Dynamic Resource Allocation using Virtual Machines for Cloud Computing Environment

Zhen Xiao, *Senior Member, IEEE,* Weijia Song, and Qi Chen

Abstract—Cloud computing allows business customers to scale up and down their resource usage based on needs. Many of the touted gains in the cloud model come from resource multiplexing through virtualization technology. In this paper, we present a system that uses virtualization technology to allocate data center resources dynamically based on application demands and support green computing by optimizing the number of servers in use. We introduce the concept of "skewness" to measure the unevenness in the multi-dimensional resource utilization of a server. By minimizing skewness, we can combine different types of workloads nicely and improve the overall utilization of server resources. We develop a set of heuristics that prevent overload in the system effectively while saving energy used. Trace driven simulation and experiment results demonstrate that our algorithm achieves good performance.

Index Terms—Cloud Computing, Resource Management, Virtualization, Green Computing.

1 INTRODUCTION

The elasticity and the lack of upfront capital investment offered by cloud computing is appealing to many businesses. There is a lot of discussion on the benefits and costs of the cloud model and on how to move legacy applications onto the cloud platform. Here we study a different problem: how can a cloud service provider best multiplex its virtual resources onto the physical hardware? This is important because much of the touted gains in the cloud model come from such multiplexing. Studies have found that servers in many existing data centers are often severely under-utilized due to over-provisioning for the peak demand [1] [2]. The cloud model is expected to make such practice unnecessary by offering automatic scale up and down in response to load variation. Besides reducing the hardware cost, it also saves on electricity which contributes to a significant portion of the operational expenses in large data centers.

Virtual machine monitors (VMMs) like Xen provide a mechanism for mapping virtual machines (VMs) to physical resources [3]. This mapping is largely hidden from the cloud users. Users with the Amazon EC2 service [4], for example, do not know where their VM instances run. It is up to the cloud provider to make sure the underlying physical machines (PMs) have sufficient resources to meet their needs. VM live migration technology makes it possible to change the mapping between VMs and PMs while applications are running [5], [6]. However, a policy issue remains as how to decide the mapping adaptively so that the resource demands of VMs are met while the number of PMs used is minimized. This is challenging when the resource needs of VMs are heterogeneous due to the diverse set of applications they run and vary with time as the workloads grow and shrink. The capacity of PMs can also be heterogenous because multiple generations of hardware co-exist in a data center.

• *Z. Xiao, W. Song, and Q. Chen are with the Department of Computer Science, Peking University.*
E-mail: {xiaozhen,songweijia}@pku.edu.cn, chenqi@net.pku.edu.cn

We aim to achieve two goals in our algorithm:

- overload avoidance: the capacity of a PM should be sufficient to satisfy the resource needs of all VMs running on it. Otherwise, the PM is overloaded and can lead to degraded performance of its VMs.
- green computing: the number of PMs used should be minimized as long as they can still satisfy the needs of all VMs. Idle PMs can be turned off to save energy.

There is an inherent trade-off between the two goals in the face of changing resource needs of VMs. For overload avoidance, we should keep the utilization of PMs low to reduce the possibility of overload in case the resource needs of VMs increase later. For green computing, we should keep the utilization of PMs reasonably high to make efficient use of their energy.

In this paper, we present the design and implementation of an automated resource management system that achieves a good balance between the two goals. We make the following contributions:

- We develop a resource allocation system that can avoid overload in the system effectively while minimizing the number of servers used.
- We introduce the concept of "skewness" to measure the uneven utilization of a server. By minimizing skewness, we can improve the overall utilization of servers in the face of multi-dimensional resource constraints.
- We design a load prediction algorithm that can capture the future resource usages of applications accurately without looking inside the VMs. The algorithm can capture the rising trend of resource usage patterns and help reduce the placement churn significantly.

The rest of the paper is organized as follows. Section 2 provides an overview of our system and Section 3 describes our algorithm to predict resource usage. The details of our

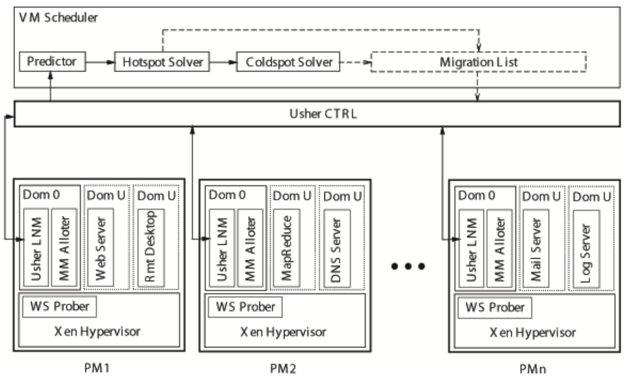


Fig. 1. System Architecture

algorithm are presented in Section 4. Section 5 and 6 present simulation and experiment results, respectively. Section 7 discusses related work. Section 8 concludes.

2 System overview

The architecture of the system is presented in Figure 1. Each PM runs the Xen hypervisor (VMM) which supports a privileged domain 0 and one or more domain U [3]. Each VM in domain U encapsulates one or more applications such as Web server, remote desktop, DNS, Mail, Map/Reduce, etc. We assume all PMs share a backend storage.

The multiplexing of VMs to PMs is managed using the Usher framework [7]. The main logic of our system is implemented as a set of plug-ins to Usher. Each node runs an Usher local node manager (LNM) on domain 0 which collects the usage statistics of resources for each VM on that node. The CPU and network usage can be calculated by monitoring the scheduling events in Xen. The memory usage within a VM, however, is not visible to the hypervisor. One approach is to infer memory shortage of a VM by observing its swap activities [8]. Unfortunately, the guest OS is required to install a separate swap partition. Furthermore, it may be too late to adjust the memory allocation by the time swapping occurs. Instead we implemented a working set prober (WS Prober) on each hypervisor to estimate the working set sizes of VMs running on it. We use the random page sampling technique as in the VMware ESX Server [9].

The statistics collected at each PM are forwarded to the Usher central controller (Usher CTRL) where our VM scheduler runs. The VM Scheduler is invoked periodically and receives from the LNM the resource demand history of VMs, the capacity and the load history of PMs, and the current layout of VMs on PMs.

The scheduler has several components. The predictor predicts the future resource demands of VMs and the future load of PMs based on past statistics. We compute the load of a PM by aggregating the resource usage of its VMs. The details of the load prediction algorithm will be described in the next section. The LNM at each node first attempts to satisfy the new demands locally by adjusting the resource allocation of VMs sharing the same VMM. Xen can change the CPU allocation among the VMs by adjusting their weights in its CPU scheduler. The MM Alloter on domain 0 of each node is responsible for adjusting the local memory allocation.

The hot spot solver in our VM Scheduler detects if the resource utilization of any PM is above the *hot threshold* (i.e., a hot spot). If so, some VMs running on them will be migrated away to reduce their load. The cold spot solver checks if the average utilization of actively used PMs (APMs) is below the *green computing threshold*. If so, some of those PMs could potentially be turned off to save energy. It identifies the set of PMs whose utilization is below the *cold threshold* (i.e., cold spots) and then attempts to migrate away all their VMs. It then compiles a migration list of VMs and passes it to the Usher CTRL for execution.

3 Predicting Future Resource Needs

We need to predict the future resource needs of VMs. As said earlier, our focus is on Internet applications. One solution is to look inside a VM for application level statistics, e.g., by parsing logs of pending requests. Doing so requires modification of the VM which may not always be possible. Instead, we make our prediction based on the past external behaviors of VMs. Our first attempt was to calculate an exponentially weighted moving average (EWMA) using a TCP-like scheme:

$$E(t) = \alpha * E(t-1) + (1-\alpha) * O(t), 0 \le \alpha \le 1$$

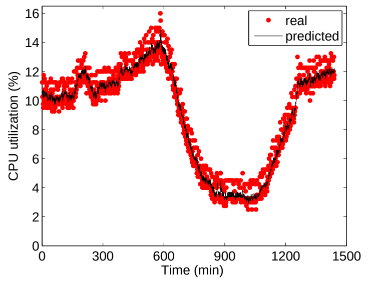
where $E(t)$ and $O(t)$ are the estimated and the observed load at time t, respectively. α reflects a tradeoff between stability and responsiveness.

We use the EWMA formula to predict the CPU load on the DNS server in our university. We measure the load every minute and predict the load in the next minute. Figure 2 (a) shows the results for $\alpha = 0.7$. Each dot in the figure is an observed value and the curve represents the predicted values. Visually, the curve cuts through the middle of the dots which indicates a fairly accurate prediction. This is also verified by the statistics in Table 1. The parameters in the parenthesis are the α values. W is the length of the measurement window (explained later). The "median" error is calculated as a percentage of the observed value: $|E(t) - O(t)|/O(t)$. The "higher" and "lower" error percentages are the percentages of predicted values that are higher or lower than the observed values, respectively. As we can see, the prediction is fairly accurate with roughly equal percentage of higher and lower values.

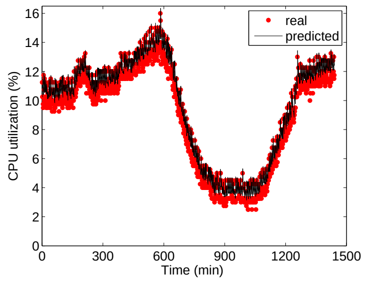
TABLE 1
Load prediction algorithms

	ewma(0.7) W = 1	fusd(-0.2, 0.7) W = 1	fusd(-0.2, 0.7) W = 8
median error	5.6%	9.4%	3.3%
high error	56%	77%	58%
low error	44%	23%	41%

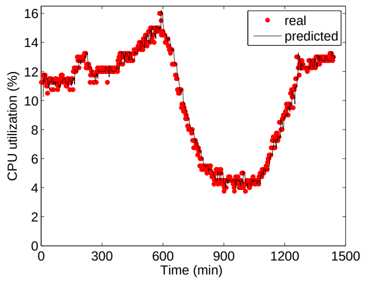
Although seemingly satisfactory, this formula does not capture the rising trends of resource usage. For example, when we see a sequence of $O(t) = 10, 20, 30$, and 40, it is reasonable to predict the next value to be 50. Unfortunately,



(a) EWMA: $\alpha = 0.7$, $W = 1$



(b) FUSD: $\uparrow \alpha = -0.2$, $\downarrow \alpha = 0.7$, $W = 1$



(c) FUSD: $\uparrow \alpha = -0.2$, $\downarrow \alpha = 0.7$, $W = 8$

Fig. 2. CPU load prediction for the DNS server at our university. W is the measurement window.

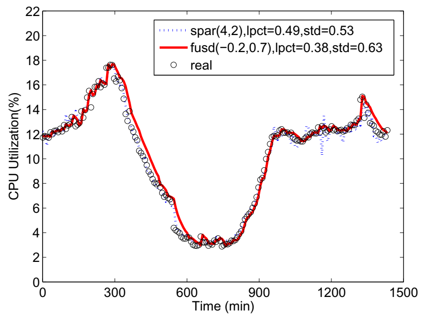


Fig. 3. Comparison of SPAR and FUSD

when α is between 0 and 1, the predicted value is always between the historic value and the observed one. To reflect the "acceleration", we take an innovative approach by setting α to a negative value. When $-1 \leq \alpha < 0$, the above formula can be transformed into the following:

$$\begin{aligned} E(t) &= -|\alpha| * E(t-1) + (1+|\alpha|) * O(t) \\ &= O(t) + |\alpha| * (O(t) - E(t-1)) \end{aligned}$$

On the other hand, when the observed resource usage is going down, we want to be conservative in reducing our estimation. Hence, we use two parameters, $\uparrow \alpha$ and $\downarrow \alpha$, to control how quickly $E(t)$ adapts to changes when $O(t)$ is increasing or decreasing, respectively. We call this the FUSD (Fast Up and Slow Down) algorithm. Figure 2 (b) shows the effectiveness of the FUSD algorithm for $\uparrow \alpha = -0.2$, $\downarrow \alpha = 0.7$. (These values are selected based on field experience with traces collected for several Internet applications.) Now the predicted values are higher than the observed ones most of the time: 77% according to Table 1. The median error is increased to 9.4% because we trade accuracy for safety. It is still quite acceptable nevertheless.

So far we take $O(t)$ as the last observed value. Most applications have their SLOs specified in terms of a certain percentiles of requests meeting a specific performance level. More generally, we keep a window of W recently observed values and take $O(t)$ as a high percentile of them. Figure 2 (c) shows the result when $W = 8$ and we take the $90\%th$ percentile of the peak resource demand. The figure shows that the prediction gets substantially better.

We have also investigated other prediction algorithms. Linear Auto-Regression(AR) models, for example, are broadly adopted in load prediction by other works [10] [11] [12]. It models a predictive value as linear function of its past observations. Model parameters are determined by training with historical values. AR predictors are capable of incorporating the seasonal pattern of load change. For instance, the SPAR(4,2) [10] estimate the future logging rate of MSN clients from six past observations, two of which are the latest observations and the other four at the same time in the last four weeks.

We compare SPAR(4,2) and FUSD(-0.2,0.7) in figure 3. 'lpct' refers to the percentage of low errors while 'std' refers to standard deviation. Both algorithms are used to predict the CPU utilization of the aforementioned DNS server in a one-day duration. The predicting window is eight minute. The standard deviation (std) of SPAR (4,2) is about 16% smaller than that of FUSD (-0.2,0.7), which means SPAR (4,2) achieves sightly better percision. This is because it takes advantage of tiding pattern of the load. However, SPAR(4,2) neither avoid low prediction nor smooth the load. The requirement of a training phase to determine parameters is inconvenient, especially when the load pattern changes. Therefore we adopt the simpler EWMA variance. Thorough investigation on prediction algorithms are left as future work.

As we will see later in the paper, the prediction algorithm plays an important role in improving the stability and performance of our resource allocation decisions.

4 The Skewness Algorithm

We introduce the concept of *skewness* to quantify the unevenness in the utilization of multiple resources on a server. Let n be the number of resources we consider and r_i be the utilization of the i-th resource. We define the resource skewness of a server p as

$$skewness(p) = \sqrt{\sum_{i=1}^{n} (\frac{r_i}{\overline{r}} - 1)^2}$$

where $\overline{r}$ is the average utilization of all resources for server p. In practice, not all types of resources are performance critical and hence we only need to consider bottleneck resources in the above calculation. By minimizing the $skewness$, we can combine different types of workloads nicely and improve the overall utilization of server resources. In the following, we describe the details of our algorithm.

Analysis of the algorithm is presented in Section 1 in the complementary file.

4.1 Hot and cold spots

Our algorithm executes periodically to evaluate the resource allocation status based on the predicted future resource demands of VMs. We define a server as a *hot spot* if the utilization of any of its resources is above a *hot threshold*. This indicates that the server is overloaded and hence some VMs running on it should be migrated away. We define the *temperature* of a hot spot p as the square sum of its resource utilization beyond the hot threshold:

$$temperature(p) = \sum_{r \in R} (r - r_t)^2$$

where R is the set of *overloaded* resources in server p and r_t is the hot threshold for resource r. (Note that only overloaded resources are considered in the calculation.) The temperature of a hot spot reflects its degree of overload. If a server is not a hot spot, its temperature is zero.

We define a server as a *cold spot* if the utilizations of all its resources are below a *cold threshold*. This indicates that the server is mostly idle and a potential candidate to turn off to save energy. However, we do so only when the average resource utilization of all actively used servers (i.e., APMs) in the system is below a *green computing threshold*. A server is actively used if it has at least one VM running. Otherwise, it is inactive. Finally, we define the *warm threshold* to be a level of resource utilization that is sufficiently high to justify having the server running but not so high as to risk becoming a hot spot in the face of temporary fluctuation of application resource demands.

Different types of resources can have different thresholds. For example, we can define the hot thresholds for CPU and memory resources to be 90% and 80%, respectively. Thus a server is a hot spot if either its CPU usage is above 90% or its memory usage is above 80%.

4.2 Hot spot mitigation

We sort the list of hot spots in the system in descending temperature (i.e., we handle the hottest one first). Our goal is to eliminate all hot spots if possible. Otherwise, keep their temperature as low as possible. For each server p, we first decide which of its VMs should be migrated away. We sort its list of VMs based on the resulting temperature of the server if that VM is migrated away. We aim to migrate away the VM that can reduce the server's temperature the most. In case of ties, we select the VM whose removal can reduce the skewness of the server the most. For each VM in the list, we see if we can find a destination server to accommodate it. The server must not become a hot spot after accepting this VM. Among all such servers, we select one whose skewness can be reduced the most by accepting this VM. Note that this reduction can be negative which means we select the server whose skewness increases the least. If a destination server is found, we record the migration of the VM to that server and update the predicted load of related servers. Otherwise, we move on to the next VM in the list and try to find a destination server for it. As long as we can find a destination server for any of its VMs, we consider this run of the algorithm a success and then move on to the next hot spot. Note that each run of the algorithm migrates away at most one VM from the overloaded server. This does not necessarily eliminate the hot spot, but at least reduces its temperature. If it remains a hot spot in the next decision run, the algorithm will repeat this process. It is possible to design the algorithm so that it can migrate away multiple VMs during each run. But this can add more load on the related servers during a period when they are already overloaded. We decide to use this more conservative approach and leave the system some time to react before initiating additional migrations.

4.3 Green computing

When the resource utilization of active servers is too low, some of them can be turned off to save energy. This is handled in our green computing algorithm. The challenge here is to reduce the number of active servers during low load without sacrificing performance either now or in the future. We need to avoid oscillation in the system.

Our green computing algorithm is invoked when the average utilizations of all resources on active servers are below the green computing threshold. We sort the list of cold spots in the system based on the ascending order of their memory size. Since we need to migrate away all its VMs before we can shut down an under-utilized server, we define the memory size of a cold spot as the aggregate memory size of all VMs running on it. Recall that our model assumes all VMs connect to a shared back-end storage. Hence, the cost of a VM live migration is determined mostly by its memory footprint. The Section 7 in the complementary file explains why the memory is a good measure in depth. We try to eliminate the cold spot with the lowest cost first.

For a cold spot p, we check if we can migrate all its VMs somewhere else. For each VM on p, we try to find a destination server to accommodate it. The resource utilizations of the server after accepting the VM must be below the *warm threshold*. While we can save energy by consolidating under-utilized servers, overdoing it may create hot spots in the future. The warm threshold is designed to prevent that. If multiple servers satisfy the above criterion, we prefer one that is not a current cold spot. This is because increasing load on a cold spot reduces the likelihood that it can be eliminated. However, we will accept a cold spot as the destination server if necessary. All things being equal, we select a destination server whose skewness can be reduced the most by accepting this VM. If we can find destination servers for all VMs on a cold spot, we record the sequence of migrations and update the predicted load of related servers. Otherwise, we do not migrate any of its VMs. The list of cold spots is also updated because some of them may no longer be cold due to the proposed VM migrations in the above process.

The above consolidation adds extra load onto the related servers. This is not as serious a problem as in the hot spot mitigation case because green computing is initiated only when the load in the system is low. Nevertheless, we want to bound the extra load due to server consolidation. We restrict the number of cold spots that can be eliminated in each run of the algorithm to be no more than a certain percentage of active servers in the system. This is called the *consolidation limit*.

Note that we eliminate cold spots in the system only when the average load of all active servers (APMs) is below the green computing threshold. Otherwise, we leave those cold spots there as potential destination machines for future offloading. This is consistent with our philosophy that green computing should be conducted conservatively.

4.4 Consolidated movements

The movements generated in each step above are not executed until all steps have finished. The list of movements are then consolidated so that each VM is moved at most once to its final destination. For example, hot spot mitigation may dictate a VM to move from PM A to PM B, while green computing dictates it to move from PM B to PM C. In the actual execution, the VM is moved from A to C directly.

5 Simulations

We evaluate the performance of our algorithm using trace driven simulation. Note that our simulation uses the same code base for the algorithm as the real implementation in the experiments. This ensures the fidelity of our simulation results. Traces are per-minute server resource utilization, such as CPU rate, memory usage, and network traffic statistics, collected using tools like "perfmon" (Windows), the "/proc" file system (Linux), "pmstat/vmstat/netstat" commands (Solaris), etc.. The raw traces are pre-processed into "Usher" format so that the simulator can read them. We collected the traces from a variety of sources:

- Web InfoMall: the largest online Web archive in China (i.e., the counterpart of Internet Archive in the US) with more than three billion archived Web pages.
- RealCourse: the largest online distance learning system in China with servers distributed across 13 major cities.
- AmazingStore: the largest P2P storage system in China.

We also collected traces from servers and desktop computers in our university including one of our mail servers, the central DNS server, and desktops in our department.We post-processed the traces based on days collected and use random sampling and linear combination of the data sets to generate the workloads needed. All simulation in this section uses the real trace workload unless otherwise specified.

The default parameters we use in the simulation are shown in Table 2. We used the FUSD load prediction algorithm with $\uparrow \alpha = -0.2$, $\downarrow \alpha = 0.7$, and $W = 8$. In a dynamic system, those parameters represent good knobs to tune the performance of the system adaptively. We choose the default parameter values based on empirical experience working with many Internet applications. In the future, we plan to explore using AI or control theoretic approach to find near optimal values automatically.

TABLE 2
Parameters in our simulation

symbol	meaning	value
h	hot threshold	0.9
c	cold threshold	0.25
w	warm threshold	0.65
g	green computing threshold	0.4
l	consolidation limit	0.05

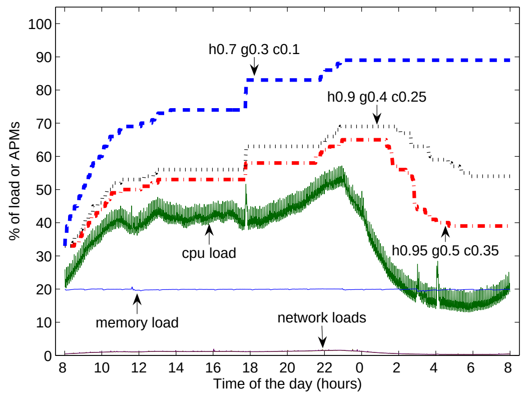


Fig. 4. Impact of thresholds on the number of APMs

5.1 Effect of thresholds on APMs

We first evaluate the effect of the various thresholds used in our algorithm. We simulate a system with 100 PMs and 1000 VMs (selected randomly from the trace). We use random VM to PM mapping in the initial layout. The scheduler is invoked once per minute. The bottom part of Figure 4 show the daily load variation in the system. The x-axis is the time of the day starting at 8am. The y-axis is overloaded with two meanings: the percentage of the load or the percentage of APMs (i.e., Active PMs) in the system. Recall that a PM is *active* (i.e., an APM) if it has at least one VM running. As can be seen from the figure, the CPU load demonstrates diurnal patterns which decreases substantially after midnight. The memory consumption is fairly stable over the time. The network utilization stays very low.

The top part of figure 4 shows how the percentage of APMs vary with the load for different thresholds in our algorithm. For example, 'h0.7 g0.3 c0.1' means that the hot, the green computing, and the cold thresholds are 70%, 30%, and 10%, respectively. Parameters not shown in the figure take the default values in Table 2. Our algorithm can be made more or less aggressive in its migration decision by tuning the thresholds. The figure shows that lower hot thresholds cause more aggressive migrations to mitigate hot spots in the system and increases the number of APMs, and higher cold and green computing thresholds cause more aggressive consolidation which leads to a smaller number of APMs. With the default thresholds in Table 2, the percentage of APMs in our algorithm follows the load pattern closely.

To examine the performance of our algorithm in more extreme situations, we also create a synthetic workload which mimics the shape of a sine function (only the positive part) and ranges from 15% to 95% with a 20% random fluctuation.

It has a much larger peak-to-mean ratio than the real trace. The results are shown in Section 2 of the supplementary file.

5.2 Scalability of the algorithm

We evaluate the scalability of our algorithm by varying the number of VMs in the simulation between 200 and 1400. The ratio of VM to PM is 10:1. The results are shown in Figure 5. The left figure shows that the average decision time of our algorithm increases with the system size. The speed of increase is between linear and quadratic. We break down the decision time into two parts: hot spot mitigation (marked as 'hot') and green computing (marked as 'cold'). We find that hot spot mitigation contributes more to the decision time. We also find that the decision time for the synthetic workload is higher than that for the real trace due to the large variation in the synthetic workload. With 140 PMs and 1400 VMs, the decision time is about 1.3 seconds for the synthetic workload and 0.2 second for the real trace.

The middle figure shows the average number of migrations in the whole system during each decision. The number of migrations is small and increases roughly linearly with the system size. We find that hot spot contributes more to the number of migrations. We also find that the number of migrations in the synthetic workload is higher than that in the real trace. With 140 PMs and 1400 VMs, on average each run of our algorithm incurs about three migrations in the whole system for the synthetic workload and only 1.3 migrations for the real trace. This is also verified by the right figure which computes the average number of migrations per VM in each decision. The figure indicates that each VM experiences a tiny, roughly constant number of migrations during a decision run, independent of the system size. This number is about 0.0022 for the synthetic workload and 0.0009 for the real trace. This translates into roughly one migration per 456 or 1174 decision intervals, respectively. The stability of our algorithm is very good.

We also conduct simulations by varying the VM to PM ratio. With a higher VM to PM ratio, the load is distributed more evenly among the PMs. The results are presented in Section 4 of the supplementary file.

5.3 Effect of load prediction

We compare the execution of our algorithm with and without load prediction in Figure 6. When load prediction is disabled, the algorithm simply uses the last observed load in its decision making. Figure 6 (a) shows that load prediction significantly reduces the average number of hot spots in the system during a decision run. Notably, prediction prevents over 46% hot spots in the simulation with 1400 VMs. This demonstrates its high effectiveness in preventing server overload proactively. Without prediction, the algorithm tries to consolidate a PM as soon as its load drops below the threshold. With prediction, the algorithm correctly foresees that the load of the PM will increase above the threshold shortly and hence takes no action. This leaves the PM in the "cold spot" state for a while. However, it also reduces placement churns by avoiding unnecessary migrations due to temporary load fluctuation.

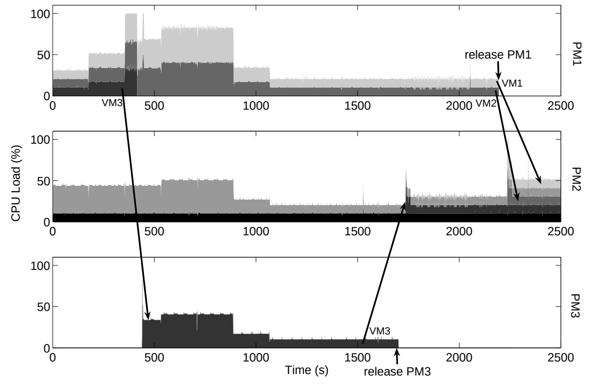


Fig. 7. Algorithm effectiveness

Consequently, the number of migrations in the system with load prediction is smaller than that without prediction as shown in Figure 6 (c). We can adjust the conservativeness of load prediction by tuning its parameters, but the current configuration largely serves our purpose (i.e., error on the side of caution). The only downside of having more cold spots in the system is that it may increase the number of APMs. This is investigated in Figure 6 (b) which shows that the average numbers of APMs remain essentially the same with or without load prediction (the difference is less than 1%). This is appealing because significant overload protection can be achieved without sacrificing resources efficiency. Figure 6 (c) compares the average number of migrations per VM in each decision with and without load prediction. It shows that each VM experiences 17% fewer migrations with load prediction.

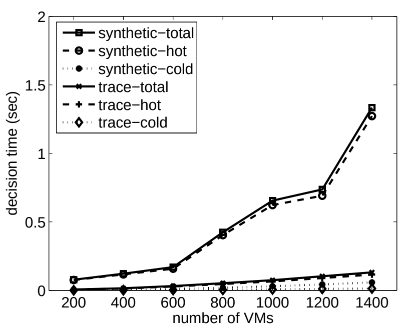
6 EXPERIMENTS

Our experiments are conducted using a group of 30 Dell PowerEdge blade servers with Intel E5620 CPU and 24GB of RAM. The servers run Xen-3.3 and Linux 2.6.18. We periodically read load statistics using the `xenstat` library (same as what `xentop` does). The servers are connected over a Gigabit ethernet to a group of four NFS storage servers where our VM Scheduler runs. We use the same default parameters as in the simulation.

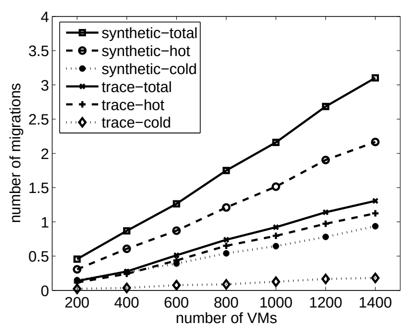
6.1 Algorithm effectiveness

We evaluate the effectiveness of our algorithm in overload mitigation and green computing. We start with a small scale experiment consisting of three PMs and five VMs so that we can present the results for all servers in figure 7. Different shades are used for each VM. All VMs are configured with 128 MB of RAM. An Apache server runs on each VM. We use httperf to invoke CPU intensive PHP scripts on the Apache server. This allows us to subject the VMs to different degrees of CPU load by adjusting the client request rates. The utilization of other resources are kept low.

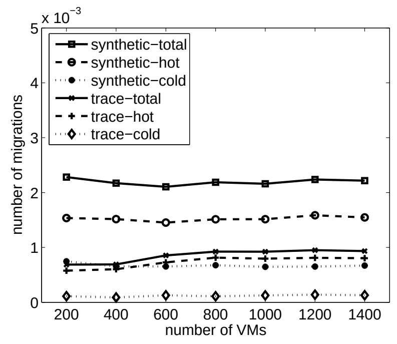
We first increase the CPU load of the three VMs on PM_1 to create an overload. Our algorithm resolves the overload by migrating VM_3 to PM_3. It reaches a stable state under high



(a) average decision time

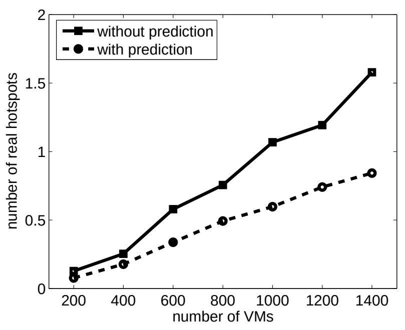


(b) average number of migrations

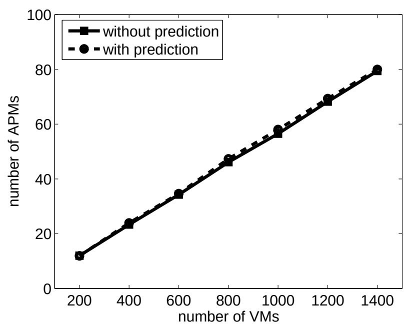


(c) number of migrations per VM

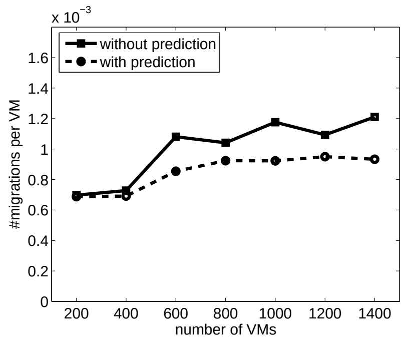
Fig. 5. Scalability of the algorithm with system size



(a) number of hot spots



(b) number of APMs



(c) number of migration

Fig. 6. Effect of load prediction

load around 420 seconds. Around 890 seconds, we decrease the CPU load of all VMs gradually. Because the FUSD prediction algorithm is conservative when the load decreases, it takes a while before green computing takes effect. Around 1700 seconds, VM_3 is migrated from PM_3 to PM_2 so that PM_3 can be put into the standby mode. Around 2200 seconds, the two VMs on PM_1 are migrated to PM_2 so that PM_1 can be released as well. As the load goes up and down, our algorithm will repeat the above process: spread over or consolidate the VMs as needed.

Next we extend the scale of the experiment to 30 servers. We use the TPC-W benchmark for this experiment. TPC-W is an industry standard benchmark for e-commerce applications which simulates the browsing and buying behaviors of customers [13]. We deploy 8 VMs on each server at the beginning. Each VM is configured with one virtual CPU and two gigabyte memory. Self-ballooning is enabled to allow the hypervisor to reclaim unused memory. Each VM runs the server side of the TPC-W benchmark corresponding to various types of the workloads: browsing, shopping, hybrid workloads, etc.. Our algorithm is invoked every 10 minutes.

Figure 8 shows how the number of APMs varies with the average number of requests to each VM over time. We keep the load on each VM low at the beginning. As a result, green computing takes effect and consolidates the VMs onto a smaller number of servers. [1] Note that each TPC-W server, even when idle, consumes several hundreds megabytes of memory. After two hours, we increase the load dramatically to emulate a "flash crowd" event. The algorithm wakes up the stand-by servers to offload the hot spot servers. The figure shows that the number of APMs increases accordingly. After the request rates peak for about one hour, we reduce the load gradually to emulate that the flash crowd is over. This triggers green computing again to consolidate the under-utilized servers. Figure 8 shows that over the course of the experiment, the number of APM rises much faster than it falls. This is due to the effect of our FUSD load prediction. The figure also shows that the number of APMs remains at a slightly elevated level after the flash crowd. This is because the TPC-W servers maintain some data in cache and hence its memory usage never goes back to its original level.

To quantify the energy saving, we measured the electric power consumption under various TPC-W workloads with the built-in watt-meter in our blade systems. We find that an idle blade server consumes about 130 Watts and a fully utilized server consumes about 205 Watts. In the above experiment, a server on average spends 48% of the time in standby mode due to green computing. This translates into roughly 62 Watts power-saving per server or 1860 Watts for the group of 30 servers used in the experiment.

6.2 Impact of live migration

One concern about the use of VM live migration is its impact on application performance. Previous studies have found this impact to be small [5]. We investigate this impact in our own

1. There is a spike on the number of APMs at the very beginning because it takes a while to deploy the 240 VMs onto 30 servers.

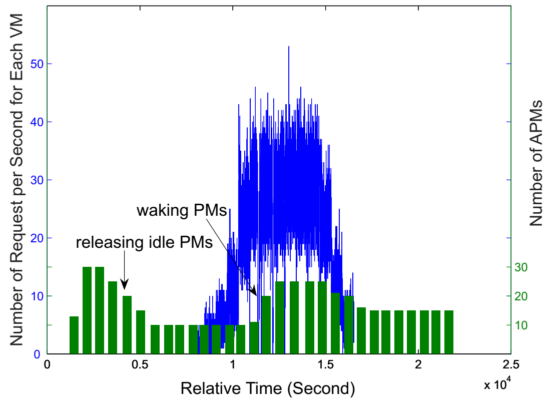


Fig. 8. #APMs varies with TPC-W load

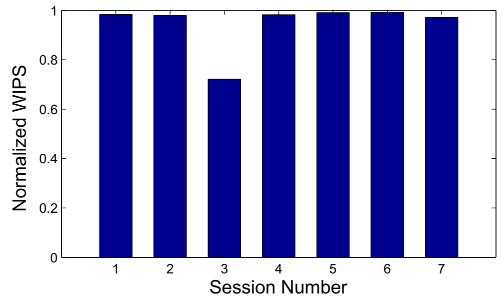


Fig. 9. Impact of live migration on TPC-W performance

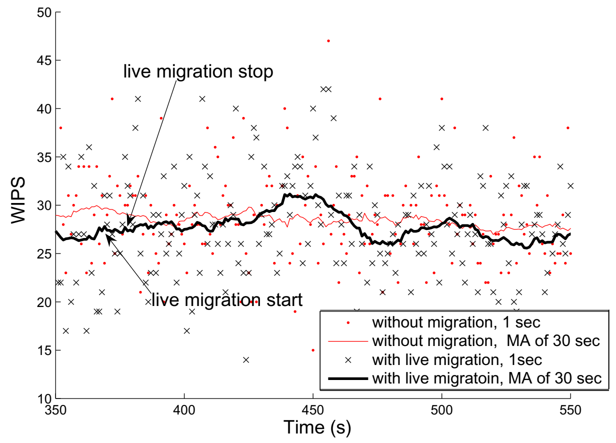


Fig. 10. TPC-W performance with and without live migration

experiment. We extract the data on the 340 live migrations in our 30 server experiment above. We find that 139 of them are for hot spot mitigation. We focus on these migrations because that is when the potential impact on application performance is the most. Among the 139 migrations, we randomly pick 7 corresponding TPC-W sessions undergoing live migration. All these sessions run the “shopping mix” workload with 200 emulated browsers. As a target for comparison, we re-run the session with the same parameters but perform no migration and use the resulting performance as the baseline. Figure 9 shows the normalized WIPS (Web Interactions Per Second) for the 7 sessions. WIPS is the performance metric used by TPC-W. The figure shows that most live migration sessions exhibit no noticeable degradation in performance compared to the baseline: the normalized WIPS is close to 1. The only exception is session 3 whose degraded performance is caused by an extremely busy server in the original experiment.

Next we take a closer look at one of the sessions in figure 9 and show how its performance vary over time in figure 10. The dots in the figure show the WIPS every second. The two curves show the moving average over a 30 second window as computed by TPC-W. We marked in the figure when live migration starts and finishes. With self-ballooning enabled, the amount of memory transferred during the migration is about 600MB. The figure verifies that live migration causes no noticeable performance degradation. The duration of the migration is under 10 seconds. Recall that our algorithm is invoked every 10 minutes.

6.3 Resource balance

Recall that the goal of the skewness algorithm is to mix workloads with different resource requirements together so that the overall utilization of server capacity is improved. In this experiment we see how our algorithm handles a mix of CPU, memory, and network intensive workloads. We vary the CPU load as before. We inject the network load by sending the VMs a series of network packets. The memory intensive applications are created by allocating memory on demand. Again we start with a small scale experiment consisting of two PMs and four VMs so that we can present the results for all servers in Figure 11. The two rows represent the two PMs. The two columns represent the CPU and network dimensions, respectively. The memory consumption is kept low for this experiment.

Initially, the two VMs on PM_1 are CPU intensive while the two VMs on PM_2 are network intensive. We increase the load of their bottleneck resources gradually. Around 500 seconds, VM_4 is migrated from PM_2 to PM_1 due to the network overload in PM_2. Then around 600 seconds, VM_1 is migrated from PM_1 to PM_2 due to the CPU overload in PM_1. Now the system reaches a stable state with a balanced resource utilization for both PMs – each with a CPU intensive VM and a network intensive VM. Later we decrease the load of all VMs gradually so that both PMs become cold spots. We can see that the two VMs on PM_1 are consolidated to PM_2 by green computing.

Next we extend the scale of the experiment to a group of 72 VMs running over 8 PMs. Half of the VMs are CPU intensive, while the other half are memory intensive. Initially, we keep the load of all VMs low and deploy all CPU intensive VMs on PM_4 and PM_5 while all memory intensive VMs on PM_6 and PM_7. Then we increase the load on all VMs gradually to make the underlying PMs hot spots. Figure 12 shows how the algorithm spreads the VMs to other PMs over time. As we can see from the figure, the algorithm balances the two types of VMs appropriately. The figure also shows that the load across the set of PMs becomes well balanced as we increase the load.

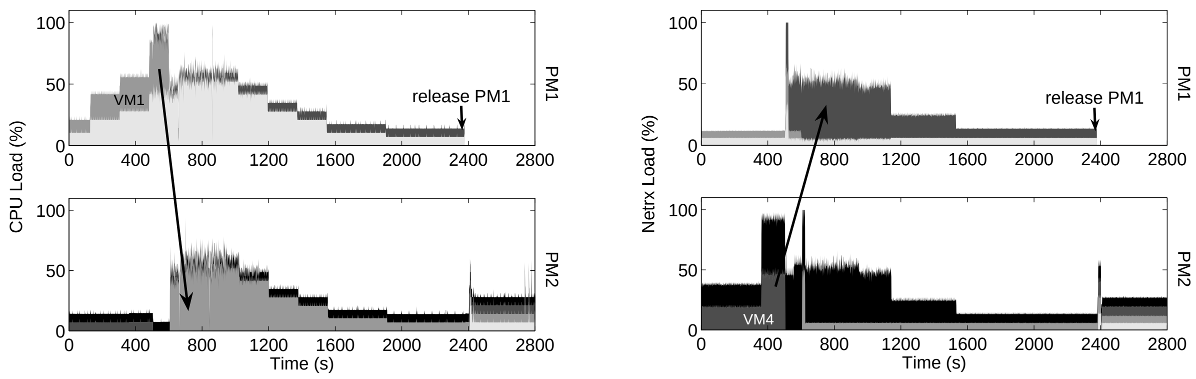


Fig. 11. Resource balance for mixed workloads

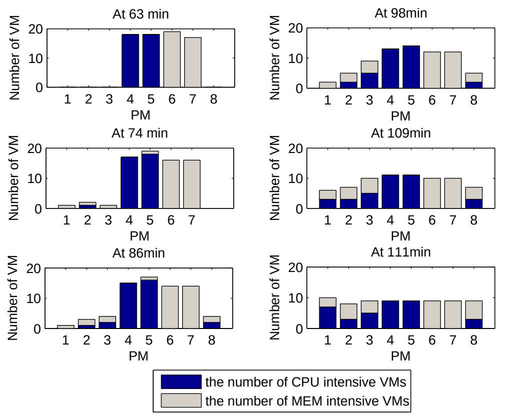


Fig. 12. VM distribution over time

7 Related work

7.1 Resource allocation at the application level

Automatic scaling of Web applications was previously studied in [14] [15] for data center environments. In MUSE [14], each server has replicas of all web applications running in the system. The dispatch algorithm in a frontend L7-switch makes sure requests are reasonably served while minimizing the number of under-utilized servers. Work [15] uses network flow algorithms to allocate the load of an application among its running instances. For connection oriented Internet services like Windows Live Messenger, work [10] presents an integrated approach for load dispatching and server provisioning. All works above do not use virtual machines and require the applications be structured in a multi-tier architecture with load balancing provided through an front-end dispatcher. In contrast, our work targets Amazon EC2-style environment where it places no restriction on what and how applications are constructed inside the VMs. A VM is treated like a blackbox. Resource management is done only at the granularity of whole VMs.

MapReduce [16] is another type of popular Cloud service where data locality is the key to its performance. Quincy adopts min-cost flow model in task scheduling to maximize data locality while keeping fairness among different jobs [17]. The “Delay Scheduling” algorithm trades execution time for data locality [18]. Work [19] assign dynamic priorities to jobs and users to facilitate resource allocation.

7.2 Resource allocation by live VM migration

VM live migration is a widely used technique for dynamic resource allocation in a virtualized environment [8] [12] [20]. Our work also belongs to this category. Sandpiper combines multi-dimensional load information into a single *Volume* metric [8]. It sorts the list of PMs based on their volumes and the VMs in each PM in their volume-to-size ratio (VSR). This unfortunately abstracts away critical information needed when making the migration decision. It then considers the PMs and the VMs in the pre-sorted order. We give a concrete example in Section 1 of the supplementary file where their algorithm selects the wrong VM to migrate away during overload and fails to mitigate the hot spot. We also compare our algorithm and theirs in real experiment. The results are analyzed in Section 5 of the supplementary file to show how they behave differently. In addition, their work has no support for green computing and differs from ours in many other aspects such as load prediction.

The HARMONY system applies virtualization technology across multiple resource layers [20]. It uses VM and data migration to mitigate hot spots not just on the servers, but also on network devices and the storage nodes as well. It introduces the *Extended Vector Product(EVP)* as an indicator of imbalance in resource utilization. Their load balancing algorithm is a variant of the Toyoda method [21] for multi-dimensional knapsack problem. Unlike our system, their system does not support green computing and load prediction is left as future work. In Section 6 of the supplementary file, we analyze the phenomenon that $VectorDot$ behaves differently compared with our work and point out the reason why our algorithm can utilize residual resources better.

Dynamic placement of virtual servers to minimize SLA violations is studied in [12]. They model it as a bin packing problem and use the well-known first-fit approximation algorithm to calculate the VM to PM layout periodically. That algorithm, however, is designed mostly for off-line use. It is likely to incur a large number of migrations when applied in on-line environment where the resource needs of VMs change dynamically.

7.3 Green Computing

Many efforts have been made to curtail energy consumption in data centers. Hardware based approaches include novel thermal design for lower cooling power, or adopting power-proportional and low-power hardware. Work [22] uses Dynamic Voltage and Frequency Scaling(DVFS) to adjust CPU power according to its load. We do not use DVFS for green computing, as explained in the Section 7 in the complementary file. PowerNap [23] resorts to new hardware technologies such as Solid State Disk(SSD) and Self-Refresh DRAM to implement rapid transition(less than 1ms) between full operation and low power state, so that it can "take a nap" in short idle intervals. When a server goes to sleep, Somniloquy [24] notifies an embedded system residing on a special designed NIC to delegate the main operating system. It gives the illusion that the server is always active.

Our work belongs to the category of pure-software low-cost solutions [10] [12] [14] [25] [26] [27]. Similar to Somniloquy [24], SleepServer [26] initiates virtual machines on a dedicated server as delegate, instead of depending on a special NIC. LiteGreen [25] does not use a delegate. Instead it migrates the desktop OS away so that the desktop can sleep. It requires that the desktop is virtualized with shared storage. Jettison [27] invents "partial VM migration", a variance of live VM migration, which only migrates away necessary working set while leaving infrequently used data behind.

8 Conclusion

We have presented the design, implementation, and evaluation of a resource management system for cloud computing services. Our system multiplexes virtual to physical resources adaptively based on the changing demand. We use the skewness metric to combine VMs with different resource characteristics appropriately so that the capacities of servers are well utilized. Our algorithm achieves both overload avoidance and green computing for systems with multi-resource constraints.

Acknowledgements

The authors would like to thank the anonymous reviewers for their invaluable feedback. This work was supported by the National Natural Science Foundation of China (Grant No. 61170056) and National Development and Reform Commission (Information Security 2011, CNGI2008-108).

References

[1] M. Armbrust *et al.*, "Above the clouds: A berkeley view of cloud computing," University of California, Berkeley, Tech. Rep., Feb 2009.

[2] L. Siegele, "Let it rise: A special report on corporate IT," in *The Economist*, Oct. 2008.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proc. of the ACM Symposium on Operating Systems Principles (SOSP'03)*, Oct. 2003.

[4] "Amazon elastic compute cloud (Amazon EC2), http://aws.amazon.com/ec2/."

[5] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proc. of the Symposium on Networked Systems Design and Implementation (NSDI'05)*, May 2005.

[6] M. Nelson, B.-H. Lim, and G. Hutchins, "Fast transparent migration for virtual machines," in *Proc. of the USENIX Annual Technical Conference*, 2005.

[7] M. McNett, D. Gupta, A. Vahdat, and G. M. Voelker, "Usher: An extensible framework for managing clusters of virtual machines," in *Proc. of the Large Installation System Administration Conference (LISA'07)*, Nov. 2007.

[8] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Black-box and gray-box strategies for virtual machine migration," in *Proc. of the Symposium on Networked Systems Design and Implementation (NSDI'07)*, Apr. 2007.

[9] C. A. Waldspurger, "Memory resource management in VMware ESX server," in *Proc. of the symposium on Operating systems design and implementation (OSDI'02)*, Aug. 2002.

[10] G. Chen, H. Wenbo, J. Liu, S. Nath, L. Rigas, L. Xiao, and F. Zhao, "Energy-aware server provisioning and load dispatching for connection-intensive internet services," in *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*, Apr. 2008.

[11] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant, "Automated control of multiple virtualized resources," in *Proc. of the ACM European conference on Computer systems (EuroSys'09)*, 2009.

[12] N. Bobroff, A. Kochut, and K. Beaty, "Dynamic placement of virtual machines for managing sla violations," in *Proc. of the IFIP/IEEE International Symposium on Integrated Network Management (IM'07)*, 2007.

[13] "TPC-W: Transaction processing performance council, http://www.tpc.org/tpcw/."

[14] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle, "Managing energy and server resources in hosting centers," in *Proc. of the ACM Symposium on Operating System Principles (SOSP'01)*, Oct. 2001.

[15] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici, "A scalable application placement controller for enterprise data centers," in *Proc. of the International World Wide Web Conference (WWW'07)*, May 2007.

[16] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in *Proc. of the Symposium on Operating Systems Design and Implementation (OSDI'08)*, 2008.

[17] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair scheduling for distributed computing clusters," in *Proc. of the ACM Symposium on Operating System Principles (SOSP'09)*, Oct. 2009.

[18] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proc. of the European conference on Computer systems (EuroSys'10)*, 2010.

[19] T. Sandholm and K. Lai, "Mapreduce optimization using regulated dynamic prioritization," in *Proc. of the international joint conference on Measurement and modeling of computer systems (SIGMETRICS'09)*, 2009.

[20] A. Singh, M. Korupolu, and D. Mohapatra, "Server-storage virtualization: integration and load balancing in data centers," in *Proc. of the ACM/IEEE conference on Supercomputing*, 2008.

[21] Y. Toyoda, "A simplified algorithm for obtaining approximate solutions to zero-one programming problems," *Management Science*, vol. 21, pp. 1417–1427, august 1975.

[22] R. Nathuji and K. Schwan, "Virtualpower: coordinated power management in virtualized enterprise systems," in *Proc. of the ACM SIGOPS symposium on Operating systems principles (SOSP'07)*, 2007.

[23] D. Meisner, B. T. Gold, and T. F. Wenisch, "Powernap: eliminating server idle power," in *Proc. of the international conference on Architectural support for programming languages and operating systems (ASPLOS'09)*, 2009.

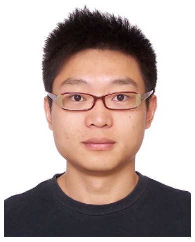
[24] Y. Agarwal, S. Hodges, R. Chandra, J. Scott, P. Bahl, and R. Gupta, "Somniloquy: augmenting network interfaces to reduce pc energy usage," in *Proc. of the USENIX symposium on Networked systems design and implementation (NSDI'09)*, 2009.

[25] T. Das, P. Padala, V. N. Padmanabhan, R. Ramjee, and K. G. Shin, "Litegreen: saving energy in networked desktops using virtualization," in *Proc. of the USENIX Annual Technical Conference*, 2010.

[26] Y. Agarwal, S. Savage, and R. Gupta, "Sleepserver: a software-only approach for reducing the energy consumption of pcs within enterprise environments," in *Proc. of the USENIX Annual Technical Conference*, 2010.

[27] N. Bila, E. d. 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 ACM European conference on Computer systems (EuroSys'12)*, 2012.

Zhen Xiao is a Professor in the Department of Computer Science at Peking University. He received his Ph.D. from Cornell University in January 2001. After that he worked as a senior technical staff member at AT&T Labs - New Jersey and then a Research Staff Member at IBM T. J. Watson Research Center. His research interests include cloud computing, virtualization, and various distributed systems issues. He is a senior member of ACM and IEEE.



Weijia Song received bachelor's degree and master's degree from Beijing Institute of Technology. He is currently a doctoral student at Peking University. His current research focuses on resource scheduling problems in cloud systems.

Qi Chen Qi Chen received bachelor's degree from Peking University in 2010. She is currently a doctoral student at Peking University. Her current research focuses on the cloud computing and parallel computing.