}$, i.e., no more than $(1 - R_{i,j})\%$ of the requests can miss their deadlines.

To save power consumption, it is desirable to discard a request, if it has a high probability to miss its deadline, so that we can save the precious resource for requests that are more likely to successfully complete in time. The problem nevertheless is how to discard these requests without compromising the QoS. To this end, we employ the $M/M/1$ queue with the reneging model [39], as illustrated in Fig. 2b.

As shown in Fig. 2b, according to the reneging model, each request is associated with a deadline. If a request is not fully served by its deadline, it is removed from the system. According to this model, there exists an provocative relationship among the request's deadline miss probability P_{miss}, arrival rate λ, processing rate μ, and deadline D, which can be formulated as [39]:

$$P_{miss} = \frac{(1-\rho)e^{\mu D(\rho-1)}}{1 - \rho e^{\mu D(\rho-1)}} \tag{7}$$

where $\rho = \frac{\lambda}{\mu}$. Accordingly, for a given $\lambda_{i,j}$, $R_{i,j}$, and $D_{i,j}$ of request $\tau_{i,j}$, we can derive $\mu^*_{i,j}$ that guarantees $Q_{i,j}$:

$$1 - R_{i,j} \leq \frac{(1 - \frac{\lambda_{i,j}}{\mu^*_{i,j}})e^{\mu^*_{i,j}D_{i,j}(\frac{\lambda_{i,j}}{\mu^*_{i,j}}-1)}}{1 - \frac{\lambda_{i,j}}{\mu^*_{i,j}}e^{\mu^*_{i,j}D_{i,j}(\frac{\lambda_{i,j}}{\mu^*_{i,j}}-1)}} \tag{8}$$

By judiciously removing the requests from the queue, we can guarantee the same QoS with lower processing rates. This conclusion is formally formulated in Theorem 1. (Due to the page limit, we moved the detailed proofs of Theorems 1 to 6 to Appendices A to F, respectively.)

Theorem 1. *A service request* $\tau = \{\lambda, D, R\}$; *where* λ, D, *and* R *refer to its arrival rate, deadline, and completion ratio requirement, respectively. Let* μ^* *and* μ *be the processing rates to satisfy* R *based on the M/M/1 queue model with and without request reneging, respectively. Then* $\mu \geq \mu^*$.

4.2 Request Multiplexing

When a service S_i has multiple request classes $\{\tau_{i,j}, j = 1, ..., r_i\}$, a common approach is to host each request class

on a single VM $\{VM_{i,j}, j = 1, ..., r_i\}$, respectively. The $R_{i,j}$-th percentile response time $t_{R_{i,j}}$ of $\tau_{i,j}$ can be formulated as:

$$t_{R_{i,j}} = \frac{1}{\mu_{i,j} - \lambda_{i,j}} ln[\frac{1}{1 - R_{i,j}}]. \quad (9)$$

When hosting each request class $\tau_{i,j}$ individually on $VM_{i,j}$, let the server pool $SP_i = \{VM_{i,j}, ..., VM_{i,r_i}\}$ has the processing rates $\{U_{i,1} = \mu_{i,1}, ..., U_{i,r_i} = \mu_{i,r_i}\}$, respectively. Then to satisfy each $Q_{i,j}$, the processing rates can be calculated according to equation 6:

$$\mu_{i,j} = \frac{ln[\frac{1}{1-R_{i,j}}]}{D_{i,j}} + \lambda_{i,j} \quad (10)$$

A better approach, withal, is to host multiple request classes in a single VM with a processing rate U that satisfies the QoS requirements of all hosted classes. We formulated this essential finding in Theorem 2.

Theorem 2. *For the i-type service requests $\{\tau_{i,j}, j = 1, ..., r_i\}$ hosted in a single node $\hat{VM}$, let the processing rate of $\hat{VM}$ be $\hat{U}$ and let*

$$U_{i,j} = \mu_{i,j} + \sum_{q=1, q\neq j}^{r_i} \lambda_{i,q}. \quad (11)$$

Then the QoS requirements for $\{\tau_{i,j}, j = 1, \ldots, r_i\}$ can be satisfied if $\hat{U} \geq \max_{j=1}^{r_i} U_{i,j}$.

From Theorem 2, when multiple classes of service requests are multiplexed in a single VM, the processing rate can be easily identified to ensure the QoS conditions for these requests. In the meantime, we show that request multiplexing helps improve resource utilization, as implied in the Theorem 3. Let us first define the processing rate $\Omega(SP_i)$ of a server pool SP_i.

Definition 1. The processing rate of a server pool, *denoted as $\Omega(SP_i)$; where $SP_i = \{VM_{i,1}, ..., VM_{i,r_i}\}$, is the sum of all VMs' required processing rates $\{U_{i,1}, ..., U_{r_i}\}$ in that server pool:*

$$\Omega(SP_i) = \sum_{j=1}^{r_i} \mu_{i,j} = \sum_{j=1}^{r_i} [\frac{ln[\frac{1}{1-R_{i,j}}]}{D_{i,j}} + \lambda_{i,j}] \quad (12)$$

Theorem 3. *Given the i-type service requests $\{\tau_{i,1}, \ldots, \tau_{i,r_i}\}$, let $SP_i = \{VM_{i,1}, \ldots, VM_{i,r_i}\}$ be all VMs, when each class of requests, $\tau_{i,j}$, is served separately with a dedicated $VM_{i,j}$, and let $\hat{SP_i} = \{\hat{VM}_{i,1}\}$ be a server pool with a single VM that serves all the requests simultaneously. Let $\Omega(SP_i)$ ($\Omega(\hat{SP_i})$) be the processing rate of the server pool SP_i ($\hat{SP_i}$, resp.) such that the QoS requirements for all $\{\tau_{i,j}, j = 1, ..., r_i\}$ are satisfied. Then $\Omega(SP_i) \geq \Omega(\hat{SP_i})$.*

Theorem 3 indicates that multiplexing different request classes on a single $VM_{i,j}$ can result in a smaller processing rate for the server pool, provided that the processing rate is feasible on $VM_{i,j}$; i.e., required processing rate must not exceed the VM's maximum capacity $C_{i,j}$. If only one VM cannot accommodate all service requests without compromising their QoS requirements, we will need more than one VM. How can we allocate service requests to VMs and optimize their utilization? We address this question next.

4.3 Request Packing

From the discussions above, clustering multiple classes of requests into the same VM helps improve the resource usage. When more than one VM is needed, the question then becomes how to group different classes of requests into each VM to minimize the server pool processing rate $\Omega = \sum_i \sum_k U_{i,k}$, and thus to maximize the overall resource usage efficiency. Consider the following example with four request classes of the same type $\{\tau_1, \tau_2, \tau_3, \tau_4\}$, with $\lambda = \{60, 40, 50, 20\}$ request' Instances Per Second (IPS), and $\mu = \{120, 80, 70, 90\}$ IPS, respectively. μ_i is the minimum processing rate of request τ_i to satisfy its QoS, when it is allocated individually to a VM. To guarantee all the QoS requirements, two request-grouping strategies are shown in Table 1. The server pool's processing rates using both strategies are $\Omega(V_1, V_2) = 300$ and $\Omega(V_1', V_2') = 280$, respectively (derived based on Theorem 2). This example clearly shows that different requests' allocation strategies lead to server pools with different processing rates and thus utilizations.

Table 1
Processing rate comparison with different requests packing strategies

Packing Strategy 1		**Packing Strategy 2**	
***VM*$_1$**	***VM*$_2$**	***VM'*$_1$**	***VM'*$_2$**
$\tau_i = \{\tau_1, \tau_2\}$	$\tau_i = \{\tau_3, \tau_4\}$	$\tau_i = \{\tau_1, \tau_4\}$	$\tau_i = \{\tau_2, \tau_3\}$
$U_1 = 160$	$U_2 = 140$	$U_1' = 150$	$U_2' = 130$

We show in Theorem 6 that the general packing problem is $\mathcal{NP}$-hard in nature. Subsequently, we focus on the development of an effective and efficient heuristic solution for this problem. To this end, we have made a number of crucial observations based on a service allocation onto two servers, which we formulate in the following theorems.

Theorem 4. *Let $\Gamma_1 = \Gamma_{1,p} \bigcup \Gamma_{1,q}$; where $\Gamma_{1,p} = \{\tau_{1,p_1}, \tau_{1,p_2}, ..., \tau_{1,p_s}\}$, $\Gamma_{1,q} = \{\tau_{1,q_1}, \tau_{1,q_2}, ..., \tau_{1,q_s}\}$, and $\Gamma_{1,p} \bigcap \Gamma_{1,q} = \emptyset$. Assume $\Gamma_{1,p}$ and $\Gamma_{1,q}$ are mapped to two VMs with the same capacity, $VM_{1,p}$ and $VM_{1,q}$, respectively. For each $\tau_{i,j}$, let*

$$\phi_{i,j} = \mu_{i,j} - \lambda_{i,j}. \quad (13)$$

Let $U_{1,p}$ ($U_{1,q}$, resp.) denote the minimum processing rate for $VM_{1,p}$ ($VM_{1,q}$, resp.) that guarantees the QoS requirements of $\Gamma_{1,p}$ ($\Gamma_{1,q}$, resp.). Then the processing rate for the server pool $SP = \{VM_{1,p}, VM_{1,q}\}$, i.e. $\Omega(SP) = U_{1,p} + U_{1,q}$, is minimized if the quantity given in equation 14 is minimized:

$$\Phi = \max_{\tau_{1,p} \in \Gamma_{1,p}} \phi_{1,p} + \max_{\tau_{1,q} \in \Gamma_{1,q}} \phi_{1,q}. \quad (14)$$

Theorem 4 means that to minimize the processing rate for a server pool SP_i, we need to minimize the sum of maximum $\phi_{i,j}$ (defined in equation 13) for the services allocated to each node. A simple heuristic is, hence, to sort all requests classes $\{\tau_{i,j}, j = 1, ..., r_i\}$ according to their $\{\phi_{i,j}, j = 1, ..., r_i\}$, and allocate as many high-ranking classes as possible on the same VM. Specifically, we have the following theorem.

Theorem 5. *Let $\Gamma_i = \{\tau_{i,1}, \tau_{i,2}, ..., \tau_{i,r_i}\}$ be mapped to two VMs with the same capacity, $VM_{i,p}$ and $VM_{i,q}$. Let $\tau_{i,k} \in \Gamma_i$ and let*

$$\Gamma_{i,k}^h = \{\tau_{i,j} \in \Gamma_i | \phi_{i,j} > \phi_{i,k}\}. \quad (15)$$

Then the processing rate for the server pool $SP_i = \{VM_{i,p}, VM_{i,q}\}$, i.e., $\Omega(SP_i) = U_{i,p} + U_{i,q}$, is minimized if $\tau_{i,k}$ is the one with the smallest *$\phi_{i,k}$ such:*

- $\Gamma^h_{i,k}$ *are feasibly allocated to one server (e.g.,* $VM_{i,p}$*);*
- $\Gamma^h_{i,k} + \{\tau_{i,k}\}$ *cannot be feasibly allocated to the same server (*$VM_{i,p}$*) simultaneously;*
- $\tau_{i,k}$ *is feasibly allocated to another server (e.g.,* $VM_{i,q}$*).*

Note that Theorem 5 helps to identify the optimal service packing solution for two VMs. However, if there are more than two VMs, finding the optimal solution becomes substantially more complicated due to the trade-off between minimizing the maximum value of ϕ for a VM, and the total number of needed VMs. In Theorem 6, we show that the service packing problem involving more than two VMs is NP-hard

Theorem 6. *Let* $SP_i = \{VM_{i,1}, VM_{i,2}, \ldots, VM_{i,m_i}\}$ *be a server pool with* $m \geq 3$*, hosts a set of requests* $\Gamma_i = \{\tau_{i,1}, \tau_{i,2}, ..., \tau_{i,r_i}\}$ *from the same service type* S_i*, but with different QoS constraints. Assume that all VMs in* SP_i *have the same capacity* C*. Then packing the request set* Γ_i *in the server pool* SP_i *such that the server pool processing rate* $\Omega_i = \sum_{j=1}^{m_i} U_{i,j}$ *is minimized is NP-hard.*

5 The Green Workload Packing and Consolidation (GWPC) Algorithm with Statistical Guarantees

We are now ready to discuss our approach for power consumption minimization in cloud data centers with guaranteed QoS. Inasmuch as the overall power consumption of a server pool depends on both the processing rates of its VMs, and the static power consumption of the physical servers hosting those VMs (see equation 1), to solve Problem 1, we need to minimize the number of VMs, and their processing rates. Thus, we develop an algorithm, called *Green Workload Packing and Consolidation algorithm (GWPC)* to allocate VMs and map service requests onto them.

First, GWPC intends to consolidate multiple request classes to the same VM. As shown in Theorem 2, when multiple classes are hosted in the same VM, the total processing rate of a server pool can be greatly reduced, which helps to minimize the dynamic power consumption of the physical cores on which VMs are allocated. Moreover, consolidating multiple request classes in a single VM reduces the total number of needed VMs, which in turn minimizes the total number of hosting physical machines and their static power. Second, GWPC adopts the reneging model to judiciously expunge service requests. Specifically, in Fig. 3, the required processing rate (e.g., μ_j) for each class of requests (e.g., τ_j) is calculated based on the reneging model (line 1). We then sort all requests based on $\phi_j = \mu_j - \lambda_j$ in a decreasing order (line 2), and pack the requests from the list to VMs with the capacity (i.e. the maximum processing rate) of C (line 3 to 13). Theorem 6 clearly demonstrates that Problem 1 is NP-hard, so we resort to the traditional first-fit bin-packing algorithm to pack the requests, and minimize the number of VMs. This helps reduce the static power consumption of the server pool. Sorting the requests according to the value of ϕ is a good heuristic with a basis presented in Theorem 5. It helps minimize the processing rate of each VM, and thus the dynamic power consumption of the server pool. In what follows, we first introduce the prototype we have developed, we then present the experiments and results we obtained based on this prototype.

Input: A set of request classes $\Gamma = \{\tau_j, j = 1, \ldots, r\}$ of a service S, each τ_j has corresponding λ_j and Q_j ($\{D_j, R_j\}$). A server pool SP of virtual servers $VM = \{VM_k, k = 1, \ldots, m\}$ with the same capacity C serve the request classes from service S.
Output: The request classes allocation.

```
1: Calculate each μ_j to satisfy Q_j based on (8);
2: dif_vect ← Sort the φ = μ − λ in the decreasing order;
3: for All requests τ_j, j = 1, 2, ..., r do
4:    for All VMs VM_k, k = 1, 2, ..., m do
5:       Add τ_j into node VM_k;
6:       Calculate VM_k's required processing rate U_k based on (2);
7:       if (U_k ≤ C) then
8:          Remove τ_j from dif_vect;
9:          Break the loop and pack the next request;
10:      else
11:         Remove τ_j from the current node VM_k;
12:      end if
13:   end for
14:   if (Request τ_j did not fit in any available VM) then
15:      Allocate a new VM and pack request τ_j;
16:   end if
17: end for
```

Figure 3. Green Workload Packing and Consolidation (GWPC) algorithm

6 The Green Cloud Computing Prototype (GCCP)

We developed a private Green Cloud Computing Prototype (GCCP) conforming to the industrial standards applied in practice. Specifically, GCCP's architecture model conforms to the model introduced by IBM in [40], whereas GCCP's infrastructure organization model conforms to the well-known cloud providers, such as Google Compute Engine, Amazon EC2, Rackspace, and Microsoft Windows Azure. GCCP consists of four functional modules implemented in Java and running on management nodes. The system workflow is automated and controlled using Python scripts at the system level, and bash scripts at the Linux nodes level. Management nodes are VMs launched using the open source Kernel Virtual Machine (KVM) hypervisor, and managed using the open source Webvirtmgr. KVM hypervisor is hosted onto two Dell Precision T1500 machines with Quad-Core Intel $i-5$ CPU, 16 GB 1333 MHz $DDR3$ memory, and a 300 GB SATA Disk Drive. Management VMs run Ubuntu Server Linux 12.04.5 LTS Precise Pangolin release with kernel version 3.2.0.76. Next, we introduce each functional module in more detail.

User Input Module This module allows users to define service types having request classes with different QoS requirements. In our experiments, we defined several scientific and cloud services. For example, we implemented a memory-intensive service modeled with a Matrix MULtiplication (MMUL) Java application, using the open source lightweight Apache Common Mathematics Library [41]. And we implemented a CPU-intensive service type modeled with an one-Dimension Fast Fourier Transform (1-D FFT) Java application, using the open source multi-threaded (FFT) library Jtransforms [42]. The submodule *Unified Workload Generation Engine (UWGE)* models and generates request instances of different service types and classes accordingly.

Service Management Module This module takes inputs from the *User Input Module* and schedules/dispatches workload among computing resources by the *Scheduler & Resource Allocator/Request Dispatcher* submodules. To investigate the performance of our approach, we implemented the following workload mapping and scheduling algorithms, which also employ failure reneging: (1) **Split**: denoted as "SPT,", the traditional method by which each request class is hosted in a separate VM [10] [43]; (2) **Random**: denoted as "RND,", the method that fills multiple request classes randomly into a VM; (3) **First-fit-decreasing**: denoted as "FFD,", the bin-packing method [44] with service classes combined based on the decreasing order of their required processing rates; (4) **Green Workload Packing and Consolidation**: denoted as "GWPC," our proposed method.

Infrastructure Management Module This module is in charge of managing the computing infrastructure, hardware and software resources, of the cloud cluster. We employ Citrix XenServer 6.5 platform, which is based on the hypervisor Xen [32], to manage the physical resources on an HP Workstation Z800 with two Intel Xeon Six-Core $E5645$ ($2.40\ GHz$, $12\ MB$ cache), $1333\ MHz\ DDR3$ memory of size ($32\ GB$), and 1 TB disk space. We also developed a *Cloud Orchestrator* submodule that communicates with the *Scheduler & Resource Allocator* and XenServer to automate the procedures of VM's creation, provisioning, and configuration, and to make VMs available online for the *Request Dispatcher* onto which requests are forwarded from the UWME submodule. In the first set of experiments, the *Cloud Orchestrator* configures each VM with $2\ GB$ memory, $20\ GB$ disk storage, and a dedicated physical core with an adjustable maximum processing rate according to a given capacity using Xen's credit scheduler [32].

Performance Monitoring Module This module monitors the system's performance, and collects performance statistics, as well as other measurement readings through three major submodules.

The *Power Metering* submodule measures the static and dynamic power consumption of server pools hosted by GCCP under different configurations and running conditions. To measure the actual power consumption, we used an AC/DC Fluke $i410$ current clamp meter with an output of $1mVolts(mV)/Amps(A)$, connected to an Agilent34401A multimeter with a resolution of $+/-120mWatts(mW)$. This submodule automates the power reading process using a C program running within Ubuntu Linux 12.04.5 LTS on a dedicated Dell desktop that communicates with the multimeter through a serial cable to automatically record electrical current readings. Consider a server pool $SP_i = \{VM_{i,1}, ..., VM_{i,m_i}\}$ allocated to physical cores $\{PCPU_{i,1}, ..., PCPU_{i,m_i}\}$, respectively. Recall that a single workstation in GCCP has at most 10 available physical cores (Note that we reserve 2 cores for Xen's Domain-0.) SP_i must be hosted by $\lceil \frac{m_i}{10} \rceil$ workstations, and the power consumption for SP_i is the total power consumption of these workstations. Now assume that $m_i \leq 10$. Based on (1), the power consumption P_i consumed by SP_i can be formulated as

$$P_i = \sum_{j=1}^{m_i} (P_{i,j}^d \varphi_{i,j}) + P_i^s, \tag{16}$$

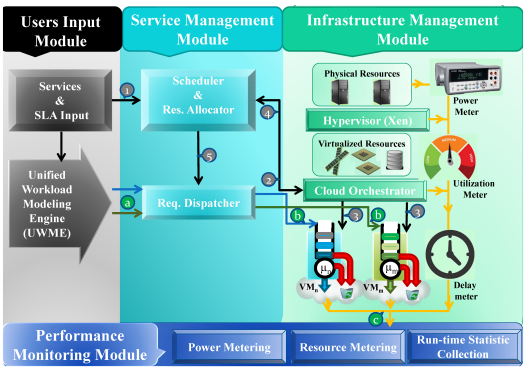


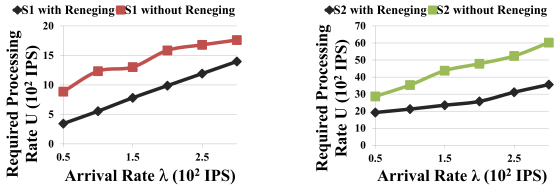
Figure 4. High Level Representation of GCCP. Steps 1 till 5 illustrate the process of initiating a new service and its request classes (step 1), running a packing and allocation algorithm (step 2), spawning and configuring VMs (step 3), returning VMs' Ids to the service manager module to be available to the request instances (steps 4 and 5). On the other hand, steps a, b and c illustrate the process of generating request instances, dispatching them to the allocated VMs, and collecting the system performance readings upon their completion.

where $P_{i,j}^d$ is the dynamic power when each core is 100% utilized, $\varphi_{i,j}$ is the utilization of $VM_{i,j}$, and P_i^s is the static power of the HP workstation when m_i physical cores are allocated to the VMs. To calculate P_i^s, we measured the drawn AC current (i.e, I^s) by the workstation when all m_i VMs are idle. Then the corresponding static power is $P_i^s = I^s \times 120V$. To calculate $P_{i,j}^d$, we used UWGE to generate enough request instances such that all VMs were kept busy and achieved 100% utilization, and we then measured the drawn AC current by the workstation to calculate the total power consumption, i.e., static and dynamic power. The difference accordingly, between the total and static measured power, is the total dynamic power consumption of m_i cores with 100% utilization. We further assume that cores hosting the same service type (e.g., S_i) with the same capacity size (e.g., C_i) consumes the same amount of dynamic power, and we thereupon divide the total dynamic power by the number of cores m_i to get $P_{i,j}^d$. As an illustration, let a server pool with a single VM. We measured its static power as $186W$. Its dynamic power consumption when running memory-intensive workload and CPU-intensive workload on a fully utilized core are $16.8W$ and $18W$, respectively.

The *Resource Metering* submodule leverages available system tools and/or our newly developed user-level tools to collect and/or measure resource usage, such as the amount of CPU usage, i.e. processing rates, consumed by each VM in terms of MHz (e.g., using the command $xentop$) and/or in terms of IPS (e.g., using scripts that parses the run-time logs generated by the *Run-time Statistic Collection* submodule, which is described next);

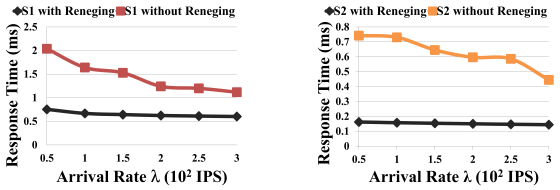
The *Run-time Statistic Collection* submodule collects and stores run-time statistics of dispatched request instances in log files based on the light-weight RAM Filesystem (Ramfs). Information collected by this submodule includes IDs of service types, request classes, instances and hosting VMs, along with instances' start times, finish times, QoS violations, completion ratios, and so on.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2642941, IEEE Transactions on Parallel and Distributed Systems



(a) S_1 : 256×256 MMUL (b) S_2 : 1×256 1-D FFT

Figure 5. Minimum required processing rates for guaranteed QoS using M/M/1 queue model with & without reneging for (a) S_1 and (b) S_2 types.



(a) S_1 : 256×256 MMUL (b) S_2 : 1×256 1-D FFT

Figure 6. Response Time Comparison for guaranteed QoS using M/M/1 queue model with & without reneging for (a) S_1 and (b) S_2 types.

7 Experiments and Results

In this section, we use experiments to validate our analytical findings, and to test the performance of the GWPC algorithm using the GCCP platform.

7.1 Performance with Request Reneging

We first empirically study the advantages of request reneging on minimizing the required processing rates and average response times with QoS guarantees. We also compare the performance of the system with and without request reneging using two different service types; i.e., memory-intensive (e.g., $S1$) and CPU-intensive (e.g., $S2$) types.

We generated two sets of classes with six different request classes each $\{\tau_{i,j} : i = 1, 2; j = 1, ..., 6\}$, with the first set from (S_1 : 256×256 MMUL), and the second set from ($S_2 : 1 \times 256$ 1-D FFT). Request classes from both types were set to have average arrival rates of $\{\lambda_{i,1} = 50, \lambda_{i,2} = 100, ..., \lambda_{i,6} = 300;\ where\ i = 1, 2\}$, completion ratios of $R_{i,j} = 95\%$, and deadlines randomly generated following a uniform distribution in the ranges $[5 - 15] \times 10^2 \mu s$ and $[2 - 6] \times 10^2 \mu s$ for S_1 and S_2 classes, respectively. All parameters and their values were arbitrarily chosen.

We generated 10^5 Instances Per Request class (IPR), and request instances of each class were executed in a VM with and without reneging. The capacity of the VM was set to 1800 IPS and 6000 IPS for S_1 and S_2, respectively. In each run, we calibrated the maximum processing rate of each VM by assigning a cap to each VM's Virtual CPU (VCPU) that limits the maximum amount of processing rates a VM receives from its allocated physical core. We then applied the traditional binary search method to find the minimum required processing rates that can meet the given QoS conditions. We repeated the experiment for each setting 10^4 times, and the average results are shown in Figs. 5 and 17.

Fig. 5 compares the minimum required processing rates with and without reneging under different arrival rates and QoS settings. We see that for both S_1 and S_2 requests, the minimum required processing rates with reneging are much lower than those without reneging. For example, when the arrival rate is 200 IPS, the minimum required processing rates with reneging for S_1's and S_2's are 1000 IPS and 4800 IPS, respectively. If no requests are reneged, the minimum required processing rates become 1600 IPS and 5600 IPS, respectively. This is an increase of 60% and 16.7% over its counterparts. The minimum required processing rates unsurprisingly increase with the increment of arrival rates. Nonetheless, the minimum required processing rates with reneging increase at a much slower rate, as clearly shown in Figs. 5. In average, the minimum required processing rates with reneging for S_1 and S_2 are 62% and 58% lower than those without reneging, respectively. These results evidently conform to the theoretical conclusion formulated in Theorem 1 that request reneging helps reduce the processing rates while guaranteeing the same QoS requirements.

We can also observe that request reneging helps lower service response times. As shown in Figs 6a and 6b, the average response times of S_1 and S_2 classes without reneging are always longer than those with reneging. On average, the average response times of S_1's and S_2's are 225% and 409% longer than those with reneging. As the arrival rates increase, the minimum required processing rates must increase to ensure the same QoS requirement. As a result, response times are reduced. While response times for requests without reneging change dramatically, as arrival rates increase, the response times for requests with reneging do not vary as significantly. This implies that requests with reneging can deliver service in a more stable manner in terms of response time variations. When comparing the response time improvement with reneging for the memory-intensive service S_1 and the CPU-intensive service S_2, we can see that S_1 benefits more than S_2 in term of the improvement for the minimum required processing rates and response times, as shown in Figure 5 and 17. We conjecture that this is due to the fact that the S_1 requests require longer I/O-related operations, and cannot be easily terminated for request reneging, which negatively affects their response times when compared with the CPU-intensive requests.

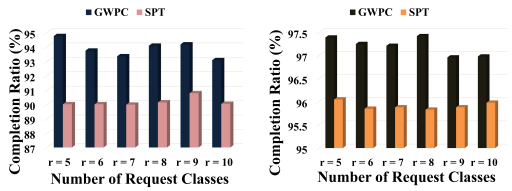
Request reneging to a great extent not only reduces processing rate requirements to guarantee the same QoS requirements as the one without reneging, but also results in a lower and more stable average response times, and is therefore a promising approach to achieve reliable and predictable performance.

7.2 Multiplexed vs. Split Request Processing

Having validated the performance improvement of request reneging, we now compare the completion ratios and average response times when serving requests in a multiplexed manner (e.g., GWPC) and a split manner (e.g., SPT).

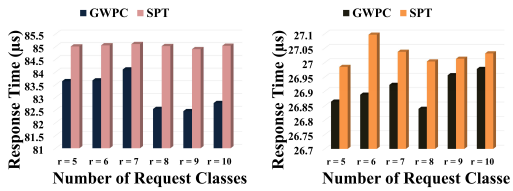
We generated two types of services, S_1 (128×128 MMUL) and $S_2 (1 \times 64$ 1-D FFT). We randomly generated six testing groups of request classes from each service type, with the number of classes in each group varying from $r = 5$ to $r = 10$, i.e. $\{\tau_{i,j}; i = 1, 2; j = 1, ..., r\}$. The arrival rates and deadlines in each request class were randomly

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2642941, IEEE Transactions on Parallel and Distributed Systems



(a) S_1 : 128×128 MMUL (b) S_2 : 1×64 1-D FFT

Figure 7. Performance of completion ratios between request multiplexing & splitting using requests of (a) S_1 & (b) S_2 types.



(a) S_1 : 128×128 MMUL (b) S_2 : 1×64 1-D FFT

Figure 8. Performance of average response times between request multiplexing & splitting using requests of (a) S_1 & (b) S_2 types.

generated with the average following a uniform distribution in the ranges $[20 - 120]$ IPS and $[5 - 6] \times 10^2 \mu s$, respectively. We set 90% and 95% completion ratios for S_1 and S_2, respectively. In each test, we generated 10^5 IPR for each of S_1 and S_2 classes, separately applied the GWPC and SPT methods on those instances using VMs with a capacity set to 10^4 IPS, and calculated the achieved completion ratios and average response times in each run for each request class. All these parameters and their values were arbitrarily chosen. We repeated the experiment for each setting 10^4 times, and the average results are shown in Figs. 7 and 8.

Fig. 7 compares the completion ratios achieved by the multiplexed and split approaches under different experiment settings. As shown, both methods successfully guarantee the required completion ratios for S_1 and S_2. Nonetheless, GWPC can achieve a much higher average completion ratios than SPT. Note that, for the test cases when there are eight different request classes (i.e. $r = 8$), GWPC achieves a completion ratio of 94% for S_1 and 97.5% for S_2, in comparison with 90.2% for S_1 and 95.8% for S_2 achieved by SPT. Those results comply with the conclusion in Theorem 3 that request multiplexing helps reduce required processing rates without compromising QoS requirements. When all VMs have the same capacity, GWPC can unsurprisingly lead to better completion ratios in all test cases. GWPC can on average achieve an average completion ratios of 93.87% for for S_1 and 97.2% for S_2, whereas SPT can only achieve 90.18% and 95.92% for S_1 and S_2, respectively.

Request multiplexing also results in less average response times than those in SPT, as shown in Fig. 8. The average response times in the multiplexed approach outperform those in SPT in all test cases. The reason for such improvement is that GWPC can efficiently utilize computing resources among different request classes, as a result of request reneging and multiplexing. Moreover, allocating a smaller number of VMs reduces the overhead on the Virtual Machine Manager (VMM), i,e., the hypervisor, especially with memory-intensive workloads, which demand more privileged operations, such as memory accesses, context switches, system calls and interrupts [45]. For instance, in Fig. 8, when the number of classes is $r = 10$, GWPC results in an average response time that is $2.26\mu s$ less than those in SPT for S_1 classes, but for S_2 and with the same number of request classes $r = 10$, GWPC shows only $0.06\mu s$ better average response time than that of the SPT.

overall, our experiments show that request multiplexing not only guarantees QoS requirements, but can also achieve higher completion ratios than required. It can better utilize computing resources than SPT does, especially with the memory-intensive service types.

7.3 Performance under Different Service Utilizations

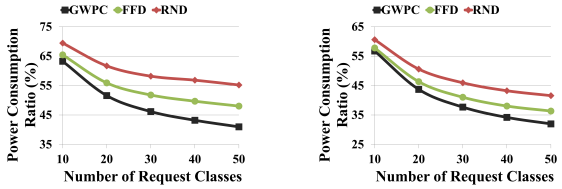
In this section, we analyze the performance of GWPC in terms of power consumption and processing rate demand, compared with SPT, FFD, and RND, which are described in Section 6.

We generated five groups of test cases, each of which has a different number of request classes $\{\tau_{i,j} : i = 1, 2; j = 1, ..., r_i\}$; where $r_i = 10, 20, ..., 50$, from two service types, i.e., (S_1 : 128×128 MMUL) and ($S_2 : 1 \times 64$ 1-D FFT). The arrival rates and completion ratios were randomly generated with the average following uniform distributions in the ranges $[20 - 500]$ IPS and $[90\% - 95\%]$, respectively. We recall that, from equation (8), when the arrival rate is a constant value, the smaller the deadline, the higher the required processing rate is. Thence, as the deadline reduces, the processing rate $\mu_{i,j}$ increases, and then the service utilization increases. We accordingly varied request utilization by changing the intervals from which we randomly picked the deadlines. For each set of request classes, we varied the deadline range among four intervals, starting from $500\ \mu s$ and $600\ \mu s$ with interval length of $50\mu s$ and $100\mu s$ for S_1 and S_2, respectively.

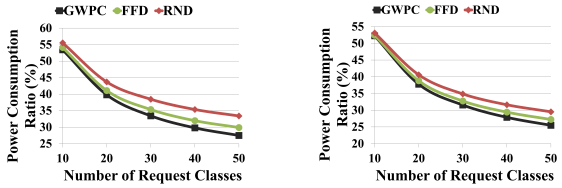
GWPC, SPT, FFD, and RND were tested using the same test cases. In each run, we generated 10^5 IPR with reneging on the VMs $VM_{i,j}; i = 1, 2; j = 1, 2, ..., m_i$. We repeated each run 10^4 times, calculated the average results, normalized them to the results by SPT, and presented them in Figs 9 to 12. Specifically, Figs 9 and 10 show the power consumption of different approaches, and Figs 11 and 12 compare the total minimum processing rates required by each server pool (i.e., $\Omega(SP_i)$) to satisfy the given QoS constraints.

From Figs. 9 and 10, we can immediately observe that GWPC outperforms the other three approaches under the different testing condition. For example, for S_1's request in Fig. 9, and when the deadlines are within the range of [50 - 550] μs, and the number of classes is 30, the power consumption by GWPC is about 46% of that by SPT. Whereas it is about 52% and 58% for FFD and RND of that by SPT, respectively. We can also see that GWPC's power-saving performance increases with increasing the number of classes, as well as with increasing the tightness of their deadline ranges. As the number of classes increases, and the solution space to map requests to different VMs increases, GWPC can henceforth take advantage of the bigger solution space, and achieve better power-saving performance. Correspondingly, when the deadlines are long, or the workload intensity is light, the differences among packing approaches become smaller, and therefrom the power consumption patterns

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2642941, IEEE Transactions on Parallel and Distributed Systems

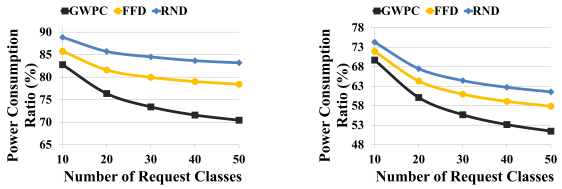


(a) $D_{1,j} \in [500 - 550]\mu s$ (b) $D_{1,j} \in [550 - 600]\mu s$

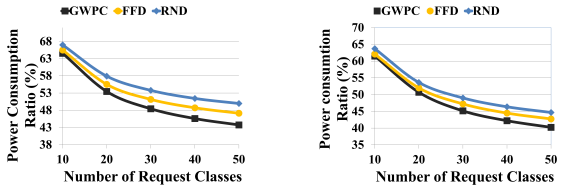


(c) $D_{1,j} \in [600 - 650]\mu s$ (d) $D_{1,j} \in [650 - 700]\mu s$

Figure 9. Power-saving performance normalized to that of SPT for S_1 : 128×128 MMUL service type with different deadline ranges.

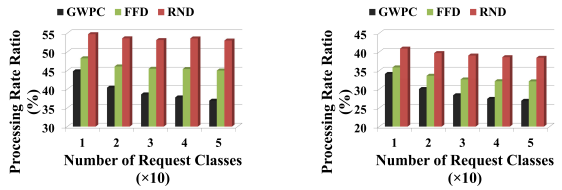


(a) $D_{2,j} \in [6 - 7] \times 10^2 \mu s$ (b) $D_{2,j} \in [700 - 800]\mu s$

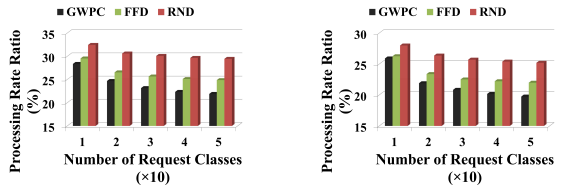


(c) $D_{2,j} \in [800 - 900]\mu s$ (d) $D_{2,j} \in [900 - 1000]\mu s$

Figure 10. Power-saving performance normalized to that of SPT for S_2 : 1×64 1-D FFT service type with different deadline ranges.



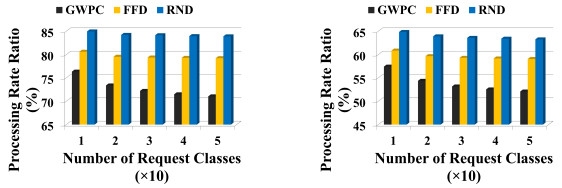
(a) $D_{1,j} \in [500 - 550]\mu s$ (b) $D_{1,j} \in [550 - 600]\mu s$



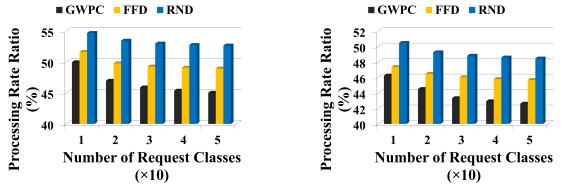
(c) $D_{1,j} \in [600 - 650]\mu s$ (d) $D_{1,j} \in [650 - 700]\mu s$

Figure 11. Processing Rate Performance normalized to that of SPT for S_1 : 128×128 MMUL service type with different deadline ranges.

become similar. As the deadlines become larger and larger, the workload becomes heavier and heavier, and thusly GWPC can greatly benefit from its effective request reneging and packing methods, and can consequently achieve



(a) $D_{2,j} \in [600 - 700]\mu s$ (b) $D_{2,j} \in [700 - 800]\mu s$



(c) $D_{2,j} \in [800 - 900]\mu s$ (d) $D_{2,j} \in [900 - 1000]\mu s$

Figure 12. Processing Rate Performance normalized to that of SPT for S_2 : 1×64 1-D FFT service type with different deadline ranges.

higher and higher power-saving performance. For example, from Figs. 10(d) and 10(a), the power consumption ratios achieved by GWPC increase from 61% to 82.8% as request utilizations decrease (e.g., the deadline range changed from $[650 - 700]\mu s$ to $[500 - 550] \times 10^2 \mu s$).

The behaviors of the minimum required processing rates and power consumption rates are closely related. Thus, it is not unforeseen to observe the similar phenomena, when studying our experimental results in terms of the minimum processing rates of the server pools. The total processing rates required by GWPC are lower than those required by the other three approaches for both types of services and under different testing cases, as shown in Figs. 11 and 12. We can also notice,from same figures, that the improvement of GWPC over the other three approaches increases, when the class number for each service type increases, as well as when the tightness of the deadline ranges increase. Recall that, in Theorem 6, we prove that request packing is an NP-hard problem [46], when a server pool SP_i needs more than two VMs. Yet, our experimental results clearly show that the heuristic proposed in GWPC, i.e. packing requests ordered by $\phi_{i,j}$ (see equation 13) is much more effective than the one (i.e. FFD) that packs requests ordered by their individual processing rates (e.g., $\mu_{i,j}$) only. For example, in Figure 12 for S_2 classes, when the class number is 30, the average total processing rate by GWPC is about 85% of that by FFD.

7.4 Performance under Different Server Capacities

Different server capacities affect how many requests can be accommodated in a single server, and how many servers are needed to ensure the QoS requirements for a given set of service requests. In this section, we investigate how the performance of different allocation and packing approaches varies with different server's capacities, (e.g. $C_{i,j}$).

We generated two types of services, S_1 (128×128 MMUL) and S_2 (1×64 1-D FFT), with five testing groups of requests from each service type, and with each group has a different number of classes $\{\tau_{i,j}; i = 1, 2; j = 1, ..., r; r = 10, 20, ..., 50\}$. The average arrival rates were randomly varied with the average following a uniform distribution in the range of $[20 - 500]$ IPS

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2642941, IEEE Transactions on Parallel and Distributed Systems

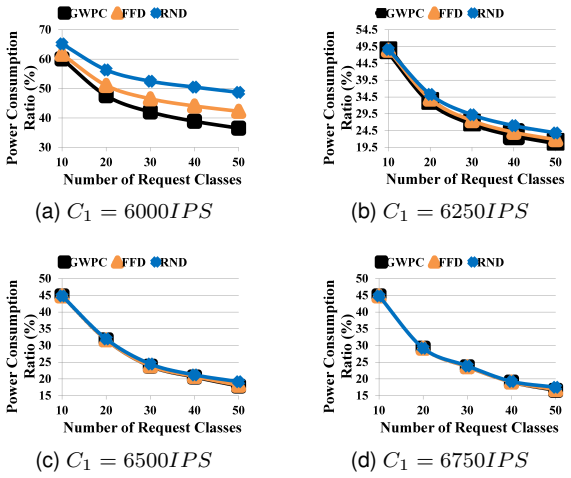
and $[500 - 1500]$ IPS for S_1's requests and S_2's requests, respectively. The deadlines of the requests were chosen randomly from the interval $[500 - 600]\mu s$. We varied the server's capacities from 6000 IPS to 6750 IPS, and from 6000 IPS to 12000 IPS for S_1 and S_2, respectively. We configured each VM with 2 GB memory, 20 GB disk storage, and a dedicated physical core with an adjustable maximum processing rate, i.e., capacity, according to a given value (e.g. $C_{i,j}$) using Xen's credit scheduler [32]. For example, to pin a VCPU to a single physical core, i.e., CPU, we can use: *xl vcpu-pin "VM Name" "VCPU-ID" "CPU ID"*. Then to adjust the processing rate of a VCPU in a VM according to a given capacity $C_{i,j}$ in terms of RPS, we can use Xen Credit scheduler that assigns a Cap to the VCPU of a VM, which limits the maximum amount of the physical CPU that VCPU can use. For example, a VM with a 100 Cap means that the VM can consume up to a 100% of its CPU's maximum processing rate: *xlsched -credit -d " VM Name" -c "Cap-Value"*. The arbitrarily chosen parameters and their values are summarized in Tables 2(a)(b) and (c). For each experimental setting, we repeated each run 10^4 times, collected the total power consumption and the total minimum processing rates, and presented the average results normalized to those by SPT in Figs. 13 to 16.

Figs. 13 and 14 show how power consumption of different approaches vary with different capacities, and Figs. 15 and 16 show how total required processing rates change with different capacities. From the results, we can see that GWPC outperforms others in terms of both power consumption rates and total processing rate demands under different server capacities. As shown in the figures, when the VMs' capacities increase from 6500 IPS (Fig. 13(a)) and from 6500 IPS (Fig.14(a)) to 6750 IPS (Fig.13(d)) and to 12000 IPS (Fig.14(d)), the improvement of GWPC over RND and FFD diminishes for S_1 (S_2) service types. This is because when the VMs' capacities increase, more request classes can be hosted together in the same VM, and thus all multiplexing approaches show similar performance. Nevertheless, we can see that the performance improvement of power-saving and the processing rate demands by the multiplexing approaches over SPT approach continue to improve as the server's capacities grow larger. For example, in Fig. 13(a), when server capacity is 6000 IPS and the number of classes is 30, the power consumption by GWPC is about 53% of that by SPT. In Fig.13(d), when the server capacity is 6750 IPS and the number of classes is 30, the power consumption by GWPC becomes about 24% of that by SPT. This again conforms to the theoretical conclusion in Theorem 3.

7.5 Validation Using Cloud Benchmarks

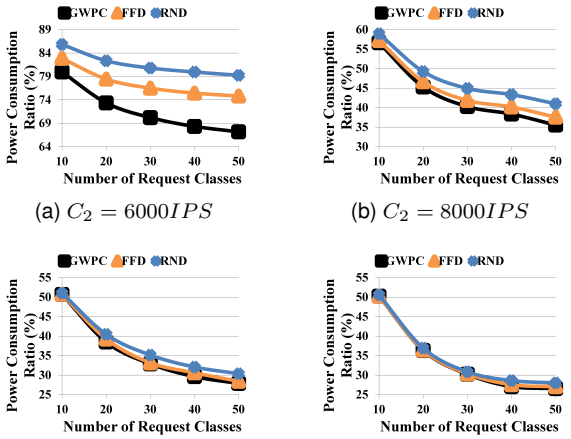
We further evaluate our methods using the Data Caching Benchmark, i.e. a benchmark that emulates the behavior of a Twitter caching server, from the benchmark suite of cloud services, CloudSuite [36]. The benchmark assumes strict quality of service guaranties such as, 95% of the request must finish within 200 ms.

We exploited the GCCP to bootstrapped two VMs with 10 GB memory capacity, and a single VCPU pinned to a single physical core in each VM. We then implemented a



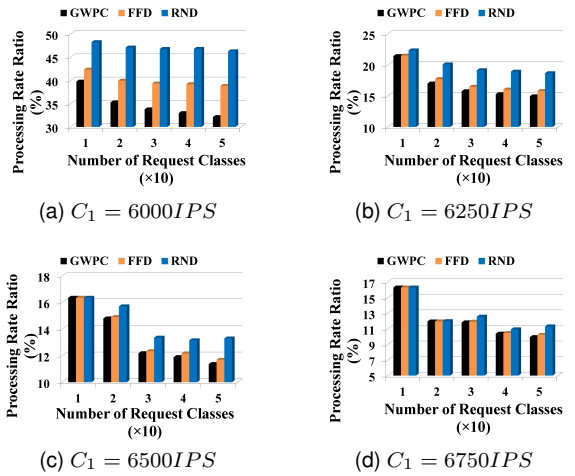
(a) $C_1 = 6000IPS$ (b) $C_1 = 6250IPS$ (c) $C_1 = 6500IPS$ (d) $C_1 = 6750IPS$

Figure 13. Power-Saving Performance normalized to that of SPT for S_1 : 128×128 MMUL service type with different server capacities.



(a) $C_2 = 6000IPS$ (b) $C_2 = 8000IPS$ (c) $C_2 = 10000IPS$ (d) $C_2 = 12000IPS$

Figure 14. Power-Saving Performance normalized to that of SPT for S_1 : 1×64 1-D FFT service type with different server capacities.

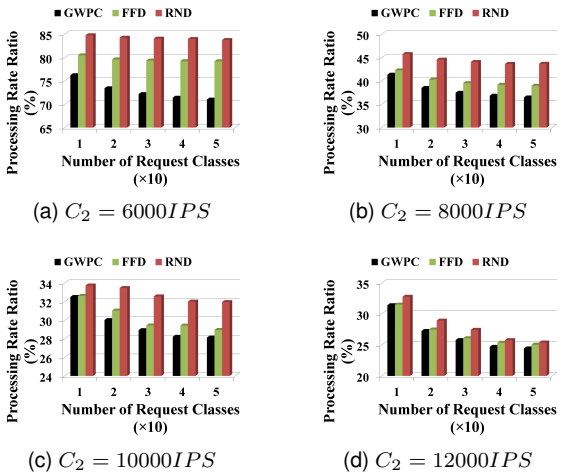


(a) $C_1 = 6000IPS$ (b) $C_1 = 6250IPS$ (c) $C_1 = 6500IPS$ (d) $C_1 = 6750IPS$

Figure 15. Processing Rate Performance normalized to that of SPT for S_1 : 128×128 MMUL service type with different server capacities.

Memcached server—a distributed memory object caching system that speeds up dynamic web applications by alleviating a database's delay—on the first VM with a single worker (e.g., a single execution queue) to process the data

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2642941, IEEE Transactions on Parallel and Distributed Systems



(a) $C_2 = 6000IPS$ (b) $C_2 = 8000IPS$

(c) $C_2 = 10000IPS$ (d) $C_2 = 12000IPS$

Figure 16. Processing Rate Performance normalized to that of SPT for $S_1 : 1 \times 64$ 1-D FFT service type with different server capacities.

Table 2
Power-Saving Performance of GWPC with Different Server Capacities

(a) Services & Nodes Specifications

S_i	SP_i 's Size (m_i Nodes)	C_i textbf(IPS)
$S_1 : 128 \times 128$ MMUL	$m_1 = \{10, 20, 30, 40, 50\}$	$C_1 = (6, 6.25, 6.50, 6.75) \times 10^3$
$S_2 : 1 \times 64$ 1-D FFT	$m_2 = \{10, 20, 30, 40, 50\}$	$C_2 = (6, 8, 10, 12) \times 10^3$

(b) Requests Specifications for S_1 and S_2

$\tau_{i,j}$	$\lambda_{i,j}$ (IPS)	$V_{i,j}$	C_{ij} (IPS))	Run ID
$(\tau_{1,1} \dots \tau_{1,r1})$	$[20 - 500]$	$V_{1,1} \dots V_{1,m1}$	6×10^3	(0 to 4)
$(\tau_{1,1} \dots \tau_{1,r1})$	$[20 - 500]$	$V_{1,1} \dots V_{1,m1}$	8×10^3	(5 to 9)
$(\tau_{1,1} \dots \tau_{1,r1})$	$[20 - 500]$	$V_{1,1} \dots V_{1,m1}$	10×10^3	(10 to 14)
$(\tau_{1,1} \dots \tau_{1,r1})$	$[20 - 500]$	$V_{1,1} \dots V_{1,m1}$	12×10^3	(15 to 19)
$(\tau_{2,1} \dots \tau_{2,r2})$	$[500 - 1500]$	$V_{2,2} \dots V_{2,m2}$	6×10^3	(20 to 24)
$(\tau_{2,1} \dots \tau_{2,r2})$	$[500 - 1500]$	$V_{2,2} \dots V_{2,m2}$	8×10^3	(25 to 29)
$(\tau_{2,1} \dots \tau_{2,r2})$	$[500 - 1500]$	$V_{2,2} \dots V_{2,m2}$	10×10^3	(30 to 34)
$(\tau_{2,1} \dots \tau_{2,r2})$	$[500 - 1500]$	$V_{2,2} \dots V_{2,m2}$	12×10^3	(35 to 39)

(c) Shared Parameters among Requests

$R_{i,j}$ (%)	IPR_{ij}	Run Repetition	r_i	$D_{i,j}(10^2 \mu s)$
$[90 - 95]$	10^5	10^4	$(10, 20, 30, 40, 50)$	$[5 - 6]$

caching requests, i.e., *Get* or *Set*. The caching requests are generated by a Memcached clients implemented within the workload engine UWME, and the Memcached server processes those requests. The UWME replicates those requests, and forward them to the second Memcached server running on the second VM with reneging. The Performance Monitoring Module collects the results on both servers.

We considered two main service types generated by the UWME; i.e., *Get* (e.g., S_1), and *Set* (e.g., S_2). We generated one set of request with nine classes each from each service type, $\{\tau_{i,j} : i = 1, 2; j = 1, ..., 9\}$, with arrival rates following the exponential distribution and averages of $\{1000, 2000, ..., 9000\}\}$ RPS. The completion ratios were set to be $R_{i,j} = 95\%$. The deadlines in both sets were set such that the maximum achieved throughput does not violate the target QoS requirements, when the requests are served with the Memcached server without reneging.

7.5.1 Data caching workload with Request Reneging

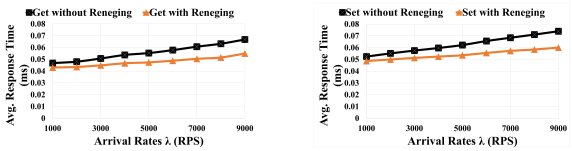
We first compare the the average response times under QoS guarantees with and without request reneging. We generated 10^7 *I*nstances *P*er *R*equest class (IPR), and processed these instances on both VM_1 and VM_2. The average results are shown in Figs. 17.

Our experimental results clearly show that request reneging can significantly reduce service response times. As shown in Fig. 17(a), the average response times of Get and Set classes without reneging are always longer than those with reneging. On average, the average response times of Get requests are 8%, 14%, and 17%, longer than those with reneging for Get requests classes with the arrival rates $\{\lambda_{1,1} = 1000, \lambda_{1,5} = 5000, \lambda_{1,9} = 9000\}$ RPS, respectively. We also observe similar trend in Fig. 17(b), where the average response time of Set requests is on average 14% longer than those with reneging. When the arrival rates increase for S_1's and S_2's requests, the minimum required processing rates must increase to guarantee the same QoS requirement, as the waiting times of requests increased. From Figs. 17(a), and 17(b), we can also observe that the response time improvement by the reneging over non-reneging server increases with the arrival rate of the requests. For example, in Fig. 17(b), and when the arrival rate of Set requests is 5000 RPS, the response time by the reneging server is 14% shorter than the non-reneging server, and it becomes 19% when the arrival rate increases to 9000 RPS.

7.5.2 Energy-saving performance of Data Caching Workload with Request Reneging

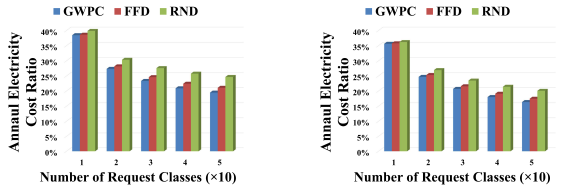
To study the potential electricity cost savings of our approach, we tested GWPC, SPT, FFD, and RND, using the same test cases defended above, with different request classes each time ranging from 10 to 50. We also varied the completion ratio for each class randomly from interval of [90%,99%]. We assume that each HP workstation with a total memory capacity of 32GB can hold up to 3 Memcached servers. We generated 10^5 IPR for each request class, and repeated each run 10^4 times with and without reneging. The results are taken among different completion ratios and service utilizations, averaged and normalized to the results by SPT and presented them in Fig. 18. More details about power measurement and electricity cost calculation can be found in section 6.

The annual electricity cost-saving performance of GWPC follows a similar pattern of the power-saving performance, and total processing rate as our experimental results showed before. First, compared with SPT, all three request multiplexing approach (i.e. GWPC, SPT, and RND), by sharing servers and reneging requests, can improve the processing efficiency and reduce the electricity cost significantly. In addition, as the number of classes increases, the improvement becomes more significant. For example, when the number of classes increases from 10 to 50, the annual electricity cost ratios over SPT by GWPC, FFD and RND decrease from 38 (35, resp.), 39 (36, resp.), and 40 (36, resp.) to 19 (16, resp.), 21 (17, resp.), and 25 (20, resp.) for S_1 (and S_2, resp.), respectively. Moreover, we can see that GWPC outperforms FFD and RND. According to Figs 18(a) and 18(b), GWPC on average saves around 4% and 3% more than FFD and 12% and 10% more than RND for S_1's and S_2's requests, respectively. Furthermore, such an improvement increases while the number of classes increases. For example, when the number of classes is 10, the relative



(a) Get's Request Classes (b) Set's Request Classes

Figure 17. Get's (a) and Set's (b) average response times under different arrival rates with and without request reneging.



(a) Get's Request Classes (b) Set's Request Classes

Figure 18. Energy-saving performance of GWPC normalized to that of SPT using Get's and Set's request classes under various QoS condition

improvement of GWPC over FFD and RND is 2.6% (2.8%, resp.) and 5.3% (2.8%, resp.) for S_1 (S_2, resp.) requests, respectively. When the number of classes is 50, the relative improvement of GWPC over FFD and RND becomes 10.5% (6.25%, resp.) and 31.6% (25.0%, resp.) for S_1 (S_2, resp.) requests, respectively. This is because that when the number of classes increases, the solution space to map requests to different VMs increases, and GWPC consequently can take advantage of the bigger solution space, and achieve better power-saving performance. Overall, our experimental results have clearly demonstrated that request reneging with data caching workload results in lower average response times, and less energy costs comparing to its counterpart. GWPC is therefore a promising approach that should be studied and applied to more complicated cloud workloads.

8 Conclusion

In this paper, we present a novel approach that can be applied in virtualized data centers to improve resource utilization and minimize power consumption when delivering cloud services with statistically guaranteed QoS. The effectiveness and efficiency of our approach is rooted in two facts: (1) our approach can effectively remove the potential failure requests as soon as possible to improve resources usage; (2) our approach allows requests with different QoS requirements to be served on the same VM. To our best knowledge, this is the first approach by which different requests with different QoS guarantees can be hosted on a single node in order to further increase resource utilization and reduce power consumption. We present several interesting characteristics of our proposed approach with formal proofs. We also present the GWPC algorithm that allocates services on the same VMs and reneges potential failure request while statistically guarantees QoS constraints in terms of deadline miss ratios. We also design a Green Cloud Computing Prototype, GCCP, to validated the GWPC algorithm, and our experimental results demonstrate that our approach can significantly outperform other traditional approaches in terms of guaranteed QoS levels, power consumption, resource demands, as well as electricity costs. In the future, we intend to extend this work to platforms with heterogeneous servers having different power consumption and processing rates characteristics. We also plan to study the effects of interferences among different service types on the performance of cloud data centers modeled by exploiting different queue models with request reneging.

References

[1] J. Spillner and A. Schill, "A versatile and scalable everything-as-a-service registry and discovery." in *CLOSER*, 2013, pp. 175–183.
[2] M. Zivkovic, J. Bosman, J. L. Van den Berg, R. Van der Mei, H. Meeuwissen, and R. Nunez-Queija, "Dynamic profit optimization of composite web services with slas," in *Global Telecommunications Conference (GLOBECOM), IEEE*, Dec 2011, pp. 1–6.
[3] "Pocket gems on google cloud platform," http://cloud.google.com/customers/pocketgems/.
[4] M. Poess and R. O. Nambiar, "Energy cost, the key challenge of today's data centers: a power consumption analysis of tpc-c results," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1229–1240, 2008.
[5] Greenpeace, "How dirty is your data?" *a look at the energy choices that power cloud computing*, 2011.
[6] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: format+ schema," *Google Inc., Mountain View, CA, USA, Technical Report*, 2011.
[7] R. Sethi, "Exoskeleton: Fast cache-enabled load balancing for key-value stores," 2015.
[8] C. Cai, L. Wang, S. U. Khan, and J. Tao, "Energy-aware high performance computing: A taxonomy study," in *IEEE 17th Int. (ICPADS)*, 2011, pp. 953–958.
[9] I. Hwang, T. Kam, and M. Pedram, "A study of the effectiveness of cpu consolidation in a virtualized multi-core server system," in *Proceedings of the ACM/IEEE int. symposium on Low power electronics and design*, 2012, pp. 339–344.
[10] A. Verma, G. Dasgupta, T. K. Nayak, P. De, and R. Kothari, "Server workload analysis for power minimization using consolidation," in *Proceedings of the Conference on USENIX Annual Technical Conference*, ser. USENIX'09, Berkeley, CA, USA, 2009, pp. 28–28.
[11] K. H. Kim, R. Buyya, and J. Kim, "Power aware scheduling of bag-of-tasks applications with deadline constraints on dvs-enabled clusters." in *CCGRID*, vol. 7, 2007, pp. 541–548.
[12] G. von Laszewski, L. Wang, A. J. Younge, and X. He, "Power-aware scheduling of virtual machines in dvfs-enabled clusters." in *CLUSTER*. IEEE, 2009, pp. 1–10.
[13] P. Lama and X. Zhou, "Efficient server provisioning with end-to-end delay guarantee on multi-tier clusters." in *IWQoS*. IEEE, 2009, pp. 1–9.
[14] M. Poess and R. O. Nambiar, "Energy cost, the key challenge of today's data centers: a power consumption analysis of tpc-c results," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1229–1240, 2008.
[15] A. Beloglazov and R. Buyya, "Energy efficient resource management in virtualized cloud data centers." in *CCGRID*. IEEE, 2010, pp. 826–831.
[16] S. Liu, S. Homsi, M. Fan, S. Ren, G. Quan, and S. Ren, "Scheduling time-sensitive multi-tier services with probabilistic performance guarantee," in *2014 20th IEEE (ICPADS)*. IEEE, 2014, pp. 736–743.
[17] I. Ahmad and S. Ranka, *Handbook of Energy-Aware and Green Computing-Two Volume Set*. CRC Press, 2016.
[18] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: research problems in data center networks," *ACM SIGCOMM computer communication review*, vol. 39, no. 1, pp. 68–73, 2008.
[19] A. Hammadi and L. Mhamdi, "A survey on architectures and energy efficiency in data center networks," *Computer Communications*, vol. 40, pp. 1–21, 2014.
[20] Z. Xiao, W. Song, and Q. Chen, "Dynamic resource allocation using virtual machines for cloud computing environment," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 24, no. 6, pp. 1107–1117, 2013.
[21] C. Mastroianni, M. Meo, and G. Papuzzo, "Probabilistic consolidation of virtual machines in self-organizing cloud data centers," *IEEE T. Cloud Computing*, vol. 1, no. 2, pp. 215–228, 2013.
[22] F. Salfner, P. Tröger, and A. Polze, "Downtime analysis of virtual machine live migration," in *DEPEND*, 2011, pp. 100–105.

[23] R. Bane, B. Annappa, and K. Shet, "Survey of dynamic resource management approaches in virtualized data centers," in *Computational Intelligence and Computing Research (ICCIC), IEEE Int. Conference on*, 2013, pp. 1–7.

[24] S. Wang, W. Munawar, J. Liu, J.-J. Chen, and X. Liu, "Power-saving design for server farms with response time percentile guarantees." in *IEEE Real-Time and Embedded Technology and Applications Symposium*, M. D. Natale, Ed., 2012, pp. 273–284.

[25] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle, "Managing energy and server resources in hosting centers," *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5, pp. 103–116, 2001.

[26] GIGASPACES, "Amazon found every 100ms of latency cost them 1% in sales," http://blog.gigaspaces.com/, 2015.

[27] F. Caglar and A. Gokhale, "ioverbook: Intelligent resource-overbooking to support soft real-time applications in the cloud," in *CLOUD, IEEE 7th Int. Conference on*, 2014, pp. 538–545.

[28] K. C. Almeroth, A. Dan, D. Sitaram, and W. B. Tetzlaff, "Long term resource allocation in video delivery systems," in *INFOCOM'97. 16th Annual Joint Conference of the IEEE Computer and Communications Societies. Driving the Information Revolution., Proceedings IEEE*, vol. 3, 1997, pp. 1333–1340.

[29] F. Caglar, S. Shekhar, and A. Gokhale, "A performance interference-aware virtual machine placement strategy for supporting soft realtime applications in the cloud," *Institute for Software Integrated Systems, Vanderbilt University, TN, USA, Tech. Rep. ISIS-2013-105*.

[30] A. Verma, P. Ahuja, and A. Neogi, "pmapper: power and migration cost aware application placement in virtualized systems," in *Middleware*. Springer, 2008, pp. 243–264.

[31] Y. C. Lee, C. Wang, A. Y. Zomaya, and B. B. Zhou, "Profit-driven service request scheduling in clouds," in *Proceedings of the 2010 10th IEEE/ACM international conference on cluster, cloud and grid computing*. IEEE Computer Society, 2010, pp. 15–24.

[32] "Xen project," http://www.xenproject.org/.

[33] T. Robertazzi, *Computer Networks and Systems: Queueing Theory and Performance Evaluation*, ser. Telecommunication networks and computer systems,2000. Springer, 2000.

[34] V. Gupta, M. Harchol-Balter, J. Dai, and B. Zwart, "On the inapproximability of m/g/k: why two moments of job size distribution are not enough," *Queueing Systems*, vol. 64, no. 1, pp. 5–48, 2010.

[35] V. Petrucci, M. A. Laurenzano, J. Doherty, Y. Zhang, D. Mosse, J. Mars, and L. Tang, "Octopus-man: Qos-driven task management for heterogeneous multicores in warehouse-scale computers," in *HPCA, IEEE 21st Int. Symposium on*, 2015, pp. 246–258.

[36] "Cloudsuite benchmark," http://parsa.epfl.ch/cloudsuite/overview.html.

[37] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *ACM SIGPLAN Notices*, vol. 47, no. 4, 2012, pp. 37–48.

[38] C. Delimitrou and C. Kozyrakis, "Quasar: resource-efficient and qos-aware cluster management," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 127–144, 2014.

[39] D. Y. Barrer, "Queuing with impatient customers and ordered service," *Operations Research,*, vol. 5, no. 5, pp. pp. 650–656, 1957.

[40] M. Dodani, "Cloud architecture." *Journal of Object Technology*, vol. 8, no. 7, pp. 35–44, 2009.

[41] C. Math, "The apache commons mathematics library," http//commons. apache. org/proper/commons-math/, 2014.

[42] P. Wendykier, "Jtransforms," https://sites.google.com/site/piotrwendykier/software/jtransforms, 2009.

[43] E. Feller, L. Rilling, and C. Morin, "Energy-aware ant colony based workload placement in clouds," in *Grid Computing (GRID), 2011 12th IEEE/ACM Int. Conference on*, Sept 2011, pp. 26–33.

[44] Y. Ajiro and A. Tanaka, "Improving packing algorithms for server consolidation." in *Int. CMG Conference*. Computer Measurement Group, 2007, pp. 399–406.

[45] J. Li, Q. Wang, D. Jayasinghe, J. Park, T. Zhu, and C. Pu, "Performance overhead among three hypervisors: An experimental study using hadoop benchmarks," in *BigData Congress, IEEE Int. Congress on*, 2013, pp. 9–16.

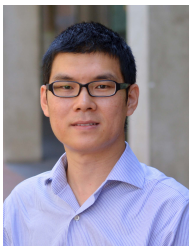
[46] R. G. Michael and S. J. David, "Computers and intractability: a guide to the theory of np-completeness," *WH Free. Co., San Fr*, 1979.

Soamar Homsi (M' 14) received his M.S. in 2014 from the Electrical and Computer Engineering Department at Florida International University, Miami, and is currently a Ph.D. candidate in the same department. His current research interests center on energy-aware cloud computing and data centers, guaranteed QoS scheduling of cloud/web services, workload modeling, and data mining. He is also interested in the thermal/power-aware real-time computing and system design. He has been a research assistant in the Advanced Real-time and Computing Systems (ARCS) laboratory in Miami since 2014.

Shuo Liu (M'09) received his B.S. in electrical engineering from Beihang University, Beijing, China, and his M.S. in electrical engineering from Utah State University, Logan, UT, and his Ph.D in electrical engineering from FIU. He is currently working at Schneider Electric as a software engineer, focusing on real-time security platform.

Gustavo A. Chaparro-Baquero (M'11) received his M.S. degree from the Department of Computer Engineering, University of Puerto Rico, PR, in 2007. He is currently working towards the Ph.D. at the Electrical and Computer Engineering Department, FIU. He worked as Auxiliary Professor with the Electrical and Computer Engineering Department at Universidad del Turabo, Puerto Rico from 2006 to 2011. His research interests include real-time systems and applications, microprocessors, and memories.

Ou Bai received his PhD in 2000 from Saga University in Japan and his BA from Tsinghua University in 1991 from Beijing, China. His current research interests include cyber-physical systems research on robotic/prosthetic optimization. He is also interested in smart and connected health research on in-home exercise/rehabilitation, monitoring cybersecurity, and privacy in cyber-physical systems sensors. He is currently an Associate Professor in FIU.



Shaolei Ren is an Assistant Professor of Electrical and Computer Engineering at University of California, Riverside. Previously, he was an Assistant Professor at FIU from 2012 to 2015. He received his B.E. from Tsinghua University in 2006, M.Phil. from Hong Kong University of Science and Technology in 2008, and Ph.D. from University of California, Los Angeles in 2012, all in electrical and computer engineering. His research interests include cloud computing, data centers, and network economics. He was a recipient of the NSF Faculty Early Career Development (CAREER) Award in 2015.

Gang Quan (M'02-SM'10) received his Ph.D. from the Department of Computer Science & Engineering, University of Notre Dame, USA, his M.S. from the Chinese Academy of Sciences, Beijing, China, and his B.S. from the Department of Electronic Engineering, Tsinghua University, Beijing, China. He is currently an associate professor in the Electrical and Computer Engineering Department, FIU. His research interests and expertise include real-time systems, embedded system design, power-/thermal-aware computing, advanced computer architecture and reconfigurable computing.

APPENDIX A
PROOF OF THEOREM 1

From (6), we have

$$\mu = \frac{ln[\frac{1}{1-R}]}{D} + \lambda \tag{17}$$

If we apply the same processing rate for requests of the same service type with reneging, and let the result completion ratio be R^*, then based on (8) we have

$$\begin{aligned} R^* &= 1 - P_{miss} \\ &= \frac{1 - e^{(\lambda D - \mu D)}}{1 - \frac{\lambda}{\mu} e^{(\lambda D - \mu D)}} \end{aligned} \tag{18}$$

Then with (17), we have

$$\begin{aligned} R^* &= \frac{1 - e^{[\lambda D - (\frac{ln[\frac{1}{1-R}]}{D} + \lambda) D]}}{1 - \frac{\lambda}{\mu} e^{[\lambda D - (\frac{ln[\frac{1}{1-R}]}{D} + \lambda) D]}} \\ &= \frac{R}{1 - \frac{\lambda}{\frac{ln[\frac{1}{1-R}]}{D} + \lambda}(1-R)} \end{aligned} \tag{19}$$

Since $(1 - \frac{\lambda}{\frac{ln[\frac{1}{1-R}]}{D} + \lambda}(1-R)) < 1$, we have:

$$R^* = \frac{R}{1 - \frac{\lambda}{\frac{ln[\frac{1}{1-R}]}{D} + \lambda}(1-R)} > R \tag{20}$$

Equation (20) indicates that the processing rate μ based on the M/M/1 queue can lead to a larger completion ratio R^*, if request reneging is allowed. Therefore, to obtain the same completion ratio (e.g., R), we have $\mu^* \leq \mu$.

APPENDIX B
PROOF OF THEOREM 2

When all classes of requests are hosted together in a single node (e.g., $\hat{VM} = VM_{i,1}$), the processing rate of $VM_{i,1}$, i.e., $U_{i,1}$, has to satisfy the QoS requirements of $\{\tau_{i,1}, ... \tau_{i,r_i}\}$. For example, assume that the required processing rate of $VM_{i,1}$ is $U_{i,1}$, according to $\tau_{i,1}$'s QoS requirements, then based on (9) and 11 we have:

$$\begin{aligned} U_{i,1} &= \mu_{i,1} - \lambda_{i,1} + \sum_{j=1}^{r_i} \lambda_{i,j} && (21) \\ &= \mu_{i,1} + \sum_{j=2, j\neq 1}^{r_i} \lambda_{i,j}. && (22) \end{aligned}$$

We can Equivalently derive the required processing rates for any $VM_{i,l}$, i.e., $U_{i,l}$; where $i \in [1 - r_i]$, according to the QoS requirements of $\tau_{i,l}$ as follows:

$$U_{i,l} = \mu_{i,l} + \sum_{j=1, j\neq l}^{r_i} \lambda_{i,j}, \tag{23}$$

Therefore, when $\{\tau_{i,1}, ..., \tau_{i,r_i}\}$ are hosted together in $\hat{VM} = VM_{i,l}$, in order to satisfy the QoS requirements for all classes of requests simultaneously, the processing rate of $\hat{VM}$, i.e. $\hat{U}$, must satisfy:

$$\hat{U} = \max\{U_{i,1}, ..., U_{i,r_i}\}. \tag{24}$$

APPENDIX C
PROOF OF THEOREM 3

When each class of requests are served separately with a dedicated VM, we have from Definition 1:

$$\Omega(SP_i) = \sum_{j=1}^{r_i} \mu_{i,j} = \sum_{j=1}^{r_i} [\frac{ln[\frac{1}{1-R_{i,j}}]}{D_{i,j}} + \lambda_{i,j}]. \tag{25}$$

When all requests are served by a single VM, we have from Theorem 2:

$$\Omega(\hat{SP_i}) = \max_{j=1}^{r_i} \hat{U}_{i,j} \tag{26}$$

where $\hat{U}_{i,j} = \mu_{i,j} + \sum_{q=1, q\neq j}^{r_i} \lambda_{i,q}$ is the required processing rate of $VM_{i,j}$ to meet $Q_{i,j}$. Because $\mu_{i,j} \geq \lambda_{i,j}$ for all $j \in [1, r_i]$, we have

$$\frac{\Omega(SP_i)}{\Omega(\hat{SP_i})} \geq 1, \tag{27}$$

or, equivalently, $\Omega(SP_i) \geq \Omega(\hat{SP_i})$.

APPENDIX D
PROOF OF THEOREM 4

From Theorem 2, we know that

$$\begin{aligned} u_{1,p} &\geq \max_{\tau_{1,p} \in \Gamma_{1,p}} \{\mu_{1,p} + \sum_{j=p_1, j\neq p}^{p_s} \lambda_{1,j}\} && (28) \\ &= \max_{\tau_{1,p} \in \Gamma_{1,p}} \phi_{1,p} + \sum_{j=p_1}^{p_s} \lambda_{1,j} && (29) \end{aligned}$$

comparatively,

$$\begin{aligned} u_{1,q} &\geq \max_{\tau_{1,q} \in \Gamma_{1,q}} \{\mu_{1,q} + \sum_{j=q_1, j\neq q}^{q_s} \lambda_{1,j}\} && (30) \\ &= \max_{\tau_{1,q} \in \Gamma_{1,q}} \phi_{1,q} + \sum_{j=q_1}^{q_s} \lambda_{1,j} && (31) \end{aligned}$$

Thusly, $\Omega(V) = U_{1,p} + U_{1,q}$ is minimized if $\Phi = \max_{\tau_{1,p} \in \Gamma_{1,p}} \phi_{1,p} + \max_{\tau_{1,q} \in \Gamma_{1,q}} \phi_{1,q}$ is minimized.

APPENDIX E
PROOF OF THEOREM 5

From Theorem 4, to minimize $\Omega(SP_i)$ we only need to minimize $\Phi = \max_{\tau_{i,p} \in \Gamma_{i,p}} \phi_{i,p} + \max_{\tau_{i,q} \in \Gamma_{i,q}} \phi_{i,q}$. Without loss of generality, assume that $\tau_{i,\alpha} \in \Gamma_i$ is the one with the largest value of ϕ, and is allocated to $VM_{i,p}$. Then to optimize $\Omega(SP_i)$ we only need to optimize $\max_{\tau_{i,q} \in \Gamma_{i,q}} \phi_{i,q}$. Note that any $\tau_{i,j} \in \Gamma_{i,k}^h$ allocated to $VM_{i,q}$ will lead to a larger $\max_{\tau_{i,q} \in \Gamma_{i,q}} \phi_{i,q}$ than it is when all $\Gamma_{i,k}^h$ are allocated to $VM_{i,p}$. But, $\Gamma_{i,k}^h + \{\tau_{i,k}\}$ cannot be feasibly allocated to the same node $VM_{i,p}$ simultaneously, and thus $\tau_{i,k}$ has to be allocated to $VM_{i,q}$. For the rest of $\tau_{i,j}$, their allocations do not affect the optimality of $\Omega(SP_i)$ as shown in Theorem 4.

APPENDIX F
PROOF OF THEOREM 6

To prove our claim, we reduce the traditional bin-packing problem [46] to our service packing problem. Consider a bin-packing problem with r_i objects, with the size of object (i, j) being $\lambda_{i,j}$, to be placed into bins with capacity of C.

The goal is to minimize the number of bins. Accordingly, we can construct our service packing problem as follows: Given a set of VMs, i.e., $SP_i = \{VM_{i,1}, \ldots, VM_{i,m_i}\}$, each with size $C_{i,j} = C + U$ (where $U > max\{\lambda_{i,j} : j = 1, ..., r_i\}$ is a constant), and a list of r_i requests with arrival rates of $\{\lambda_{i,1}, ..., \lambda_{i,r_i}\}$, let the required processing rate of each request in Γ_i be $\{\mu_{i,1} = \lambda_{i,1} + U, ..., \mu_{i,r_i} = \lambda_{i,r_i} + U\}$. The goal is to select m VMs and allocate requests to them such that the processing rate for the server pool Ω_i is minimized, i.e.:

$$\{\Omega_i = \sum_{j=1}^{m_i} U = m \times U + \sum_{j=1}^{r_i} \lambda_{i,j}\} \tag{32}$$

Since Ω_i is minimized if m is minimized, if we can find the minimum required processing rate Ω_i in a deterministic polynomial time, we can also find the minimum number of bins in the original bin-packing problem in a deterministic polynomial time. Seeing that the original bin packing problem is NP-hard, minimizing Ω_i is also NP-hard.