Precise Power Capping for Latency-Sensitive Applications in Datacenter

Song Wu, *Member, IEEE,* Yang Chen, Xinhou Wang, Hai Jin, *Senior Member, IEEE,* Fangming Liu, *Member, IEEE,* Haibao Chen, Chuxiong Yan

Abstract—Power capping is widely used in cloud datacenters to mitigate power over-provisioning problem, thus improve datacenter capacity and cut off their operation cost. However, inappropriate or aggressive power capping may lead to performance degradation of applications (especially latency-sensitive ones), and there are few effective methods that can accurately evaluate and control such negative impact caused by aggressive power capping. In this paper, we propose *Fine-Grained Differential Method* (FGD) to quantitatively analyze how inappropriate power capping degrades the performance of latency-sensitive applications. By using FGD, we can minimize the provisioned power for each server by setting a precise power budget according to application's *Service Level Agreement* (SLA). And we further propose *Precise Power Capping* (PPCapping) which is designed to increase the datacenter capacity with a fixed power supply by means of FGD. Our research also provides an insight of precise tradeoff between applications' SLAs and datacenter capacity. We verify FGD and PPCapping by using real world traces from Tencent's datecenter with 25328 servers. The experimental results show that FGD can accurately analyze the impact of power capping on the performance of latency-sensitive applications, and PPCapping can effectively increase datacenter capacity compared with the typical power provisioning strategy.

Index Terms—Datacenter capacity, Power provision, Latency-sensitive application, Cloud computing

1 Introduction

Datacenters have become the first option for IT companies, especially leading ones like Google, Amazon, and Tencent [1], to deploy their latency-sensitive Internet services such as web-based applications, online games and streaming media [2]. Power supply is a common restriction in datacenter construction [3]. Typically, datacenters are extremely expensive to build with an average cost of about 10$\sim$20 USD per watt in industry [4]. Given such a high cost, it is important to fully utilize the power capacity of datacenters.

Unfortunately, the available power in cloud datacenters is often not fully utilized [5]. Datacenters typically allocate power budget for a server at its power rating[1]. However, the power used by a server is always far below its power rating [5, 6], which leads to *power over-provisioning*.

Power over-provisioning causes *power margin* which is the *unused* power budget of a server. Power margin directly decreases available power utilization of datacenters. Assume the power demand of a typical server in a period of time is shown in Fig. 1(a), if we set its power budget at the power rating as shown in Fig. 1(b), there exists a huge power margin (shown as the gray area). Due to over-provisioning problem, power margin widely exists in datacenters [5], which is a horrendous waste of power budget. It is no doubt that the elimination of the widespread power margin has a great effect on increasing available power utilization of datacenters.

Power capping is widely used in cloud datacenters to mitigate power over-provisioning problem, thus improve capacity and cut off operation cost of datacenters [3, 5, 7–11]. A cost-effective way to do power capping is restricting power budget of servers. Since the power supply of a datacenter is typically fixed, restricting servers' power budget will allow more servers to be deployed in datacenter. Take Fig. 1 as an example, when we restrict the power budget (shown as red dashed line) below the power rating as shown in Fig. 1(c), there is no performance degradation and a significant part of the power budget is saved compared with Fig. 1(b). The saved power budget of servers can be aggregated and used to provision extra servers to increase datacenter capacity.

However, restricting the power budget may result in performance degradation when the total power demand of a server exceeds its budget. For example, when we further restrict the power budget as shown in Fig. 1(d), power budget violation occurs between t_0 and t_1 (shown as the yellow area), which will degrade applications' performances.

Hence, there exists a tradeoff between the degradation of applications' performance and the power budget saving in datacenters. On the one hand, sometimes little performance degradation may bring much more power budget saving (e.g., the saved power budget in Fig. 1(d) is much more than that in Fig. 1(c)) as long as such degradation is acceptable for applications. On the other hand, datacenters are very

• *S. Wu, Y. Chen, X. Wang, H. Jin and F. Liu are with the Services Computing Technology and System Lab, Big Data Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China.*
E-mails: {wusong, ychen1, xwang, hjin, fmliu }@hust.edu.cn
• *H. Chen is with Chuzhou Colledge.*
E-mail: chb@chzu.edu.cn
• *C. Yan is with Tencent, Inc.*
E-mail: chuxiongyan@tencent.com

1. Similar to [4, 5], power rating refers to as the measured maximum power consumption of a server rather than the nameplate value that is often higher.

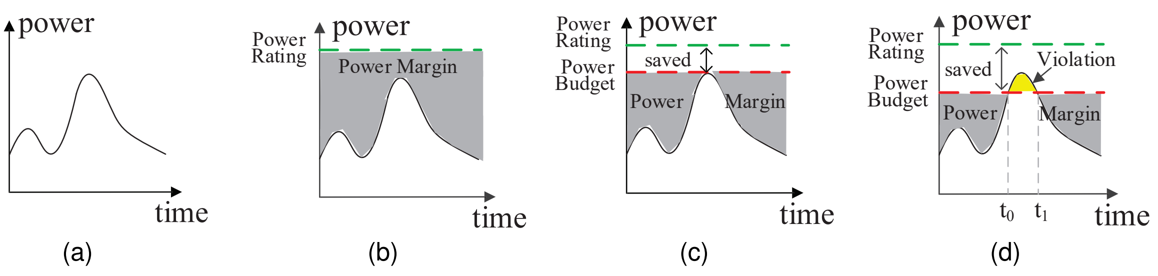


Fig. 1. (a) Power demand curve of a typically server in a period of time. (b) If setting power budget for the server at power rating, there exists a large power margin (shown in gray area). (c) If setting power budget at observed peak power, a notable part of power budget can be saved. (d) Violation occurs when further restricting the power budget.

cautious about the performance degradation of applications, especially latency-sensitive ones, due to the concern about the violation of *Service Level Agreement* (SLA) signed with tenants.

The key challenge of power capping is to negotiate the right tradeoff thus appropriately restrict the power budget to maximize the elimination of power margin while keeping applications' performance degradation in a tolerable range. So, the question arises that: Can we accurately depict the effect of power capping on such performance degradation?

Several methods [5, 9, 10, 12] have been proposed to evaluate the performance degradation caused by power budget violation. However, they are mainly designed for batch applications and are defective for latency-sensitive applications. In this paper, we focus on latency-sensitive applications, and propose *Fine-Grained Differential Method* (FGD) to accurately analyze the performance degradation of latency-sensitive applications caused by power budget violation. By using FGD, we can minimize the provisioned power for each server by setting a precise power budget according to application's SLA. And we further propose *Precise Power Capping* (PPCapping) which is designed to increase the datacenter capacity with a fixed power supply by means of FGD. The main contributions of our work are as follows:

- We propose an accurate and fine-grained analysis approach based on calculus, FGD, which can provide an intuitive perspective about how server power budget impacts the performance of a latency-sensitive application.
- We propose a power capping algorithm based on FGD, PPCapping, which assigns a precise power budget for each server and provides an insight of precise tradeoff between applications' SLAs and datacenter capacity.
- We verify the accuracy of FGD by using real world web server traces, and evaluate PPCapping with real world traces consisting of 25000+ servers from Tencent's production datacenter. The experimental results show that FGD can accurately analyze the impact of power capping on the performance of latency-sensitive applications and PPCapping can effectively increase datacenter capacity.

The rest of the paper is organized as follows. We elaborate the background and our motivation for this research in Section 2 and give details about *Fine-Grained Differential Method* in Section 3. Section 4 proposes our *Precise Power Capping* and its usage in datacenter. We validate the accuracy of FGD and evaluate effectiveness of PPCapping in Section 5 while discuss some limitations about our work in Section 6. We review the related work in Section 7 and finally conclude the paper in Section 8.

2 Background and Motivation

In this section, we first introduce the background of power capping technology and introduce the characteristics of latency-sensitive applications on which we focus in this paper. After that, we point out the weaknesses of state-of-the-art methods of evaluating performance degradation used in power capping, which motivates our approach.

2.1 Power Capping and Latency-Sensitive Applications

Typically, a cloud infrastructure provisions power in four levels, *datacenter*, *power distribution unit (PDU)*, *rack*, and *server*. The datacenter has a fixed power supply which is normally partitioned into several PDUs, and then into tens of racks. Each rack has a power budget. Servers are deployed in racks and are provisioned commonly according to their power ratings, which leads to serious power margin. Assuming a rack's power rating is 6KW and a server's power rating is 300W, one can only launch 20 servers in this rack, which is conservative.

Power Capping. Deploying an aggressive number of servers into a rack will incur a risk of exceeding power rating of the rack, or power oversubscription [13]. Power oversubscription may break the circuit and power off all servers, which is a catastrophic event for users. Power capping technology is used to limit facilities' power budgets to protect them from overloading. Power capping usually triggers *Dynamic Voltage and Frequency Scaling* (DVFS) when the power of a server is beyond its power budget. Note that, previous researches [5, 14, 15] recognize the strong correlation between a server's CPU utilization and its full-system (including CPU, memory, disk, etc.) power consumption[2]. The finding suggests resizing the CPU utilization has almost the same effect as capping the power budget. Therefore, in this paper, we resize a server's CPU utilization threshold instead of power budget to simplify the analysis of power

2. The reason for such strong correlation is that other components have very small dynamic power range or their activity levels correlate well with CPU activity at the full-system level.

capping impact. Also, for simplicity, we throttle CPU utilization[3] so as to equivalently throttle the power consumption of a server.

Latency-Sensitive Applications. In this paper, we focus on latency-sensitive applications such as Internet services which are widely deployed in cloud datacenters. Latency-sensitive applications usually contain a large number of request-response operations (e.g., a request sent to the server and its response back to the client), and are sensitive to the response time (i.e., latency) of their requests. Typically, each latency-sensitive application has a strict SLA especially on *tail latency* (i.e., the distribution of latency) [13]. Therefore, the performance of latency-sensitive applications can be easily affected by power capping. For example, a HTTP web server's response time will increase seriously if power budget violation occurs [8, 16]. Given that *latency* is the major concern of latency-sensitive applications, in this paper, if not otherwise specified, SLA refers to the latency SLA. Additionally, in this paper, we assume that a server runs one application at a time, which is a common practice for IT companies (e.g., Tencent Inc.) to guarantee their latency-sensitive applications' performance.

2.2 Existing Methods for Evaluating Performance Degradation

Current power capping tends to be conservative or aggressive when setting the power budget of servers [5, 17]. The conservative power capping always sets the power budget according to the power rating or *observed peak power*. However, the majority of servers cannot reach the power budget for most of the time. Though the conservative power capping can guarantee applications' SLAs, it cannot achieve potential maximum power utilization. On the contrary, the aggressive one always sets a low power budget to substantially eliminate the power margin thus improving the power utilization, which, however, frequently incurs SLA violation.

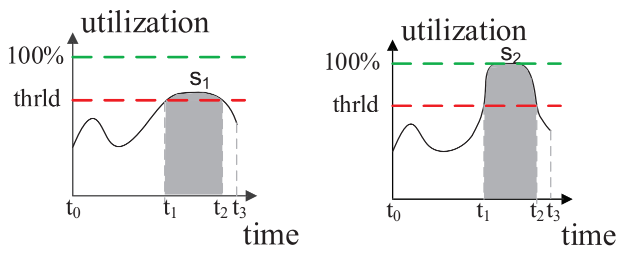
In summary, current power capping cannot set a precise power budget to maximize power utilization while meeting SLAs of applications. The crux of the problem is how to accurately evaluate the performance degradation of latency-sensitive applications caused by power budget violation.

Several existing methods have been proposed to evaluate the performance degradation caused by power budget violation [5, 9, 10, 12]. However, we find that they have obvious limitations when evaluating the performance degradation of latency-sensitive applications. In the following, we will analyze the problems of two typical state-of-the-art methods, which motivate us to devise a new one.

One method proposed by [5, 10] takes an evaluation approach that measures the performance degradation by the time percentage of the power budget violation, which is denoted by *percentage of budget violation* (PBV). That is, the longer the violation of the power budget, the severer the performance degradation.

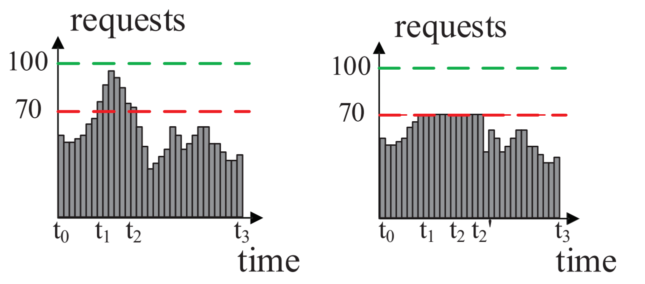
Fig. 2 shows two cases of power budget violation in a scenario where the two curves represent the CPU utilization demands of two servers. The shaded areas represent the

3. We can easily throttle CPU utilization with Linux command *cpulimit* or *cgroups*.



(a) Case of slight violation (b) Case of serious violation

Fig. 2. The performance degradation given by PBV in the above cases are the same, but the actual performance degradation degrees are different.



(a) The incoming requests (b) The processed requests

Fig. 3. In this case, the performance degradation indicated by PPL is 0, but it exists in fact.

parts of jobs suffering from power budget violation, respectively. As shown in Fig. 2(a) and 2(b), the power budget is violated during the time period from t_1 to t_2, which lies in the total time period from t_0 to t_3. In these two cases, the performance degradation values given by PBV are both $(t_2 - t_1)/(t_3 - t_0)$. However, the CPU utilization demand in Fig. 2(b) is much steeper than that in Fig. 2(a) during the violation time period, and its shaded area shown in s_2 is much larger than s_1 in Fig. 2(a). As we mentioned, if lack of CPU resource (i.e., power budget violation occurs), the performance of latency-sensitive applications will be greatly affected. Intuitively, for a latency-sensitive application, the performance degradation in Fig. 2(b) is much severer, although its performance degradation value given by PBV is equal to that in Fig. 2(a). In summary, PBV cannot describe the performance degradation of latency-sensitive applications accurately, because it cannot reflect the degree of violation during the violation time period.

The other method proposed by [9, 12, 18] uses the percentage of work done during an observed time period to measure the performance loss of applications, which is denoted by *percentage of performance loss* (PPL). This method is useful for batch jobs that mainly concern how long their completion time has been delayed. Note that, some methods used in other studies are the variants of PPL, such as [19]. These metrics are all time-related, and they the same in essence.

PPL is too coarse-grained to evaluate the performance degradation of latency-sensitive applications which usually contain a great deal of request-response operations. To illustrate the inapplicability of PPL when measuring performance degradation of latency-sensitive applications, we take a web application scenario as an example. Assuming the application can process 100 requests per second at its best effort and 70 requests per second after restricting a fixed power budget of the server. As shown in Fig. 3, assuming a client sends requests at the rate in Fig. 3(a) and the number of total requests between t_0 and t_3 is 2000. The request-processing rate under the fixed power budget is shown in Fig. 3(b), where the number of requests processed between t_0 and t_3 is also 2000. That is, no requests are lost. Therefore, the performance degradation given by PPL is 0. However, performance degradation exists when power budget violation occurs, because some requests arriving between t_1 and t_2 cannot be processed immediately. This scenario shows that PPL is not suitable for evaluating the performance degradation of latency-sensitive applications because it fails to capture the performance loss of those tiny parts of these applications (e.g., the delay of a request-response operation of web applications).

To address the limitations of existing methods, we devise a fine-grained analysis method, *Fine-Grained Differential Method*, to evaluate the performance degradation of latency-sensitive applications.

3 Fine-Grained Differential Method

In this section, we first introduce the basic idea of our method. Then we discuss *Fine-Grained Differential Method* (FGD) in detail, based on the concept of *differential Workload*, to effectively evaluate the performance degradation of latency-sensitive applications caused by power budget violation.

3.1 Basic Idea

As analyzed above, both PBV and PPL have their limitations in evaluating performance degradation of latency-sensitive applications. In this subsection, we introduce the basic idea of our method, which can address problems of the state-of-the-art methods.

In Fig. 2, we use the shaded area to represent the degree of power budget violation. And actually it can also reflect the degree of applications' performance degradation. Typically, the larger the shaded area, the severer the power budget violation. For example, the shaded area in Fig. 2(b) is larger than that in Fig. 2(a), which implies that more serious performance degradation exists in the case of Fig. 2(b). In order to measure the shaded area, we introduce the concept of CPU Workload (*Workload* for short), which is the integral of CPU utilization (denoted by $u(t)$) during a period of time (e.g., from t_0 to t_1), as shown in Equation 1.

$$Workload = \int_{t_0}^{t_1} u(t)dt \tag{1}$$

The meaning of *Workload* is the CPU cycles needed for an application during the time period from t_0 to t_1. Since each request-response operation of the latency-sensitive application consumes a certain number of CPU cycles, *Workload* can refer to a set of request-response operations. In short, *Workload* can be understood as a set of requests. By using *Workload*, the problem of PBV presented in Fig. 2 can be solved.

As for PPL, it is too coarse-grained to evaluate latency-sensitive applications (it neglects the violation occurring from t_1 to t_2 shown in Fig. 3(a)). In order to solve this problem, we introduce another concept, *differential Workload*, which is *Workload* in a very narrow time period. By calculating the performance degradation of each *differential Workload*, the subsistent performance degradation from t_1 to t_2 in Fig. 3 can be easily captured, which is crucial for evaluating performance degradation of latency-sensitive applications. Hence, the problem of PPL presented in Fig. 3 can be solved.

3.2 Detail of Our Method

In this subsection, we first introduce the design of our method, and then present how to implement it.

3.2.1 Design

To clearly describe our design, first we introduce the metric of *tail latency* considering in FGD, then we present *P-Table* as the goal of our method, and finally we show how to achieve the goal.

tail latency. As we mentioned above, *tail latency* is one of the most concern of latency-sensitive applications. Obviously, the shorter application's latency, the better its performance. Thus we adopt *tail latency* as the performance metric of our approach. For simplicity, a two-tuple (*percentage of requests*, *latency time*) is used to represent the tail latency to describe the performance of a latency-sensitive application. For example, (95%, 200ms) means that the performance is "95% of requests' latencies are below 200ms". Setting power budget for a server may cause longer latency of requests. Performance degradation can be clearly reflected in requests' tail latency. For example, the performance may drop from (95%, 200ms) to (95%, 400ms) after setting a lower power budget.

Clearly, there is a negative effect on the performance after setting a lower power budget (i.e., CPU *Threshold*). Now we take CPU *Threshold* into account and extend the two-tuple (*percentage of requests*, *latency time*) to a three-tuple (*percentage of requests*, *latency time*, *Threshold*). For example, (95%, 200ms, 60%) means 95% of requests' latencies are below 200ms with 60% CPU utilization *Threshold*, and the performance may drop from (95%, 200ms, 60%) to (95%, 400ms, 50%) after restricting CPU *Threshold* from 60% to a lower value 50%.

A request takes a certain period of time to be processed, and the total latency of a request is the sum of basic processing time (baseline latency) and the waiting time (additional latency) caused by power budget violation. Since the baseline latency can be considered as a fixed value for the same types of requests, adding it to the additional latency would not affect the distribution of tail latency. Thus, for simplicity, we assume that the baseline latency is constant, and only calculate additional latency to highlight the power capping impact on SLA.

P-Table. The goal of our method is to quantify performance with the fixed CPU *Threshold*. To show the quantified results in an intuitive way, we put three-tuple performance results in a *performance table* (*P-Table* for short). Table 1 is an example of *P-Table*. By looking up *P-Table*, we can easily obtain the performance with any fixed CPU threshold. Obviously, we can also obtain the performance degradation by comparing corresponding three-tuples in *P-Table*. A natural question arises: how to obtain *P-Table* of a latency-sensitive application?

TABLE 1
The example of *P-Table* for an application, which records the latency time (ms) with all possible request percentiles and CPU thresholds.

thrld \ ptile	...	**95%**	...	99%	100%
...	...	...	...	...	...
31%	...	300	...	800	900
32%	...	**100**	...	750	800
...	...	...	...	...	...
99%	...	50	...	100	200
100%	...	30	...	80	150

The crux of obtaining *P-Table* is to obtain requests' latency with a fixed CPU threshold. For latency-sensitive applications, as mentioned in Section 3.1, *differential Workload* can be equivalent to tiny requests. Thus by evaluating the latency of each *differential Workload* for a certain application, we can equivalently obtain the corresponding latency of any request. Hence, the following, we explore how to get the latency of every *differential Workload* of an application.

Latency of *differential Workload*. To get the latency of each *differential Workload*, first we analyze how power budget violation impacts *Workload* in a time period as shown in Fig. 4. Specifically, Fig. 4(a) presents the CPU demand curve over time (denoted by $f(t)$) of a server. Recall that capping the CPU utilization has almost the same effect as resizing the power budget, if setting a CPU utilization threshold at the value of $thrld$, the power budget violation will occur during the time from t_0 to t_1 and from t_3 to t_4, and its actual CPU utilization will be the curve (denoted by $g(t)$) shown in Fig. 4(b). *Differential Workload* (e.g., the one shown as the dark gray bar at t_x in Fig. 4(a)) cannot be served immediately, and it will be accumulated and processed until there is abundant CPU resource (e.g., the *differential Workload* at t_x in Fig. 4(a) is finally served at t_y in Fig. 4(b)).

To evaluate the impact of power budget violation on performance of latency-sensitive applications, we introduce two key functions as follows.

- The delay of *differential Workload* at time t, denoted as $Delay(W_t)$, which is caused by the power budget violation. For example, $Delay(W_{t_x}) = t_y - t_x$, and $Delay(W_{t_z}) = 0$ in the scenario of Fig. 4. $Delay(W_t)$ is exactly what we need to solve.
- The amount of accumulated *Workload* at time t, denoted as $h(t)$, which reflects the degree of CPU resource shortage.

As shown in Fig. 4(c), from the beginning of t_0 (when power budget violation occurs), more and more *Workload* is accumulated, so $h(t)$ keeps growing. From t_1, the CPU utilization demand shown in Fig. 4(a) falls below $thrld$, which means CPU resource is abundant. Thereby $h(t)$ starts to fall from t_1 and becomes 0 at t'_1. Hence, $h(t)$ can reflect the real-time relationship between the CPU utilization demand $f(t)$ and actual CPU utilization $g(t)$. That is, when $h(t) > 0$, CPU resource is in shortage, leading $g(t)$ to $thrld$; when $h(t) = 0$, CPU resource is abundant and $g(t)$ equals to $f(t)$.

As shown in Fig. 4(c), $h(t)$ is always greater than or equal to 0, and it is greater than 0 when power budget violation happens periodically. Apparently, $h(t)$ equals to 0 before t_0, the time point when power budget violation occurs, and starts to grow up from t_0. The growing speed of $h(t)$ is determined by the difference between the server's CPU demand $f(t)$ and the $thrld$. Thus in a power budget violation period, $h(t)$ can be expressed as Equation 2. Once a power budget violation period begins, it will last until $h(t)$ decreases to 0 again (e.g., one period starts from t_0 and ends at t'_1 in Fig. 4(c)). The end time point can be obtained by Equation 3 with the meaning of $h(t) = 0$.

$$h(t) = max\left\{\int_{t_0}^{t}(f(t) - thrld)dt, 0\right\} \tag{2}$$

$$\int_{t_0}^{t}(f(t) - thrld)dt = 0 \tag{3}$$

$h(t)$ is a repetitive segmented function (see Fig. 4(c)) since the power budget violation occurs periodically. Thus, by repeating above steps, we can get the entire $h(t)$.

As discussed above, when $h(t) = 0$, $g(t) = f(t)$. In other cases, $g(t) = thrld$. Thereby, we can obtain $g(t)$ as Equation 4.

$$g(t) = \begin{cases} f(t) & h(t) = 0 \\ thrld & else \end{cases} \tag{4}$$

Assuming *Workloads* are served in the *First In First Out* (FIFO) principle, the arriving *Workload* have to wait until the accumulated *Workload* has been served. According to the principle of conservation of *Workload*, the latency of the *differential Workload* at time point t ($Delay(W_t)$) must comply with Equation 5.

$$\int_0^t f(t)dt = \int_0^{t+Delay(W_t)} g(t)dt \tag{5}$$

Assuming the *Workload* at t_x in Fig. 4(a) are finally served at t_y in Fig. 4(b), when we calculate $Delay(W_{t_x})$, the intuition of Equation 5 is that the light gray area between 0 and t_x in Fig. 4(a) should be equal to the one between 0 and t_y in Fig. 4(b).

By solving Equation 5, we can obtain the $Delay(W_t)$ of any *differential Workload* as Equation 6.

$$Delay(W_t) = t_d\Big|_{\int_0^t f(t)dt = \int_0^{t+t_d} g(t)dt} \tag{6}$$

Once getting $Delay(W_t)$ of each *differential Workload*, we can derive the tail latency by sorting latency of all requests, which forms a row of *P-Table*. Finally, we can get entire *P-Table* by looping the above process with specific CPU threshold fixed (1%-100%).

So far, we have elaborated on how to calculate $Delay(W_t)$ of any *differential Workload* thus obtaining *P-Table*. The CPU utilization over time, i.e. $f(t)$, is the precondition of our method. In real-world datacenters, monitoring systems can record each server's CPU utilization and other status data by the agents deployed in servers, and store

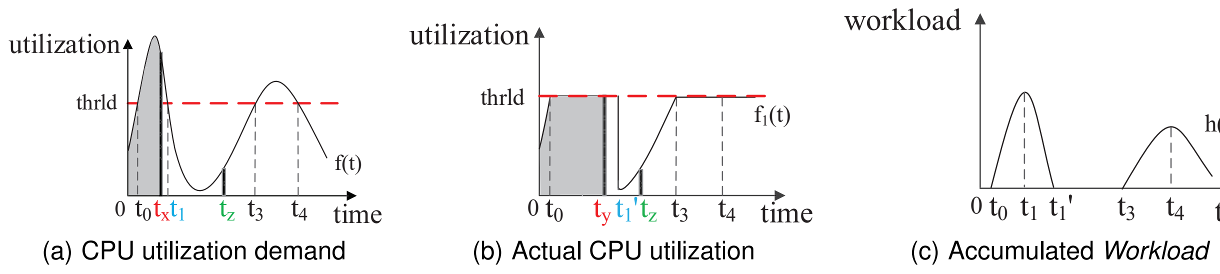


Fig. 4. This scenario shows a server's variation of CPU utilization and its accumulated *Workload* caused by power budget violation. When the server's CPU utilization demand exceeds the $thrld$, its actual CPU utilization will be capped and accumulated *Workload* (shown as function h(t)) will increase.

these traces in database. The CPU utilization over time in these traces (see Table 2(a)) can be used as the discrete $f(t)$, which enable the precondition of our method when calculating $Delay(W_t)$.

3.2.2 Implementation

In this part, we show how to obtain *P-Tables* of latency-sensitive applications so as to accurately describe their performance degradation at different power budgets.

P-Table obtaining process is illustrated in Fig. 5. The process contains two steps. First, we calculate the latency of *Workload*, i.e. $Delay(W_t)$, through the trace, and put the latency into $RequestsLatency$ table which is used as a transitional table to obtain *P-Table*. Table 2 shows examples of the trace and the $RequestsLatency$ table. Second, we yield *P-Table* by sorting all obtained $RequestsLatency$ tables with different CPU thresholds. The details are as follows.

Step 1. As shown in Fig. 5, to simulate the accumulating process of *Workload*, we introduce a FIFO accumulated *Workload* queue (denoted by ARQ). Given a fixed CPU threshold $thrld$, we take samples of CPU utilization (denote the i-th sample by u_i) from the trace and transform each sample to discrete *differential Workload* as $u_i \times \Delta t$, where Δt is the sample interval of CPU utilization. We enqueue each *differential Workload* into ARQ and meanwhile dequeue *Workload* at the best of the server's processing capability $thrld \times \Delta t$. We record the latency of *differential Workload* ($Delay(W_t)$) as the difference between its dequeue time point and enqueue time point.

As we mentioned before, the *differential Workload* can be equivalent to a small number of requests. Thus the latency of *differential Workload* can be regarded as the latency of these requests. Recall that we assume the baseline latency is constant and only record additional latency in the $RequestsLatency$ table to highlight the power capping impact on SLA. For example, assuming a *differential Workload's* enqueue time point is t_i as shown in Fig. 5, and its dequeue time point is t_j due to the queuing effect in ARQ, we record $t_j - t_i$ as the latency of these $u_i \times \Delta t$ requests and put it into $RequestsLatency$ table (see Table 2(b)).

Step 2. After step 1, we can get all $RequestsLatency$ tables by setting $thrld$ at all possible CPU utilization levels (from 1% to 100%). We sort each $RequestsLatency$ table by latency to obtain tail latency. For example, Table 2(b) is sorted, and the total number of requests is 24690, thus the 95-th and the 99-th tail latency of requests is 300ms and 500ms, respectively. We can calculate all percentiles of tail latency, which forms a row of *P-Table* (see Table 1). The pseudo code of *P-Table* obtaining process is shown in Algorithm 1.

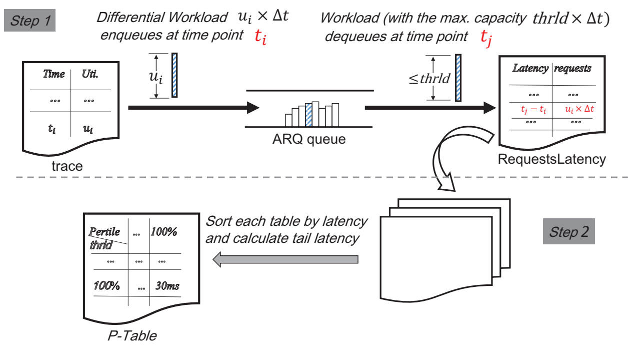

Step 1
Differential Workload $u_i \times \Delta t$ enqueues at time point t_i
Workload (with the max. capacity $thrld \times \Delta t$) dequeues at time point t_j
Time
Uti.
t_i
u_i
trace
u_i
ARQ queue
$\leq thrld$
Latency
requests
$t_j - t_i$
$u_i \times \Delta t$
RequestsLatency
Sort each table by latency and calculate tail latency
Step 2
Pertile
thrld
100%
30ms
P-Table

Fig. 5. *P-Table* obtaining process with two steps. 1) Each sample of CPU utilization (denote the i-th sample by u_i) in the trace is transformed to discrete *differential Workload* as $u_i \times \Delta t$ requests, where Δt is the CPU utilization sampling interval. We simulate the process in ARQ and record requests' latency in $RequestsLatency$ table. 2) Sorting each $RequestsLatency$ table by latency to obtain tail latency, which forms a row in *P-Table*.

TABLE 2
Examples of the trace and the *RequestsLatency* table. The trace records sampling time points and the corresponding CPU utilization. *RequestsLatency* table records the latency time and the corresponding *Workload* (i.e., the number of requests) in order to obtain tail latency.

(a) trace

time	CPU Uti.
12:00:00	25%
12:05:00	30%
12:10:00	58%
...	...
t_n	u_n

(b) RequestsLatency

latency	# of requests
0ms	15000
50ms	7800
300ms	1300
500ms	570
1000ms	20

Cost analysis: In our algorithm, each round of computing iterates all n samples, and it loops 100 rounds (constant), thus its time complexity is $O(n)$. For each server, the algorithm uses a queue ARQ and a table $RequestsLatency$ to record the latency of all *differential Workload*, hence its space complexity is $O(n)$. Sorting $RequestsLatency$ by latency costs $O(n)$ with the help of the extra space of table $RequestsLatency$. Therefore, the total time complexity of solving *P-Table* of a server is $O(n)$. Note that, the algorithm is performed offline, and it has only $O(1)$ cost for the

Algorithm 1 *P-Table* obtaining algorithm

```
Input: CPU utilization trace of a server with $n$ samples, and
    denote the i-th sample by $u_i$.
Output: P-Table of the server.
1:  for $thrld \leftarrow 1$ to 100 do
2:      Initialize $ARQ$ and $RequestsLatency$ table which is
          used to record the number of requests with different
          latencies.
3:      $w$ = 0 /* Accumulated Workload, h(t) in Fig. 4(c) */
4:      $time$ = 0 /* timestamps */
5:      for $u_i \in trace$ do
6:          Insert the tuple comprising the current number
            of Workload $u_i * \Delta t$ and the current time point $time$
            into $ARQ$.
7:          $w = w + u_i * \Delta t$
8:          /* $thrld * \Delta t$ in next line means the max. number
            of Workload that CPU can serve in a cycle*/
9:          if $w > thrld * \Delta t$ then
10:             Dequeue $thrld * \Delta t$ of Workload from $ARQ$
                  and recording their latencies (the difference
                  between current time point and enqueue time
                  point) into$RequestsLatency$.
11:             $w = w - thrld * \Delta t$
12:         else
13:             Dequeue all Workload from $ARQ$ and recording
                  the number of requests with corresponding
                  latencies into $RequestsLatency$.
14:             $w = 0$
15:         end if
16:         $time = time + 1$
17:     end for
18: end for
19: Sort each $RequestsLatency$ by latency to obtain tail latency
      which can form the P-Table.
```

TABLE 3
The Mapping table that maps server's CPU utilization to its power.

CPU utilization	power
100%	250W
99%	248W
98%	246W
...	...
32%	**180W**
31%	178W
...	...

TABLE 4
The example of the power budget *Suggestion table*

SID	SLA(s)	power budget (CPU thrld)
10001	(95%, 100ms) (95%, 200ms) ...	180W (32%) 175W (30%) ...
10002	(99%, 500ms) ...	208W (50%) ...
10003	(99%, 500ms) ...	195W (35%) ...
...	...	...

Algorithm 2 Precise Power Budget Obtaining Algorithm

```
Input: Traces, the Mapping table and SLA of the application.
Output: Precise power budget for the server.
1: Obtaining P-Table by calling Algorithm 1 which uses traces
     of the server as input.
2: Finding out the precise CPU threshold for the server by
     looking up P-Table according to application's SLA.
3: Returning precise power budget for the server by looking
     up the Mapping table.
```

datacenter to find the performance degradation with a given CPU threshold of a server once *P-Table* is obtained.

4 Precise Power Capping

In this section, we first obtain the precise power budget for each server according to application's SLA by using FGD, then we design *Precise Power Capping* (PPCapping) to maximize the usage of power and increase the capacity of datacenter.

4.1 Precise Power Budgets for Servers

In a large-scale datacenter, every server is typically assigned a unique key denoted by SID. Taking a server whose SID is 10001 as example, if we want to evaluate its performance under different power budgets, we need its CPU utilization trace (as shown in Table 2(a)) and a table mapping its CPU utilization to power (Table 3). Typically, Table 3 is provided by the manufacturer of this server. If this information is not provided, it is easy for administrators to obtain such information by measurements. By inputting Table 2(a), Algorithm 1 can output the *P-Table* shown as Table 1, in which each row presents the tail latency of requests with a fixed CPU utilization threshold.

We denote the SLAs of latency-sensitive applications as the two-tuple expressed tail latency. For example, previous research [20] mentioned that latency-sensitive application-s particularly require 95th-percentile latency of requests. Users will propose SLA with an acceptable performance. Then we can find out the most appropriate CPU threshold for a server by looking up *P-Table*. For instance, if SLA requires (95%, 100ms), then we search Table 1 in the column with percentile '95%' bottom-up to find the lowest CPU utilization threshold meeting this SLA. In this example, the precise CPU utilization threshold is 32%. Then we consult the Mapping table and find out the precise power budget is 180W. We plot the detail algorithm in Algorithm 2.

Further, administrators can apply Algorithm 2 to all servers in their datacenters, and they will finally obtain a power budget *Suggestion table* for all servers as shown in Table 4. Therefore, administrators can allocate precise power budgets to all servers by looking up this table.

4.2 Precise Power Capping Algorithm

So far, we have obtained the *Suggestion table* containing precise power budget for each server by using Algorithm 2. In this subsection, we propose *Precise Power Capping Algorithm* (PPCaping) to increase the capacity thus achieving higher power utilization of large-scale datacenter.

By using PPCapping, we can deploy more servers into the datacenter. Fig. 6 illustrates how PPCapping works. Since there is no power budget record in the *Suggestion table* for each new server running a certain type of application, we set the power budget of an existing server who runs the same application as the new one's initial power budget. After that, we use the *First-Fit Bin-packing* method to deploy this server. That is to say, we search all racks and find the first rack which can accommodate this server, i.e., $\overline{P_{new}}$ +

$\sum_{k=1}^{N} \overline{P_k} \leq \widetilde{P}$, while $\overline{P_i}$ and $\widetilde{P}$ refer to the set power budget for the i-th server and rack's power rating, respectively. Our work can be easily extended by integrating more efficient *Bin-packing* methods [21, 22], but this is beyond the scope of this paper. Finally, in order to accurately reflect the recent characteristics of applications, we periodically update the power budget *Suggestion table* by calling Algorithm 2 and the period depends on the specified application. In particular, we find a week is suitable for most applications in Tencent datacenter. Algorithm 3 is the pseudo code of PPCapping.

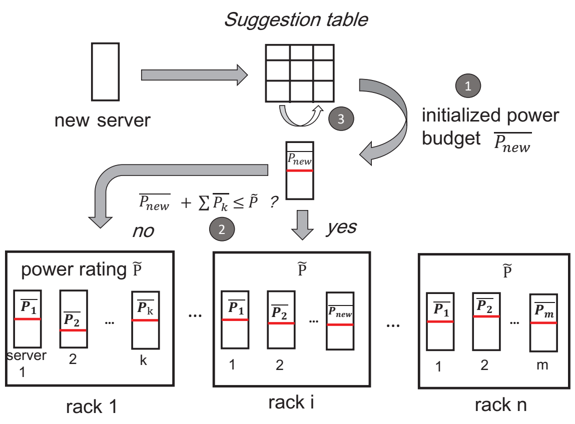


Fig. 6. PPCapping works with three steps. 1) First we set an initial power budget for the new server according to the *Suggestion table* and the type of application it will run. 2) Then we deploy the server into a rack by using the *First-Fit Bin-packing* method. $\overline{P_i}$ and $\widetilde{P}$ refer to the set power budget for the i-th server and rack's power rating, respectively. 3) We periodically update the power budget *Suggestion table* to accurately reflect the recent characteristics of applications.

Algorithm 3 Precise Power Capping Algorithm

```
1: Calling Algorithm 2 and obtaining power budget Suggestion table for all servers in datacenter.
2: Finding a server who is running the same type application as each new server will do, and use its power budget to initialize the new one's (denoted by $\overline{p_{new}}$).
3: for each $rack$ in $all\ racks$ do
4:     /* Current $rack's$ power rating and total power budgets of servers in current $rack$ are denoted by $\widetilde{P}$ and $\sum_{k=1}^{N} \overline{P_k}$, respectively. */
5:     if $\overline{P_{new}} + \sum_{k=1}^{N} \overline{P_k} \leq \widetilde{P}$ then
6:         /* first fit */
7:         Deploying the new server in this $rack$ and break.
8:     end if
9: end for
10: After a period of time (e.g., a week), calling Algorithm 2 to update the power budget Suggestion table.
```

After deploying new servers by using Algorithm 3, we can increase datacenter's capacity without violating racks' power rating while still meeting the SLA of applications.

5 Evaluation

The experiments consist of four parts. In Section 5.1, we verify the accuracy of FGD when evaluating performance degradation of applications. After that, in Section 5.2, we apply PPCapping on production servers to show their performance degradation under different power budgets and to explore how much power budget can be saved in a large number of servers with acceptable performance degradation. Further, we show advantage of PPCapping in performance guaranteeing in Section 5.3. Finally, in Section 5.4, we demonstrate the effectiveness of PPCapping and provide an insight of precise tradeoff between applications' SLAs and datacenter capacity.

5.1 Accuracy of FGD

In this subsection, we verify the accuracy of FGD, PBV and PPL by setting different CPU utilization thresholds for the experimental server and comparing the performance degradation evaluated by these three methods to the performance degradation obtained by actual measurement. We carry out experiments with a real world trace, Mini-Challenge 3 [23].

Mini-Challenge 3 is a dataset including network flow data and network health and status data. We only take the number of requests sent to the server in each second and transform it into a CPU utilization trace as shown in Fig. 7.

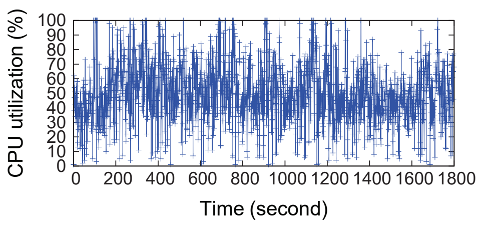


Fig. 7. The CPU utilization of a server in Mini-Challenge 3.

We use two physical 16-core nodes (16 × Intel(R) Xeon(R) CPU E5-2670 0@2.60GHz, 64G RAM) to act as a web server and a client, respectively. The web server hosts an Apache HTTP Server application and constantly serves the HTTP requests from the client. All requests sent to the server access the same PHP page and consume fixed CPU resources. At the client node, we devise a HTTP requests generator which can send a specified amount of requests per second as needed. By controlling the sending rate of requests from the client, we can control the CPU utilization of the server node. Then we record the response time of every request in the log under the fixed CPU utilization threshold of the server. Note that, it takes a certain base time to process a request. We use the difference between response time and the basic processing time as the latency caused by power budget violation, which forms the measured latency results.

The measured results and the FGD's evaluated results are shown in Fig. 8. In the experiment, we set the CPU threshold at 50%, 62.5%, 75%, and 87.5%, respectively. The individual difference between the evaluated results and measured one are 6.9%, 4.5%, 3.6% and 0.5%.

We further illustrate the comparison between the measured performance degradation and the results given by FGD, PBV, PPL and CPI[1] in Fig. 9. With CPU utilization thresholds of 50%, 62.5%, 75%, and 87.5% for Mini-Challenge 3, the measured performance degradation results are 95.0%, 30.0%, 10.0% and 2.0%; FGD results are 91.1%,

1. [19] uses *cycles per second* (CPI) to evalaute the performance degradation of an application, and we denote this method as "CPI".

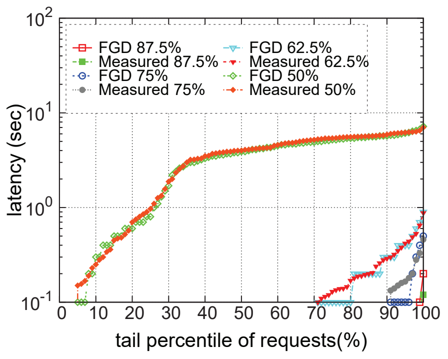


Fig. 8. The comparison between the measured tail latency and the evaluated one obtained by FGD at different CPU utilization thresholds (87.5%, 75%, 62.5%, and 50%).

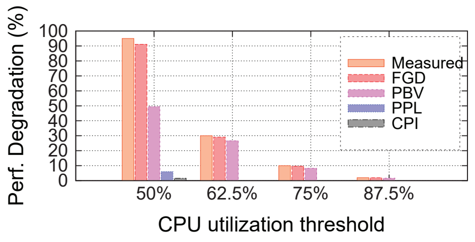


Fig. 9. The comparison between the measured performance degradation and the results given by FGD, PBV, PPL and CPI.

29.1%, 9.7%, and 2.0% while PBV results are 49.4%, 26.7%, 8.3%, and 1.7%; PPL results are 6.0%, 0.0%, 0.0%, and 0.0%, and the CPI results are 1.6%, 0.0%, 0.0%, and 0.0%, respectively. The overall average differences between the measured results and FGD, PBV and PPL are 5.0%, 44.0% and 98.4%, i.e., the FGD results are very close to the measured ones which means our FGD approach is more accurate than PBV, PPL and CPI when evaluating the performance degradation of latency-sensitive applications.

Actually, Fig. 9 shows that the evaluated results obtained by PBV, PPL and CPI are far from the measured results, proving that these three methods are unsuitable to evaluate the performance degradation of latency-sensitive applications, which is consistent with our previous analysis in Section 2.2. In addition, these three methods do not take into account *SLA* which is widely used in cloud datacenters. They might be very useful when applying to their particular occasions like in [5, 12, 19]. However, based on above reasons, we do not use these methods to evaluate performance degradation of production servers in our next experiments.

5.2 Potential Power Budget Saving in Datacenter

We have already verified the accuracy of FGD in previous experiments. In this subsection, first, we introduce the real-world traces from Tencent's datacenter. Then, we select some representative servers based on application types and average load, and evaluate their performance degradation under different CPU utilization thresholds. Further, we show how to set a CPU utilization threshold with acceptable performance degradation for these typical servers. After that, we apply PPCapping on the traces of all 25328 servers hosting latency-sensitive applications to see how much power budget can be saved in a large number of servers with acceptable performance degradation.

5.2.1 Tencent Traces

The experiments are based on real world traces from Tencent's datacenter which runs a large number of latency-sensitive applications. These traces comprise 10166, 7824, and 7338 servers deploying web applications, games, and streams (e.g., videos), respectively. We denote them as *web*, *game*, and *stream* for short. Tencent's traces are collected by agents deployed on running servers and stored in the MySQL database. The monitoring data comprise fields including CPU, memory, disk, network, and alert information. Recall that capping the CPU utilization has almost the same effect as resizing the power budget in Section 2, we only use CPU utilization for our analysis in this paper.

5.2.2 Evaluation on Typical Servers

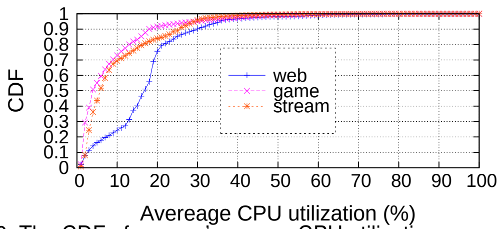


Fig. 10. The CDF of servers' average CPU utilization.

Many previous researches [5, 14, 15] have proven that CPU utilization is closely related to server power, and there is a mapping between CPU utilization and power. First, we explore the CPU utilization feature of servers in Tencent's datacenter. To do so, we calculate the average CPU utilization distribution of these servers. Fig. 10 presents the cumulative distribution function (CDF) of three different types of servers, i.e., *web*, *game* and *stream* servers. The figure shows that, for *web* servers, 80% of servers' average CPU utilizations are lower than 20%. The average CPU utilizations of *game* and *stream* servers are much lower (80% of servers are lower than 13%). A possible reason is that games and streams are much less CPU-intensive.

Without loss of generality, we choose nine representative servers based on application types and average load in our evaluation. We classify the servers into three classes based on their average CPU utilization for each application type, that is *high* for the 10% ranking server, *middle* for 40%, and *low* for 70%. These nine representative servers are denoted by *web-high*, *web-middle*, *web-low*, *game-high*, *game-middle*, *game-low*, *stream-high*, *stream-middle*, and *stream-low*, respectively.

The performance degradation of these nine representative servers given by FGD are shown in Fig. 11. With the help of Fig. 11, we can have an intuitive understanding of *P-Table*. For example, in Fig. 11(b), when CPU threshold is 26%, the 95th percentile latency of requests is 200ms, which means we can set CPU threshold to 26% if SLA requires (95%, 200ms).

The most common feature reflected by these figures is that when the value of CPU threshold is lower and the value of percentage of requests is larger, latency becomes bigger. This trend is consistent with our expectation. If a strict SLA requires 99% of requests should not be affected (i.e., (99%, 0)), as we can see, CPU threshold of *web-low*, *game-low*, *stream-low*, *web-middle*, *game-middle*, *stream-middle*, *web-high*, *game-high*, and *stream-high* are 33%, 15%, 6%, 40%,

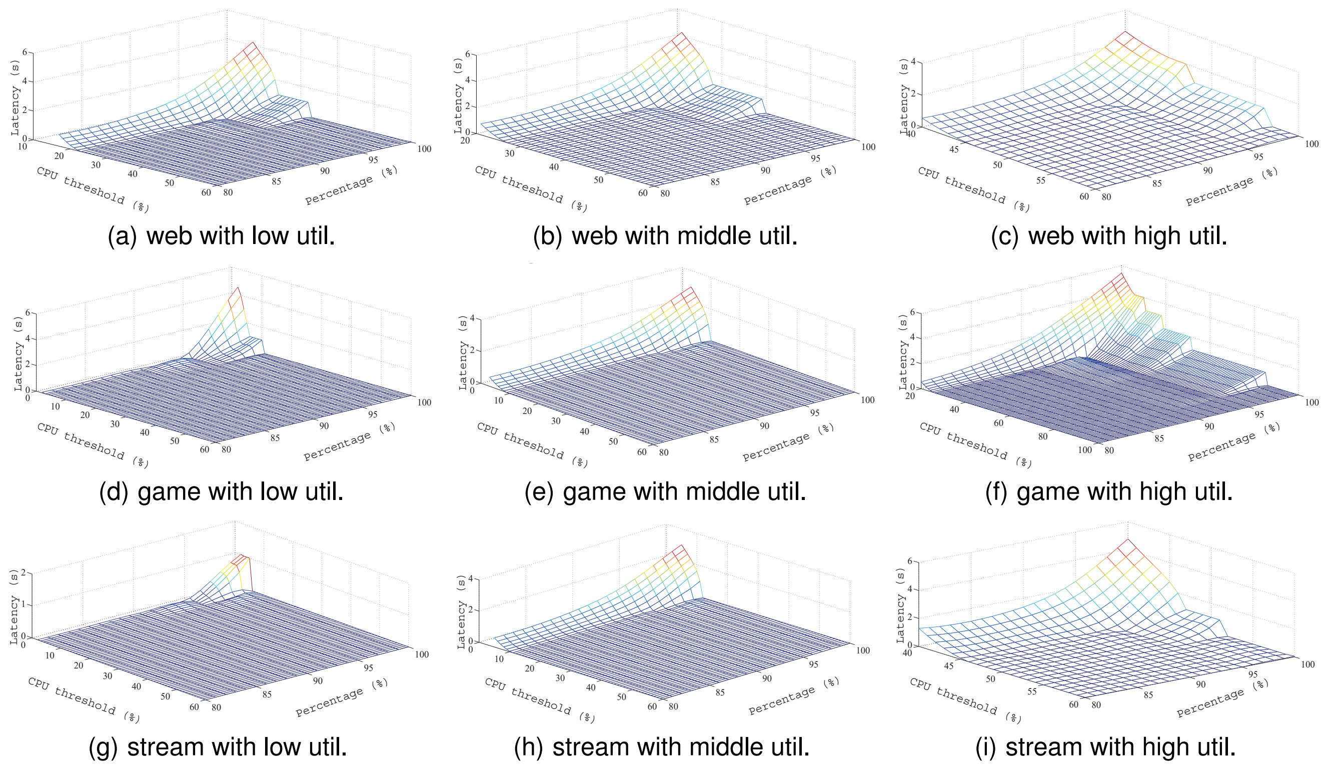


Fig. 11. Tail latencies with different CPU thresholds of typical servers from Tecent.

10%, 12%, 57%, 87%, and 52%, respectively. These values can give us suggestions on capping restrictions for different types of servers.

With further insight of the results in Fig. 11, we can infer that the CPU utilization can be largely restricted without affecting the performance for all servers. For example, *web-high* can reduce its CPU utilization threshold to 57% with almost no performance degradation for all requests, and the CPU utilization threshold can be even lower for *web-middle* and *web-low*. This implies that we can potentially restrict immense power budget with servers deploying latency-sensitive applications.

5.2.3 Evaluation on Large Scale Datacenter

In the previous subsection, we have carried out performance degradation evaluations on some representative servers. Now, we aim to find out how much power budget can be saved for a datacenter. In this experiment, we take seven days traces of all 25328 servers (deploying web applications, games, and streams) in Tencent datacenter to explore the savings of the power budget.

To transform CPU utilization to server power usage, we adopt the model described in [5], $p = p_{idle} + (p_{busy} - p_{idle}) * u_i$, where p_{idle} and p_{busy} refer to the power consumption when the CPU utilization is 0 and 100%, and u_i refers to CPU utilization. For simplicity, we empirically set p_{busy} as 300 W and p_{idle} as 150 W for all servers [5]. Thus, if the CPU utilization of a server is 50%, then its power usage is 225 W. This model is useful when we calculate how much power budget can be saved after determining the CPU utilization threshold.

To explore what percentage of power budget can be saved for all servers with different acceptable performance degradations, we choose three typical percentiles of tail latency, i.e., 95%, 99% and 100%, and plot the results in Fig. 12. Fig. 12(a) demonstrates that even no performance degradation is accepted, we can save 34% of the power budget of all servers. With little performance degradation, we can further restrict the power budget (e.g., we can save 41.3% of the power budget when SLA requirement is (95%, 200ms)).

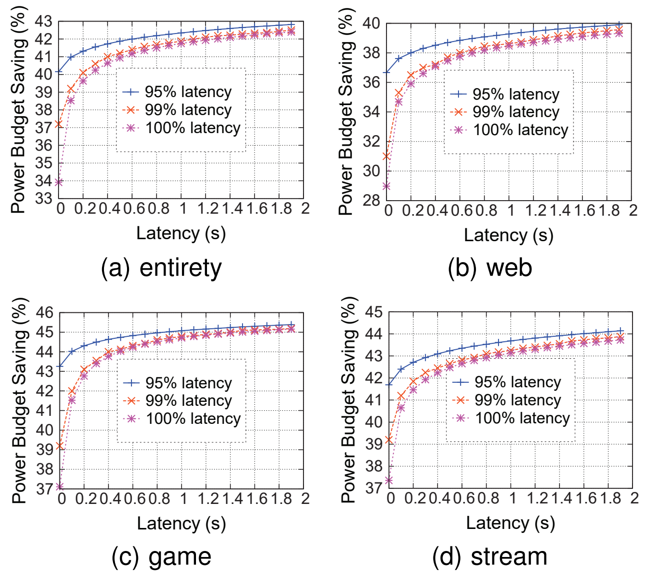


Fig. 12. Power budget savings with three typical percentile tail latencies (95%, 99%, and 100%).

If we divide all these servers into three groups of web applications, games, and streams, their results are shown in Fig. 12(b), 12(c), and 12(d) respectively. As we can see, power budget savings of *web* servers are smaller than that of *game* and *stream* servers. For example, if SLA requirement is (95%, 200ms), power budget savings of *web*, *game*, and *stream* are 38.0%, 44.3%, and 42.7%, respectively. This demonstrates that *game* and *stream* servers are more potential to save power budget in the Tecent datacenter.

In summary, the datacenter is potential to save at least 34% of total power budget. With little performance degradation, the room is even bigger. That is to say, it is very profitable to use FGD as guidance to restrict the power budget of servers.

5.3 Guaranteeing Applications' SLAs

In this subsection, we show our advantage in performance guaranteeing. In particular, we compare PPCapping to Facebook's state-of-the-art power capping technique, Dynamo[24]. Dynamo is a multi-level power management system that monitors the entire power hierarchy, and its lowest level, called leaf controllers, uses a heuristic high-bucket-first method to determine the amount of power that must be cut from each server within the same priority group. We use three physical 16-core nodes (16× Intel(R) Xeon(R) CPU E5-2670 0@2.60GHz, 64G RAM) to simulate a rack. For a fair comparison, we replay three real web server traces belonging to the same priority group, respectively.

Fig. 13 shows three servers' power consumption over time. We manually set the rack's power cap as 720W and web service's SLA as (95%, 200ms) in this scenario. As we can see, at the time point of 1700 minutes, the power of three servers are 292.5W, 234.5W and 223W, respectively, and the total power exceeds the cap incurring a power violation of about 30W. By using PPCapping, we look up *P-Tables* of these three servers, and according to their SLAs, we set 280.5W, 226.5W and 213W for these three servers as their power budgets, respectively. That is, we distribute 12W, 8W, and 10W of the power violation of 30W to three servers to cut, respectively.

Dynamo uses a heuristic high-bucket-first method. As suggested in [24], we set the bucket size is 20W. In this scenario, only server#1 belongs to the highest bucket, and servers #2 and #3 belong to a lower bucket. The power of server#1 exceeds server#2 and #3 by almost three bucket sizes, thus server#1 may take all 30W power cut since Dynamo is heuristic and has no idea about the exactly lowest power cap allowed for each server, which leads to SLA violation.

Fig. 14 illustrates the result of the 95-th percentile latency of requests with power capping. Clearly, PPCapping can meet the SLA requirement (95%, 200ms) for all servers while server #1 of Dynamo violates the requirement severely.

5.4 Improving Datacenter's Power Utilization and Capacity

SLAs guaranteeing rather than the capacity should be the first-order of concern in datacenters deploying lots of latency-sensitive applications. However, Dynamo can hardly guarantee SLA of latency-sensitive applications though it can greatly improve datacenter's capacity. In this subsection, we compare PPCapping with another typical method that

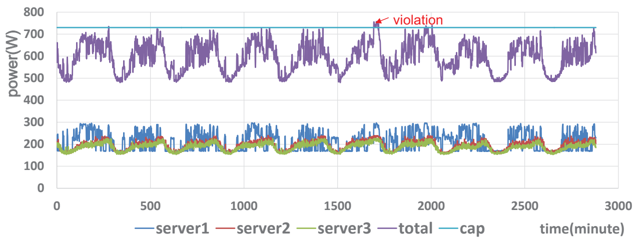


Fig. 13. The respective and total power consumption of three real web servers.

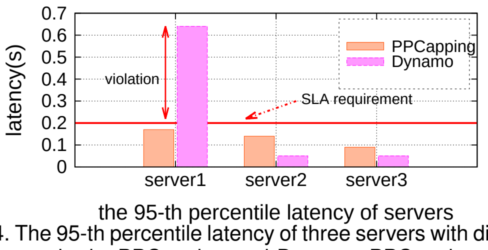


Fig. 14. The 95-th percentile latency of three servers with different power capping methods, PPCapping and Dynamo. PPCapping can meet the SLA constraint (200ms) for all servers while the latency of server #1 (640ms) in Dynamo violates the constraint severely.

can guarantee SLA of latency-sensitive applications, i.e., *observed peak power*, in which servers are provisioned according to their observed peak power. In turn, the number of servers that can be placed in the rack is the quotient of rack's power rating and provisioned power of a server. This strategy is adopted by most existing datacenters [25].

Note that PPCapping applies the FGD method to precisely assign power budget for each server to meet different predefined SLAs thus increasing racks' average power utilization and datacenter capacity. Fig. 15 shows the CDF of racks' average power utilization with different SLA requirements with PPCapping and *observed peak power* methods. Without loss of generality, we assume all servers have the same SLA when calculating the CDF.

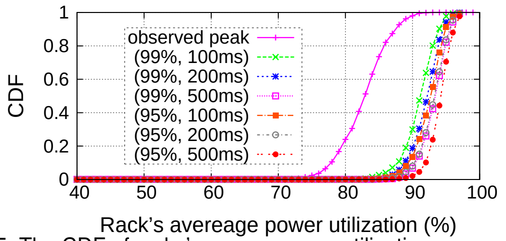


Fig. 15. The CDF of racks' average power utilization.

As we can see, PPCapping can effectively improve the average utilization with 85%-96% for all racks while *observed peak power* achieves 75%-90%. Generally, the less strict of SLA, the higher improvement of racks' average power utilization. For example, PPCapping improves racks' average utilization by 1.12× with applications' SLA (95%, 200ms) compared with *observed peak power*.

With regard to the capacity increment of datacenter, in Table 5, we plot the results of PPCapping compared with *observed peak power*. Table 5 shows that PPCapping can effectively increase datacenter capacity. For example, PPCapping can increase 12.6% of datacenter capacity with SLA constraint of (95%, 200ms), which means a large-scale datacenter with a fixed power supply can hold extra thousands of servers.

TABLE 5
The capacity increment of PPCapping compared with the typical power provisioning strategy, *observed peak power*.

Server	SLA(s)	Capacity increment compared to observed peak
web	(99%, 100ms)	9.7%
	(99%, 200ms)	11.9%
	(95%, 100ms)	13.8%
	(95%, 200ms)	14.6%
game	(99%, 100ms)	8.6%
	(99%, 200ms)	10.7%
	(95%, 100ms)	12.5%
	(95%, 200ms)	13.1%
stream	(99%, 100ms)	6.6%
	(99%, 200ms)	7.9%
	(95%, 100ms)	8.8%
	(95%, 200ms)	9.4%
entirety	(99%, 100ms)	8.7%
	(99%, 200ms)	10.3%
	(95%, 100ms)	12.0%
	(95%, 200ms)	12.6%

6 Discussion

In this section, we discuss some issues related to our work.

Sample interval of CPU utilization. There is correlation between the accuracy of FGD and the sample interval (i.e., Δt in Section 3.2.2) in CPU utilization trace. This sample interval is practically determined by the sampling strategy of datacenter's monitoring system, and it is 5 minutes in Tencent datacenter. Though a finer-grained sample interval may make our algorithm more precise, it comes with a much higher monitoring overhead and probably does harm to applications' SLA and overall system performance. That is the reason why Tencent's choice on the sample interval is relatively conservative.

Trace-based evaluation. FGD relies on the CPU utilization trace of a server. However, if the trace cannot properly reflect the behavior of the workload, it will greatly affect the accuracy of the method. As for unpredictable special days, like Thanksgiving, our method may fail. This is the limitation of all trace-based researches. To our best knowledge, it is still a big challenge to handle the unpredictable situations while meeting the performance requirements of applications. Based on the analysis of Tencent's trace, we select seven days' traces for our evaluation, which can reflect the behavior of servers' workload in common situation. As for servers in other datacenters, the proper length of traces used in evaluation could be different.

Latency-insensitive applications. We concentrate on evaluating the performance degradation of latency-sensitive applications in this paper. However, when fine-grained method is applied to latency-insensitive applications (e.g., batch jobs), it may be not as useful as existing approaches (e.g., PBV and PPL), because the makespan is the most concern of this kind of application instead of the latency of each request. Therefore, our fine-grained method is complementary to existing methods.

7 Related work

Fan et al. [5] expose the severe power over-provisioning problem by investigating the power consumption in the Google datacenter. Further, many researches [5, 9, 11, 26–28] have found that power over-provisioning has greatly impede the increase of datacenter capacity which is an important factor affecting the total cost of ownership (TCO).

To manage power of datacenter and ease power over-provisioning problem, there are three major related fields, power capping, workload scheduling, and performance degradation caused by power management approaches. Power capping and performance degradation evaluation are most related to our work. We briefly introduce these three categories here.

7.1 Power capping

Power capping technology is a straightforward way to ease the power over-provisioning problem and increase the datacenter capacity, which has been studied by many researches [5, 7, 9, 12, 25, 29–37].

Generally, these power capping techniques can be classified into several levels, i.e., *datacenter/cluster-level*, *PDU-level*, *rack-level* and *server-level*. In datacenter-level, for example, Facebook's Dynamo [24] is the first datacenter-wide power management system in a real production. Wang et al. [36] design SHIP to control the power consumption of an entire datacenter dynamically, while Verma et al. [38] propose a correlation based server consolidation scheme to minimize power consumption. Since these methods are control-theoretic frameworks where feedback loops operate at multiple levels, they may suffer from power capping latency and instability [25]. For the PDU-level power capping, statistical profiling-based techniques have been proposed to provision servers within power constraints [39].

While in the rack-level, Wang et al. [40] design control-theoretic methods to optimize system performance and [11, 41] propose heuristic-based methods by controlling rack-level power consumption. The proposed PPCapping is also a rack-level method but we control the rack-level power by setting a precise power budget for each server while meeting the predefined SLAs.

In server-level power capping, control theory has been used to control power of a single server in [29, 31, 32] and Felter et al. [42] apply the open-loop control to maintain servers' power budget by switching power between processor and memory. Gandhi et al. [30] control the power to meet the power budget by inserting idle cycles during execution. Furthermore, [34] proposes heuristic solutions for power budgeting for virtual machines and [12] designs dynamic server power capping techniques for VMs. Ma et al. [35] propose a capping technique to power gate the cores and apply percore DVFS for multi-threaded applications. Intel Node Manager [43] also proposes some techniques to facilitate power capping.

Additionally, there are some recent work on energy storage device sizing framework for fuel cell powered datacenters, which coordinates energy storage devices with power capping policies to handle power shortfalls in datacenters, such as [17, 44]. These solutions could wisely mitigate the impact of power shortfalls, however, with an expensive cost of energy storage devices.

7.2 Workload scheduling

Guosai Wang et al. [4] implements dynamic power management by directly affecting the workload scheduling to fully utilize available power resources thus increasing datacenter capacity. [45–48] use server consolidation approach to ease power over-provisioning. Server consolidation unites as many applications as possible into fewer servers to make fully use of power budget. However, there are some problems with this method in real-world datacenters [20]. First, due to the higher density of applications in a server, a *single point of failure* will affect more applications. Meanwhile, server consolidation is typically based on virtualization, which could bring some overheads and unfair contentions among applications. Finally, there is inevitable cost of application migration when consolidating applications.

7.3 Performance degradation evaluation

Performance degradation of applications caused by power management is another important issue. [5, 10] evaluate the performance degradation by measuring what percentage of time is spent in budget violation (PBV), and [9, 12] evaluate that by measuring how much of the work done is lost (PPL). Although these two methods are suitable in some specialized scenarios (e.g., PPL is very effective in evaluating the performance degradation of non latency-sensitive applications), they have drawbacks in describing the performance degradation of latency-sensitive applications. Our proposed FGD is a fine-grained evaluation method which can accurately evaluate each request's latency of latency-sensitive applications.

Meanwhile, there are also some other methods that are used to evaluate the performance degradation of servers. [45] carries out experiments to measure the performance by consolidating any two jobs. This method can be accurate, but it also incurs a substantial experimental cost. [49] assumes that the servers with similar CPU utilization characteristics suffer from similar performance degradations. This method is efficient when there are abundant jobs whose CPU utilization characteristics are similar. [50] includes memory access time as another factor that affects latency other than the CPU time. It provides a new model for predicting job latency with multiple factors. Similar to the thought of *Workload* accumulation in our work, [42] also assumes that *Workload* not accomplished will be delayed to the next time interval, but it does not give an evaluation of the performance degradation when power budget violation occurs.

8 Conclusions

How to reduce the TCO in turn to improve the profitability of datacenter has been a hot topic in cloud computing. Increasing datacenter capacity with acceptable performance degradation is an effective way. Unfortunately, there is no effective method to measure the performance degradation of latency-sensitive applications at present. In this paper, we analyze the reasons why the state-of-the-art methods are not appropriate to evaluate the impact of power budget violation on latency-sensitive applications' performance. Then we propose a new method, *Fine-Grained Differential Method*, which can provide a fine-grained way to evaluate such an impact precisely. Further, we propose *Precise Power Capping* to improve power utilization and capacity of datacenters. We verify *Fine-Grained Differential Method* by replaying real world web traces and evaluate the performance degradation of 25328 servers in a production datacenter. Experimental results demonstrate that our *Fine-Grained Differential Method* behaves more accurately than other state-of-the-art methods, and we can effectively improve power utilization and datacenter capacity while maintaining the applications' performance degradation within controllable and acceptable range by using *Precise Power Capping*. In particular, PPCapping improves racks' average utilization up to 1.12× and datacenter capacity by 12.6% with SLA of (95%, 200ms) compared with *observed peak power*.

Acknowledgments

The authors would like to thank Tencent Inc, in particular colleagues in Infrastructure Platform of TEG (Technology and Engineering Group), for making their invaluable trace available.

References

[1] "Tencent corporation, http://www.tencent.com/, 2016."
[2] A. Li, X. Yang, S. Kandula, and M. Zhang, "Cloudcmp: comparing public cloud providers," in *proc. of IMC*, pp. 1–14, ACM, 2010.
[3] C. Isci, A. Buyuktosunoglu, C. Cher, P. Bose, and M. Martonosi, "An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget," in *proc. of MICRO*, pp. 347–358, IEEE Computer Society, 2006.
[4] G. Wang, S. Wang, B. Luo, W. Shi, Y. Zhu, W. Yang, D. Hu, L. Huang, X. Jin, and W. Xu, "Increasing large-scale data center capacity by statistical power control," in *proc. of EuroSys*, p. 8, ACM, 2016.
[5] X. Fan, W. Weber, and L. A. Barroso, "Power provisioning for a warehouse-sized computer," in *proc. of ISCA*, vol. 35, pp. 13–23, ACM, 2007.
[6] T. Imada, M. Sato, Y. Hotta, and H. Kimura, "Power management of distributed web savers by controlling server power state and traffic prediction for qos," in *proc. of IPDPS*, pp. 1–8, IEEE, 2008.
[7] A. Saifullah, S. Sankar, J. Liu, C. Lu, R. Chandra, and B. Priyantha, "Capnet: A real-time wireless management network for data center power capping," in *proc. of RTSS*, pp. 334–345, 2014.
[8] Y. Wang, X. Wang, M. Chen, and X. Zhu, "Partic: Power-aware response time control for virtualized web servers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 2, pp. 323–336, 2011.
[9] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu, "No power struggles: Coordinated multi-level power management for the data center," in *proc. of ASPLOS*, vol. 36, pp. 48–59, ACM, 2008.
[10] X. Wang and M. Chen, "Cluster-level feedback power control for performance optimization," in *proc. of HPCA*, pp. 101–110, IEEE, 2008.
[11] P. Ranganathan, P. Leech, D. Irwin, and J. Chase, "Ensemble-level power management for dense blade servers," in *proc. of ISCA*, vol. 34, pp. 66–77, IEEE Computer Society, 2006.
[12] H. Chen, C. Hankendi, M. C. Caramanis, and A. K. Coskun, "Dynamic server power capping for enabling

data center participation in power markets," in *proc. of ICCAD*, pp. 122–129, IEEE Press, 2013.

[13] L. A. Barroso, J. Clidaras, and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synthesis lectures on computer architecture*, vol. 8, no. 3, pp. 1–154, 2013.

[14] S. Rivoire, P. Ranganathan, and C. Kozyrakis, "A comparison of high-level full-system power models.," *HotPower*, vol. 8, pp. 3–3, 2008.

[15] D. Meisner and T. F. Wenisch, "Peak power modeling for data center servers with switched-mode power supplies," in *proc. of ISLPED*, pp. 319–324, ACM, 2010.

[16] S. Wang, J. Chen, J. Liu, and X. Liu, "Power saving design for servers under response time constraint," in *proc. of ECRTS*, pp. 123–132, IEEE, 2010.

[17] S. Govindan, D. Wang, A. Sivasubramaniam, and B. Urgaonkar, "Leveraging stored energy for handling power emergencies in aggressively provisioned datacenters," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, pp. 75–86, 2012.

[18] K. Tang, D. Tiwari, S. Gupta, P. Huang, Q. Lu, C. Engelmann, and X. He, "Power-capping aware checkpointing: On the interplay among power-capping, temperature, reliability, performance, and energy," in *proc. of DSN*, pp. 311–322, 2016.

[19] R. Bianchini, R. Bianchini, R. Bianchini, R. Bianchini, and R. Bianchini, "Fast power and energy management for future many-core systems," *Acm Transactions on Modeling and Performance Evaluation of Computing Systems*, vol. 2, no. 3, p. 17, 2017.

[20] D. Meisner, C. M. Sadler, L. A. B., W. Weber, and T. F. Wenisch, "Power management of online data-intensive services," in *proc. of ISCA*, pp. 319–330, IEEE, 2011.

[21] M. M. Baldi, T. G. Crainic, G. Perboli, and R. Tadei, "Branch-and-price and beam search algorithms for the variable cost and size bin packing problem with optional items," *Annals of Operations Research*, vol. 222, no. 1, pp. 125–141, 2014.

[22] Y. Cui, Y. Yao, and Y.-P. Cui, "Hybrid approach for the two-dimensional bin packing problem with two-staged patterns," *International Transactions in Operational Research*, vol. 23, no. 3, pp. 539–549, 2016.

[23] "Vast challenge 2013, http://www.vacommunity.org/VAST+Challenge+2013\%3A+Mini-Challenge+3, 2016."

[24] Q. Wu, Q. Deng, L. Ganesh, C.-H. Hsu, Y. Jin, S. Kumar, B. Li, J. Meza, and Y. J. Song, "Dynamo: facebook's data center-wide power management system," in *proc. of ISCA*, pp. 469–480, IEEE, 2016.

[25] A. A. Bhattacharya, D. Culler, A. Kansal, and S. Govindan, "The need for speed and stability in data center power capping," in *proc. of IGCC*, pp. 183–193, 2012.

[26] A. Gandhi, M. Harchol-Balter, R. Das, and C. Lefurgy, "Optimal power allocation in server farms," in *proc. of SIGMETRICS*, vol. 37, pp. 157–168, ACM, 2009.

[27] S. Pelley, D. Meisner, P. Zandevakili, T. F. Wenisch, and J. Underwood, "Power routing: dynamic power provisioning in the data center," in *proc. of ASPLOS*, vol. 45, pp. 231–242, ACM, 2010.

[28] V. Kontorinis, L. E. Zhang, B. Aksanli, J. Sampson, H. Homayoun, E. Pettis, D. M. Tullsen, and T. S. Rosing, "Managing distributed ups energy for effective power capping in data centers," in *proc. of ISCA*, pp. 488–499, IEEE, 2012.

[29] C. Lefurgy, X. Wang, and M. Ware, "Power capping: a prelude to power shifting," in *proc. of CLUSTER*, vol. 11, pp. 183–195, Springer, 2008.

[30] A. Gandhi, M. Harchol-Balter, R. Das, J. O. Kephart, and C. Lefurgy, "Power capping via forced idleness," in *proc. of WEED*, 2009.

[31] R. J. Minerick, V. W. Freeh, and P. M. Kogge, "Dynamic power management using feedback," in *proc. of COLP*, Citeseer, 2002.

[32] K. Skadron, T. Abdelzaher, and M. R. Stan, "Control-theoretic techniques and thermal-rc modeling for accurate and localized dynamic thermal management," in *proc. of HPCA*, pp. 17–28, IEEE, 2002.

[33] C. Hankendi, S. Reda, and A. K. Coskun, "vcap: Adaptive power capping for virtualized servers," in *proc. of ISLPED*, pp. 415–420, IEEE Press, 2013.

[34] R. Nathuji, K. Schwan, A. Somani, and Y. Joshi, "Vpm tokens: virtual machine-aware power budgeting in datacenters," in *proc. of CLUSTER*, vol. 12, pp. 189–203, Springer, 2009.

[35] K. Ma and X. Wang, "Pgcapping: exploiting power gating for power capping and core lifetime balancing in cmps," in *proc. of PACT*, pp. 13–22, ACM, 2012.

[36] X. Wang, M. Chen, C. Lefurgy, and T. W. Keller, "Ship: A scalable hierarchical power control architecture for large-scale data centers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 1, pp. 168–176, 2012.

[37] W. Hou, C. Yu, L. Guo, and X. Wei, "Virtual network embedding for power savings of servers and switches in elastic data center networks," *Science China Information Sciences*, vol. 59, no. 12, pp. 1–14, 2016.

[38] A. Verma, G. Dasgupta, T. K. Nayak, P. De, and R. Kothari, "Server workload analysis for power minimization using consolidation," in *proc. of ATC*, pp. 28–28, USENIX Association, 2009.

[39] S. Govindan, J. Choi, B. Urgaonkar, A. Sivasubramaniam, and A. Baldini, "Statistical profiling-based techniques for effective power provisioning in data centers," in *proc. of EurSys*, pp. 317–330, ACM, 2009.

[40] X. Wang, M. Chen, and X. Fu, "Mimo power control for high-density servers in an enclosure," *Transactions on Parallel and Distributed Systems*, vol. 21, no. 10, pp. 1412–1426, 2010.

[41] M. E. Femal and V. W. Freeh, "Boosting data center performance through non-uniform power allocation," in *proc. of ICAC*, pp. 250–261, IEEE, 2005.

[42] W. Felter, K. Rajamani, T. Keller, and C. Rusu, "A performance-conserving approach for reducing peak power consumption in server systems," in *proc. of ICS*, pp. 293–302, ACM, 2005.

[43] "Intel node manager, http://www.intel.com/content/www/us/en/data-center/data-center-management/node-manager-general.html, 2016."

[44] Y. Li, D. Wang, S. Ghose, J. Liu, S. Govindan, S. James, E. Peterson, J. Siegler, R. Ausavarungnirun, and O. Mutlu, "Sizecap: Efficiently handling power surges in fuel cell powered data centers," in *proc. of HPCA*, pp. 444–456, IEEE, 2016.

[45] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *proc. of MICRO*, pp. 248–259, ACM, 2011.

[46] L. Tang, J. Mars, and M. L. Soffa, "Compiling for niceness: Mitigating contention for qos in warehouse scale computers," in *proc. of CGO*, pp. 1–12, ACM, 2012.

[47] N. Bila, E. de Lara, K. Joshi, H. A. Lagar-Cavilla, M. Hiltunen, and M. Satyanarayanan, "Jettison: efficient idle desktop consolidation with partial vm migration," in *proc. of the 7th ACM EurSys*, pp. 211–224, ACM, 2012.

[48] M. Lin, A. Wierman, L. L. Andrew, and E. Thereska, "Dynamic right-sizing for power-proportional data centers," *IEEE/ACM Transactions on Networking*, vol. 21, no. 5, pp. 1378–1391, 2013.

[49] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," *proc. of MICRO*, vol. 34, pp. 17–30, 2014.

[50] B. Su, J. L. Greathouse, J. Gu, M. Boyer, L. Shen, and

Z. Wang, “Implementing a leading loads performance predictor on commodity processors,” in *proc. of ATC*, pp. 205–210, USENIX Association, 2014.