Virtual Batching: Request Batching for Energy Conservation in Virtualized Servers

Yefu Wang, Robert Deaver, and Xiaorui Wang
Department of Electrical Engineering and Computer Science
University of Tennessee, Knoxville, TN 37996
{*ywang38, rdeaver1, xwang*}*@eecs.utk.edu*

***Abstract*—Many power management strategies have been proposed for enterprise servers based on dynamic voltage and frequency scaling (DVFS), but those solutions cannot further reduce the energy consumption of a server when the server processor is already at the lowest DVFS level and the server utilization is still low (*e.g.*, 5% or lower). To achieve improved energy efficiency, request batching can be conducted to group received requests into batches and put the processor into sleep between the batches. However, it is challenging to perform request batching on a virtualized server because different virtual machines on the same server may have different workload intensities. Hence, putting the shared processor into sleep may severely impact the performance of all the virtual machines.**

This paper proposes Virtual Batching, a novel request batching solution for virtualized servers with primarily light workloads. Our solution dynamically allocates CPU resources such that all the virtual machines can have approximately the same performance level relative to their allowed peak values. Based on this uniform level, our solution determines the time length for periodically batching incoming requests and putting the processor into sleep. When the workload intensity changes from light to moderate, request batching is automatically switched to DVFS to increase processor frequency for performance guarantees. Empirical results based on a hardware testbed and real trace files show that Virtual Batching can achieve the desired performance with more energy conservation than several well-designed baselines, *e.g.*, 63% more, on average, than a solution based on DVFS only.

I. INTRODUCTION

The need to provide a guaranteed level of service performance is important for data centers. This is largely due to a business model driven by strict service-level agreements (SLAs) based on metrics such as response time, throughput and reserve capacity. However, energy demands and associated costs are increasing at an alarming rate; it is projected that data centers in the US alone will consume 100 billion kWh of energy at a cost of 7.4 billion dollars per year by 2011 [11]. This poses a dilemma for data center operators; they must satisfy new and existing service contracts while minimizing energy consumption in order to reduce cost and strain on power generation facilities.

Data centers generally provision based on a worst-case scenario which leads to the average server utilization in modern data centers being lower than 10% [22]. These under-utilized servers spend a large portion of their time in an idle state. Several recent studies have shown that a server uses approximately 60% of its required peak power when it is idle [4, 17]. This over-provisioning leads to large amounts of energy waste in data centers. On the other hand, average US electricity costs increased 146% between 2000 and 2008 [9]. Therefore, reducing energy waste, while guaranteeing SLA agreements, can lead to significantly reduced operating costs in data centers.

A well-known approach to addressing this problem is to transition the processor from high-power states to low-power states using Dynamic Voltage and Frequency Scaling (DVFS) whenever the performance allows. This approach effectively reduces the power consumption of the computer systems when the server has a medium intensity workload (we define workload intensity in Section II). However, the capability of this approach to reduce power consumption is limited when the server has a low-intensity workload due to two reasons. First, when the utilization of the processor is very low, the leakage power, which cannot be significantly reduced by DVFS, contributes a major portion of the power consumption. Second, many high-performance processors only allow a small range of DVFS levels and even the lowest level provides a higher speed than is required for some light workloads. For example, in a case study on our testbed, the power consumption of an idle server with an Intel Xeon 5360 processor can only be reduced from 163W to 158W when the processor is transitioned from the highest DVFS level to the lowest one.

To further reduce energy consumption, processors need to be put into sleep states such as Deep Sleep. In Deep Sleep, the processor is paused and consumes significantly less power. For example, the power consumption of an Intel processor may be reduced to 23% of its peak value when it is switched to the Deep Sleep state [14]. When the processor is in Deep Sleep, the server can be configured to use Direct Memory Access (DMA) to place incoming packets into memory buffers for processing when the processor is returned to the active state, thus avoiding harming the functionality of the hosted server applications. Therefore, to save more power (than what DVFS can do) for servers with light workloads, we can perform *request batching* to put the processor into the Deep Sleep state when there are no incoming requests. During the sleep time, we delay and batch the requests when they arrive and wake the processor up when the earliest request in the batch has been kept pending for a certain batching timeout.

However, it is challenging to perform request batching directly on a virtualized server. Virtualization technologies such as Xen, VMware, and Microsoft Hyper-V allow provisioning multiple virtual machines onto a single physical server. However, all the VMs on a single physical server are correlated due to sharing the same physical hardware, *i.e.*,

This work was supported, in part, by NSF Grants CNS-0720663, CNS-0845390, CNS-0915959, and ONR Grant N00014-09-1-0750

any state changes in the hardware affect all the VMs. Since different VMs may have different workloads and performance requirements, putting the processor into Deep Sleep based on the performance of one VM may affect the performance of other VMs.

In this paper, we propose Virtual Batching, a novel request batching solution for virtualized enterprise servers with primarily light workloads. Our solution dynamically allocates the CPU resource such that all the virtual machines can have approximately the same performance level relative to their allowed peak values. Based on the uniform level, our solution then determines the time length for periodically batching incoming requests and putting the processor into sleep. When the workload intensity changes from light to medium, request batching is automatically switched to DVFS to increase processor frequency for performance guarantees.

Specifically, this paper has the following contributions:

- We propose a novel request batching technique in virtualized environments to achieve further energy conservation when the server workload becomes light.
- We integrate request batching and DVFS to provide energy conservation for virtualized servers when the workload varies at runtime. Our solution allows energy to be saved across a wide range of workload intensities.
- We design a two-layer control architecture that relies on feedback control theory as a theoretical foundation to achieve analytical assurance of control accuracy and system stability.
- We conduct experiments on a hardware testbed with real trace files and present empirical results to demonstrate the effectiveness of our control solution to conserve energy for virtualized enterprise servers.

The rest of the paper is organized as follows. Section II introduces the overall architecture of our Virtual Batching solution. Section III presents the modeling, design, and analysis of the resource balancing controller. Section IV discusses the request batching controller. Section V describes the implementation details of our testbed. Section VI presents our empirical results. Section VII highlights the distinction of our work by discussing related work and Section VIII concludes the paper.

II. System Architecture

In this section, we give a high-level description of the Virtual Batching system architecture. The goal of Virtual Batching is to control the response times of all the virtual machines on a physical server to their administrator-defined set points while minimizing the energy consumption of the server. To achieve this goal, we adopt three techniques: request batching, DVFS, and CPU resource allocation. Figure 1 provides an overview of the system architecture.

A. Performance Balancing

Implementing a technique such as request batching or DVFS directly in a virtualized environment is challenging because both methods rely on modifying the performance of the physical system (via CPU hardware states) to manage energy consumption. Because hardware state transitions affect the performance of all virtual machines hosted on a physical server, the virtual machines are correlated. Without effective resource

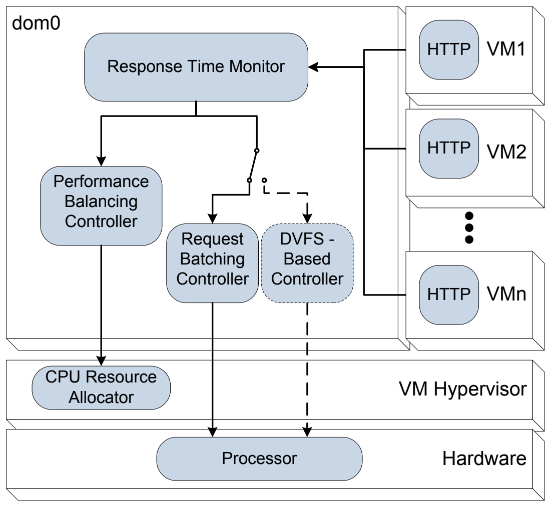


Fig. 1: Request Batching System Architecture

management, the methods of controlling the response times of enterprise servers inside the virtual machines are limited; using the average response time has a chance of allowing the busier servers to violate their response time goals while using the longest response time among the servers may waste CPU resource and energy. To overcome this limitation, our design uses a *performance balancing controller* to balance the relative response times of all the virtual machines by dynamically allocating the CPU resource. The relative response time is defined as $Rt_{measured,i}/Rt_{setpoint,i}$ where $Rt_{measured,i}$ is the measured response time of the server in the i^{th} virtual machine and $Rt_{setpoint,i}$ is the corresponding set point defined by the system administrator for the i^{th} virtual machine. This method allows the server in each virtual machine to have a different set point which gives flexibility in how virtual machines are provisioned on physical servers.

The performance balancing controller periodically uses the response time monitor to measure the average response time of the server applications in each virtual machine hosted on the server. It then adjusts the fraction of CPU time allocated to each virtual machine and gives more CPU time to virtual machines with relative response times above the average. Once the CPU allocations have been determined, they are enforced by a CPU Resource Allocator (*e.g.*, Xen's credit scheduler).

To design the performance balancing controller, we need to address the following challenges. First, the number of VMs running on the server may change at runtime. It is common that the VMs are created, terminated, or migrated out of the server, so the performance balancing controller should be able to adapt to any of these changes. Second, when the workload is light, it is possible that a VM may become completely idle for a certain interval of time such that the response time monitor cannot give a reading of the response time. To address the two challenges, we design our performance balancing controller as a collection of VM-level controllers. Every VM-level controller reads the relative response time of the VM from the VM-level response time monitor, collects the average relative response time of all non-idle VMs running on the server, and controls the VM-level relative average response time to the average relative response time. Compared with a centralized server-level controller that manages all the VMs on the server through a multi-input-multi-output (MIMO) model, in this design, the model of each VM-level controller does not need to be changed at runtime when the number of VMs on the server changes and thus can scale better.

B. Integration of Request Batching and DVFS

With a well-designed performance balancing controller, the relative response times of the individual VMs can be controlled to be close to each other. Thus, the average relative response time of the VMs indicates the server-level performance. We adopt two server-level power management techniques, request batching and DVFS, to minimize the energy consumption while guaranteeing the average relative response times of the VMs hosted on the server. Both techniques provide a compromise between the average relative response time and energy consumption. A request batching-based controller periodically controls the average relative response time by tuning the *duty cycle*, *i.e.*, the fraction of time that the processor is in the active state. The system then enforces the duty cycle by switching the processor between the sleep and active states. A longer duty cycle results in a shorter response time but higher energy consumption. Similarly, the DVFS-based controller manages the response time by tuning the DVFS level. A higher DVFS level means a shorter response time and higher energy consumption.

The two techniques are targeted for different workload intensity ranges of the server. As discussed in Section I, the request batching-based controller can be more power efficient when the workload intensity is relatively low and the DVFS-based controller works better when the workload intensity is moderate. Virtual Batching uses both controllers, dynamically switching between them. When the request batching-based controller is running, it uses the average relative response time to determine the duty cycle and drive the average relative response time to the set point. If the average relative response time is longer than the set point, the controller will increase the duty cycle to allow the processor to operate in the active state for a larger fraction of time. When the relative average response time is still higher than the set point and the duty cycle is already increased to 100%, we assume that the current intensity of the workload has become medium. We then deactivate the request batching-based controller and switch to the DVFS-based controller. The DVFS-based controller will then increase the CPU frequency to control the response time to the set point. When the DVFS-based controller is running, it tunes the CPU frequency to control the average relative response time. When the average relative response time is still shorter than the set point after the CPU frequency has been decreased to the lowest level, we assume that the current intensity of the workload is low. We then deactivate the DVFS-based controller and activate the request batching-based controller. It is important to note that light and medium workloads are *not* defined based on a preset CPU utilization point. Instead, once the request batching controller needs to switch to the DVFS controller in order to guarantee application-level performance, the workload intensity is assumed to have increased from light to medium.

In this paper, the DVFS-based controller is designed based on the control algorithm presented in our previous work [31]. The focus of this paper is on the request batching controller and the integration of request batching with DVFS to handle different workload intensities.

C. Request Batching Policy

To minimize the server energy consumption under the response time constraint, we design a request batching policy that includes two steps.

In the first step, when the server is completely idle, the processor is put into Deep Sleep until new web requests arrive. This step requires that the server should be able to automatically wake up on demand when a web request comes. These mechanisms are often available in computer systems. For example, network adapters are capable of waking up the processor from sleep modes using the support of Wake-on-LAN feature. The ACPI interface allows waking up the processor at a given point of time in the future. After the processor is waken up, the system enters the second step.

In the second step, when the server is not completely idle, it is possible that the response time is unnecessarily shorter than the desired value at the cost of more energy consumption. In this case, it is desirable that the processor be put into Deep Sleep in short periods to allow reduced energy usage. To achieve response time guarantees, the power state of the processor is switched between Deep Sleep and the active mode on a time scale much shorter than the response time requirement of a web request based on a value called *duty cycle*, *i.e.*, the fraction of time when the processor is put into Deep Sleep. For example, to enforce a duty cycle of 50%, the processor may switch between Deep Sleep and the active states periodically on a time scale of hundreds of microseconds to tens of milliseconds. When the processor is sleeping, incoming requests are batched by the network adapter. When there are no pending requests to process, the system is switched to the first step and put into Deep Sleep again. We design a request batching controller to dynamically tune the duty cycle to control the response time of the VMs to a certain set point. The details of the request batching controller are available in Section 4.

Note that though different VMs sharing the server may have different workloads, they share the same duty cycle. When the processor is in the active mode, the performance balancing controller gives more CPU resource to the VMs with heavy workloads such that all VMs will have their desired performance.

The duty cycle is enforced by switching the processor between the sleep and active states on a small time scale. Based on a recent study [18], the transition time of putting a processor into the Deep Sleep mode is about 30 μs for many processors, such as the Intel Pentium M processor. For processors that do not directly support Deep Sleep, such as Xeon 5400 series, clock gating can be used instead, which can effectively reduce the processor power from 80W to 16W in nanoseconds [18]. This small overhead makes it feasible to apply request batching in real server systems. In addition, it has been demonstrated that some other components in a server, such as DRAM DIMM, can also be transitioned into the sleep mode in less than 1 μs in future server design. As a result, the Virtual Batching technique can be extended to put those components into sleep as well.

III. Performance Balancing Controller

As discussed in Section II, a performance balancing controller is designed to balance the VM-level response times relative to their response time requirements by dynamically adjusting the CPU resource allocated to the VMs. In this section, we present the problem formulation, modeling, and design of the performance balancing controller.

A. Problem Formulation

Performance balancing control can be formulated as a dynamic optimization problem. We first introduce some notation. A physical server hosts N virtual machines. T_1 is the control period. d_i is the maximum allowed response time of i^{th} virtual machine VM_i. $rt_i(k)$ is the average response time of VM_i in the k^{th} control period. $r_i(k)$ is the relative response time of VM_i. Specifically, $r_i(k) = rt_i(k)/d_i$. $r(k)$ is the average relative response time of all the virtual machines in the k^{th} control period, namely, $r(k) = \sum_{i=1}^{N} r_i(k)/N$. The goal of the performance balancing controller is to balance the relative response time of all the VMs, *i.e.*, to make $r_i(k) = r(k)$.

To achieve this goal, we adjust a parameter in the virtual machine hypervisor, *weight* $w_i(k)$. The scheduler of the VM hypervisor allocates CPU time to VM_i proportionally to $w_i(k)$.

The performance balancing problem can be formulated as:

$$\min_{\{w_j(k)|1\le j\le N\}} \sum_{i=1}^{N} (r_i(k) - r(k))^2 \tag{1}$$

B. System Modeling and Controller Design

The response time model, *i.e.*, the relationship between $r_i(k)$ and $w_i(k)$, is normally nonlinear due to the complexity of computer systems. Since nonlinear control can lead to unacceptable runtime overhead, a standard method used in control theory to handle such systems is to linearize the system model. Therefore, we use a linear model as:

$$r_i(k) = \sum_{j=1}^{m_i} \alpha_{ij} r_i(k-j) + \sum_{j=0}^{n_i} \boldsymbol{\beta}_{ij} w_i(k-j) + \gamma_i \tag{2}$$

We conduct extensive analysis using workloads in a wide range of intensities on the testbed introduced in detail in Section V. The results show that $m_i = 1$ and $n_i = 1$ provides a reasonable compromise between model accuracy and model simplicity. Therefore, we use $m_i = 1$ and $n_i = 1$. Model (2) can be rewritten as the following difference equation:

$$r_i(k) = r_i(k-1) + \alpha_i(r_i(k-1) - r_i(k-2)) + \beta_i \Delta w_i(k-1) \tag{3}$$

where $\Delta w_i(k) = w_i(k+1) - w_i(k)$ and α_i and β_i are constant parameters whose nominal values can be decided based on least square method in an offline system identification experiment, as introduced in [29]. For each specific VM, the real values of α_i and β_i can be dynamically updated using the online model estimator proposed in [30].

Based on control theory, we design the VM-level response time controller as a P (Proportional) controller:

$$\Delta w_i(k) = K_i(r(k) - r_i(k)) \tag{4}$$

where $K_i = \frac{1}{2\beta_i}$.

In each control period, the VM-level controller measures the relative response of VM_i, $r_i(k)$, compares it with the average relative response time of all VMs, $r(k)$, and decides the change in *weight* based on the controller in (4). All of the VM-level controllers together compose the performance balancing controller which dynamically equalizes the relative response times of all the VMs.

C. Controller Analysis

In this subsection, we analyze the stability of the performance balancing controller. A fundamental benefit of the control theoretic approach is that it gives us confidence for system stability, even when the controller is used under different working conditions.

For N VMs sharing a physical server, the N VM-level controllers are coupled because they all rely on the same average response time, which is dependent on the response time of every VM. Thus, to analyze the stability of the controller, we need to consider all the VMs together.

The closed-loop behavior of the performance balancing controller can be derived by substituting (4) into (3), so we get:

$$\begin{aligned} r_i(k+1) &= r_i(k) + \alpha_i(r_i(k-1) - r_i(k-2)) \\ &\quad + \beta_i K_i\Big(\frac{1}{N}\sum_{j=1}^{N} r_j(k) - r_i(k)\Big) \end{aligned}$$

This is equivalent to:

$$D(k+1) = Q \cdot A \cdot P \cdot D(k) \tag{5}$$

where $D(k) = [r_1(k) - r_2(k), \cdots, r_{N-1}(k) - r_N(k), r_1(k-1) - r_2(k-1), \cdots, r_{N-1}(k-1) - r_N(k-1)]^T$, Q and P are constant matrices consisting of 0, 1 and -1, A is a constant matrix depending on α_i, β_i and K_i.

To demonstrate the behavior of our system, we study the stability of the close loop system in (5). We say (5) is stable if $\lim_{k\to\infty} d_i(k) = 0, (1 \le i \le N-1)$, which means $\lim_{k\to\infty} r_i(k) = \lim_{k\to\infty} r_j(k), (0 \le i, j \le N)$, *i.e.*, the relative response time of all the VMs are balanced. Based on control theory, the system in (5) is stable if and only if all the eigenvalues of $Q \cdot A \cdot P$ reside inside the unit circle.

Example: We now apply the stability analysis approach to the server used in our experiment under light workload. Based on an offline analysis, we know that $\alpha_i = 0$ when the workload is light. This is because when the workload is sparse, the server queue is less likely to build up, thus the condition in the last period is less likely to affect the response time in the current period. Thus, the matrix A in (5) is:

$$A_{ij} = \begin{cases} 1 - \beta_i K_i + \frac{1}{N}\beta_i K_i & \text{if } i = j \\ \frac{1}{N}\beta_i K_i & \text{otherwise} \end{cases} \tag{6}$$

where $K_i = 0.35$ and $\beta_i = 1.43$.

When the system does not have model variations, the eigenvalues of the matrix QAP are $\{0.5, 0.5\}$. Thus, the system is stable.

When the controller is applied to a real system, model variations are common, *i.e.*, the value of the parameter β_i in (6) may change at runtime due to changes in workloads or in the VMs. However, the parameter K_i is designed based on the system with a nominal value of β_i. Thus, it is important to evaluate the stability of the system when the controller is used with model variations. Now we analyze the stability when parameter β_i in the model changes to $g_i\beta_i$ due to the changes in workload in two different cases. In the first case, we assume that all the VMs have a uniform model variation, *i.e.*, $g_i = g, 1 \le i \le n$. We derive the range of g as $0 < g \le 2$. This means that a system controlled by the controller designed based on the estimated model will remain stable as long as the real system parameters are smaller than twice of the values used to design the controller. In the second the case, each VM has a different model variation, our second way of analyzing the system stability is to analyze one VM at a time. Our results show that the range of g_i is $0 < g_i \le 5.4$. Other model variation patterns can be analyzed in a similar way. We used a Matlab program to perform the above stability analysis procedure automatically.

IV. Request Batching Controller

In this section, we present the problem formulation, modeling, and design of the request batching controller.

A. Problem Formulation

As discussed in Section 2, the request batching policy periodically puts the processor into Deep Sleep to save power while guaranteeing the desired response time.

We denote the processor duty cycle as $\lambda(k)$, *i.e.*, $0 < \lambda(k) \le 1$. $\lambda = 1$ means that the CPU is active 100% of the time, *i.e.*, there is no sleeping time. The request batching controller controls $r(k)$, the average relative response time of all of the VMs, to a desired set point by dynamically adjusting $\lambda(k)$ to put the processor into Deep Sleep to save power based on $\lambda(k)$.

B. System Modeling and Controller Design

As discussed in Section II, in this paper we focus on controlling the server-side average response time. The server-side response time depends on the time spent by the processor in processing the web requests and the time spent in IO subsystems, etc. Therefore, we model the server-side response time as two parts: the first part depends on the active time of the processor and can be delayed when the processor is put into Deep Sleep. The second part remains relatively constant when the processor shifts between Deep Sleep and active modes. Suppose the average relative response time of the server is r_0, given a duty cycle $\lambda(k)$, the response time is modeled as:

$$r(k) = \lambda(k)r_0 + r_{IO} \tag{7}$$

This model can be rewritten as a difference equation:

$$r(k) = r(k-1) + r_0\Delta\lambda(k-1) \tag{8}$$

We design the controller as a P-controller:

$$\Delta\lambda(k) = K_{p2}(r(k) - r_s(k)) \tag{9}$$

where $K_{p2} = 1/r_0$

In every control period, the controller gets the measured relative average response time $r(k)$ from the response time monitors of all the VMs. Then it decides a new duty cycle based on the controller in (9).

We now analyze the stability of the system. The system model (8) may change at runtime due to changes in workloads, the number of VMs, etc. To model the variation in the parameter r_0, we write (8) as:

$$r(k) = r(k-1) + gr_0\Delta\lambda(k-1) \tag{10}$$

The controller is designed based on the model with $g = 1$. We have proven that the system remains stable when $0 < g \le 2$. This means that a system controlled by the controller based on the estimated model will remain stable as long as the real system parameters are smaller than twice of the values used to design the controller.

V. System Implementation

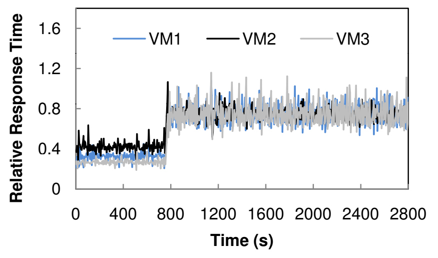
Our hardware testbed is configured with an AMD Opteron 2222 SE CPU and 4GB RAM. The hypervisor is Xen 3.1 and the Linux 2.6.18 kernel is used for the privileged/administrative domain (dom0) and three VMs (domU). Apache 2.2.3 is installed in all domU VMs and configured to serve a dynamic web page via PHP.

The performance balancing controller, request batching controller, and CPU frequency modulator are configured to run as daemons in dom0 along with a response time monitor. The response time monitor accepts incoming connections for all of the web servers. When a request is received, the following steps are conducted. First, a time stamp is recorded. Second, the HTTP request is passed to the appropriate virtual machine. Third, when the HTTP server in the virtual machine generates a response, a second time stamp is taken. Fourth, the response generated by the HTTP server in the virtual machine is returned to the requesting client. Finally, the difference between the timestamps generated in step one and step three is taken to calculate the response time for that particular request.

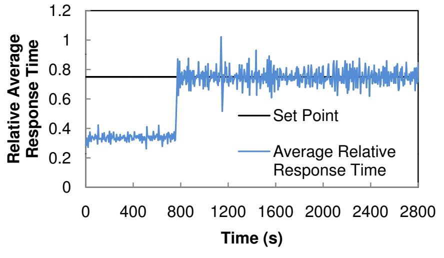
The response time monitor is also capable of replaying the web requests based on a trace file or a concurrency level. When it generates workloads based on a trace file, it sends requests based on the time stamps of recorded activities in the trace file. When it generates workloads based on a concurrency level, it dynamically issues HTTP requests to make the number of requests being processed in the server the same as the concurrency level, *i.e.*, the workload is continuous and stable. In this paper, we test our algorithms using workloads generated in both ways.

In our experiments, since Linux has well-known incompatibility issues with the ACPI implementation of the Dell servers deployed in our testbed, we use emulation to place the processor into Deep Sleep. Specifically, when the processor needs to be put into sleep, the request batching controller issues a SIGSTOP to all the apache processes on each virtual machine, waits for the appropriate time based on the current duty cycle, then issues a SIGCONT to all the apache processes. Note that the transition overheads for the server to sleep and wake up has been accounted by deducting the typical overhead (30 μs per transition [18]) from the desired sleep interval in our energy measurements. Therefore, our emulation results can provide accurate insights for the performance evaluations.

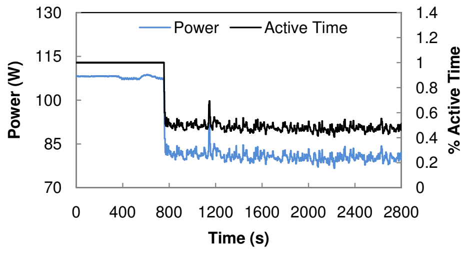
To record the energy use of the testbed, power and energy measurements are taken with a *Wattsup PRO* meter. For experiments involving request batching, logs are kept of the amount of time the CPU would have spent in Deep Sleep allowing net energy consumption to be approximated using $e_{net} = e_{gross} - t_{sleep} * (p_{awake} - p_{sleep})$, where e_{net} is the



(a) Relative response times of the three VMs.



(b) Average relative response time.



(c) Power consumption and CPU activity.

Fig. 2: A typical run of Virtual Batching. The controllers are activated at time 750s.

approximate net energy usage, e_{gross} is the measured energy usage, t_{sleep} is the total amount of time the CPU would have spent in Deep Sleep, p_{awake} is the average power used by the server during the waking periods, and p_{sleep} is the estimated power usage of the server when the CPU would be in Deep Sleep.

In every control period, the request batching controller decides a new duty cycle based on the controller equation in (9). The system then enforces the duty cycle by switching the processor between the sleep and active states on a small time scale denoted as a subinterval T_s. We use a modulator to approximate the duty cycle. For example, to enforce a duty cycle of 0.25, the modulator will place the processor into Deep Sleep for $3T_s$ and the active state for T_s.

The length of T_s needs to be carefully selected. A longer T_s means that the processor will be kept in Deep Sleep for a longer period of time, thus increases the probability that a request arriving during the sleep interval may miss the response time requirement. A shorter T_s means that the processor may be switched between the Deep Sleep and active modes more frequently, thus incurs a higher overhead. Furthermore, the resolution of the timer in the operating system limits the length of T_s. We select the length of T_s to be $min\{100T_o, T_{timer}\}$, where T_o is the time spent to switch the processor between the Deep Sleep and active states and T_{timer} is the resolution of the timer in the OS, such that the overhead of enforcing the duty cycle is less than 1%.

VI. Evaluation

The testbed described in Section V is used in three experiments to show the effectiveness of our solution. In the first experiment, we examine the performance balancing and request batching controllers. Then, we conduct the second experiment to show the effectiveness of integrating DVFS and request batching to handle both light and moderate workloads. The last experiment compares Virtual Batching against three baselines with bursty workloads.

A. *Baselines*

The first baseline, called Conservative, is a simple extension of the request batching policy for non-virtualized servers used in [10] to allow it to function in a virtualized environment. The Conservative policy places the processor into Deep Sleep whenever there is no workload for the entire physical server *i.e.*, no virtual machines are processing requests. Once new incoming traffic is detected for any web server on a network interface, the system waits for a batching timeout then wakes up the CPU to process requests. The batching timeout is determined periodically by an ad-hoc controller to drive the web server with the longest response time (no performance balancing is used) to the set point. The ad-hoc controller simply increments the batching timeout by 10ms if the response time is at or below the set point and decrements the batching timeout by 10ms if the batching timeout is above the set point. Using this method, all the virtual machines can operate at or below the set point.

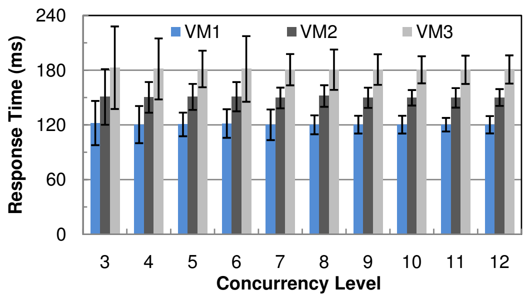
The second baseline, called Aggressive, uses a request batching controller for each virtual machine to determine batching timeouts and when it is safe for the controller's virtual machine to be put into sleep. A master controller is used to place the processor into Deep Sleep when more than half of the virtual machine controllers determine it is safe to sleep. When new traffic is detected after the master controller has placed the processor is in Deep Sleep, the system waits for the batching timeout associated with the virtual machine that is serving the traffic, then wakes up the CPU to respond to the requests.

The third baseline, called DVFS-Only, only runs the DVFS-based method [31] to save energy without putting the processor into Deep Sleep.

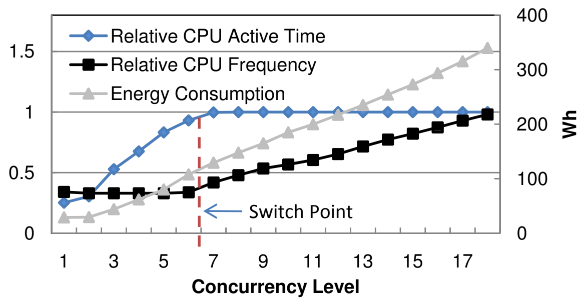
B. *Request Batching*

In this experiment, the web server in each virtual machine is subjected to a light and constant workload to demonstrate the functionality of the performance balancing and request batching controllers. The controllers are activated at time 750s. Figure 2(a) shows that the *relative* response times, defined in Section II, of all the three virtual machines are controlled to be approximately equal (all three Web servers had an equal relative response time goal) after the controller is activated at time 750s. To balance the web server response times, the performance balancing controller manipulates the scheduling weight of each VM. Here the scheduling weight is based on Xen's credit scheduler [1]. Xen allots CPU time to each virtual machine based on this weight; a VM with a weight of 512 will get twice as much CPU time as a VM with a weight of 256. The performance balancing controller calculates a new weight for each VM in every control period and the new weights are given to the credit scheduler for enforcement.

The request batching controller is used to control the response time of the web servers in the virtual machines to the desired set point. Figure 2(b) plots the average of the relative response times shown in Figure 2(a) along with the administrator-defined set point of 0.75. This figure shows that the average response time has been controlled to the set point



(a) VM response times under different workload intensities.



(b) CPU frequency and system energy consumption under different workload intensities. The dashed line shows the switch point between request batching and DVFS.

Fig. 3: Integration of request batching and DVFS.

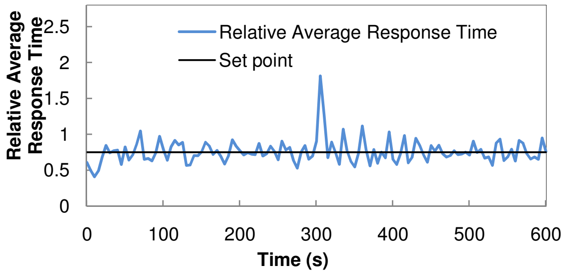


Fig. 4: Request batching switching to DVFS at time 300s during a typical run.

after time 750s. Figure 2(c) shows the power consumption and the percentage of time the processor would spend in Deep Sleep during each control period. These results show that the Virtual Batching can effectively divide the resources of a single physical server among multiple virtual machines and control the response times of the web servers in each virtual machine to the desired set point.

C. *Integration of Request Batching and DVFS*

The second experiment shows that the system is capable of controlling each virtual machine to a specific, possibly different, set point across a broad range of workloads. This experiment subjects the server to workloads ranging from light to heavy by manipulating the number of concurrent requests addressed to each web server throughout the experiment.

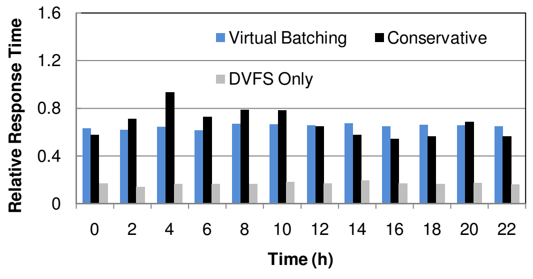
Figure 3(a) shows the relative response time of each web server for each virtual machine across different concurrency levels. The response times are controlled very closely to their set points. Note that in this experiment, each virtual machine has a different set point; VM1's set point is 120ms, VM2's set point is 150ms, and VM3's set point is 180ms.

Figure 3(b) plots the percentage of time the processor is active and its relative frequency under different levels of concurrent requests, as well as the energy consumption on the secondary axis. As expected, the processor spends very little time in the active state when the concurrency level is one to three: 25% to 50% respectively, and spends a larger percentage of time being active after the concurrency level increases to six. When the concurrency level is smaller than seven, the request batching controller is used to manage response time. When the concurrency level reaches seven, the CPU is active 100% of the time, which shows that the system has automatically switched to the DVFS-based controller to manage the response time. The switch point is denoted as the dashed line in Figure 3(b). For concurrency levels one through six, the processor is at its lowest frequency. This is the range for which the request batching controller is activated. After the concurrency levels increase to seven, the DVFS-based controller begins to manage response time and increases the CPU frequency to keep the response times of the web servers at their desired set points.

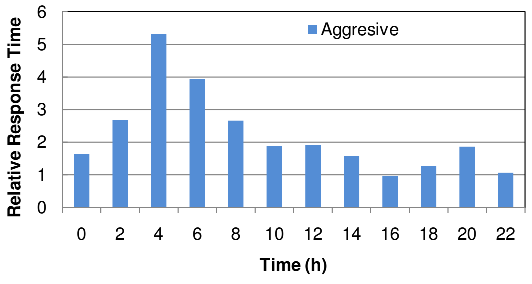
The energy consumption of the system increases with the concurrency level from a minimum of 28.9Wh at concurrency level one to a maximum of 339.8Wh at concurrency level eighteen. The points of note for energy usage, however, are for concurrency levels one through six; for these concurrency levels, the energy consumption is markedly lower than what could be achieved with DVFS alone. At concurrency level seven, the energy consumption is 129.2Wh which is the minimum energy usage available for a DVFS only solution; compared to the minimum energy consumption of 28.9Wh, request batching provides an additional energy savings of 77.6%. This set of experiments demonstrates that request batching is an effective way for further energy conservation when the workload is light, a situation commonly observed in today's data centers.

Figure 4 shows the average of the response times of all the virtual machines in a typical run, in which the controller switches from request batching to DVFS. From time zero to 300s the workload is very low (at the concurrency level of 3 in this example). At time 300s the workload is increased heavily from the concurrency level of 3 to 23 for VM2, leading to the spike shown in the figure. Once the increase in workload is detected by the system, it manipulates the duty cycle to give more processor time to the VMs until the duty cycle reaches 1, *i.e.*, the processor is active 100% of the time. After that, the request batching controller is switched to the DVFS-based controller to increase the CPU frequency to drive the response time back to the set point. Despite the controller switch, the average response time has been controlled to the desired set point most of the time.

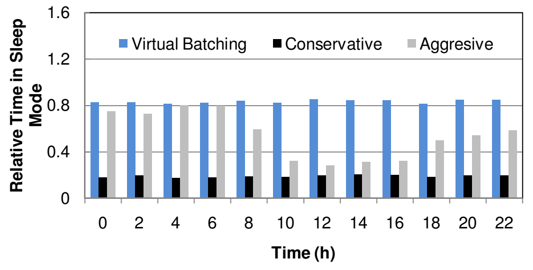
These results show that Virtual Batching is able to use a combination of request batching and DVFS to control the response time of web servers in the virtual machines under different workload intensities.



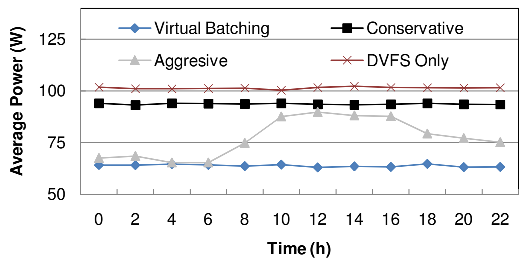
(a) Average relative response time



(b) Average relative response time - Aggressive



(c) Time in Deep Sleep



(d) Power Usage

Fig. 5: Comparison with baselines for bursty workloads based on a real 24-hour trace file.

D. Comparison with Baselines for Bursty Workloads

In this experiment, we test our system using trace files from several real-world servers [2]. We use three trace files SDSC-HTTP, EPA-HTTP and ClarkNet-HTTP from three different HTTP servers to generate workloads in an entire day (24 hours starting from 00:00 AM) for the three VMs running on our server, and compare our solution to three baselines, Conservative, Aggressive, and DVFS-Only. The characteristics of the trace files are available in [2].

Figure 5(a) plots the average relative response time, with a set point of 0.8, every 2 hours in the execution. The response time of Conservative is the closest to that of Virtual Batching due to its ability to batch requests, albeit with an ad-hoc controller. DVFS-Only has a substantially lower relative response time because it does not batch requests. Conservative sometimes has longer average response times than Virtual Batching because it cannot adjust CPU resource allocations between VMs as it does not have a performance balancing controller. When the VMs face different workload patterns in different time periods of a day, some VMs may have very long response times, making the average response time long.

The response time of Aggressive is shown in Figure 5(b) because it is significantly longer than those of other solutions. Aggressive strives to minimize power consumption, but does so at the expense of response time violation. At four hours into the experiment, the relative response time peaks at 5.3. With a set point of 0.8, this is a huge performance violation and would be unacceptable in a real-world application.

Figure 5(c) displays the relative amount of time the CPU spends in Deep Sleep for Virtual Batching, Conservative, and Aggressive. Virtual Batching remains active for approximately 20% throughout the day while Aggressive starts high and keeps the processor busier throughout the morning and afternoon then backs off as traffic decreases for the evening. This shows that Aggressive is not as efficient at handling the workload fluctuations. Under Conservative, the processor is active approximately 80% throughout the day, which leads to a higher power consumption than Virtual Batching.

Figure 5(d) shows the average power comparison. Aggressive is the closest contender to Virtual Batching with regard to power usage, but does so at the expense of response time and shows excessive response time violations shown in Figure 5(b). Conservative uses considerably more power than Virtual Batching, approximately 93.7W on average which translates to a large amount of energy waste. This is expected because Conservative focuses on managing response time over energy usage. DVFS-Only shows the highest power consumption of all, averaging 101.4W. This is because once the processor is in its lowest DVFS level, no more power can be saved by DVFS-Only. Virtual Batching achieves an average power savings of 63% over DVFS-Only which translates into real-world energy conservation.

VII. Related Work

Virtualization technology has provided a promising way to manage application performance by dynamically reallocating resources to VMs. Several management algorithms have been proposed to control application performance for virtualized applications [7, 21, 22, 32]. For example, Padala et al. [21] used MIMO control theory to control throughputs for virtualized servers. In contrast, our algorithm controls response time, a user-perceived performance metric. In addition, our solution provides energy savings through DVFS and request batching.

Energy conservation has been one of the most important design constraints for Internet servers. The majority of the prior work has attempted to reduce power consumption by reducing the energy consumption of individual server components [16]. Several algorithms have been proposed to utilize DVFS to manage the energy consumption at the server level [17], the cluster level [5, 12, 19, 24, 27], and the data center level [28]. Recently, several research projects have addressed power or energy problems in platforms employing virtulization technology by using DVFS techniques [20, 31]. However, these algorithms cannot further reduce energy consumption when the processor is already at the lowest DVFS level.

Elnozahy et al. [10] proposed a request batching technique for non-virtualized web servers by putting the processor into the sleep mode when the server is idle, significantly reducing the energy consumption of web servers when the workload is light. In contrast, our solution provides additional energy savings by allowing the processor to sleep even when the server is not completely idle. Further, we provide a performance balancing controller to dynamically adjust the CPU resource allocation to the VMs such that request batching can be utilized in virtualized environments.

VM migration is an important tool for resource and power management in virtualized computing environments. Several recent studies propose to dynamically consolidate VMs to a smaller number of servers and putting the unused servers into the sleep mode for energy savings [15, 23, 25, 26]. In contrast to these algorithms, our solution has several advantages. First, the request batching technique used in our solution has a much less overhead than VM migration, thus can be used on a much smaller time scale. Second, due to resource limitations, such as the memory and bandwidth, sometimes it is infeasible to further consolidate VMs to a smaller number of servers even if the utilization of the processor is still low. In this situation, request batching can save additional energy by putting the processor to the sleep mode periodically. Our algorithm can be integrated with VM consolidation algorithms to save power in different situations.

Several papers have explored using low power states of server components [6, 31]. AbouGhazaleh [3] and Choi et al. [8] provided studies based on DVFS. Horvath and Skadron [13] proposed using dynamic cluster configuration and multiple ACPI sleep states to reduce energy consumption in server muilti-tier clusters. Zhu et al. [33] and Nathuji et al. [19] proposed giving fewer resources to the operating system or VM. None of these, however, consider using a batching technique.

VIII. Conclusions

In order to save more energy (than what DVFS can) for servers with light workloads, request batching can be conducted to group received requests into batches and putting the processor into Deep Sleep between the batches. However, it is challenging to perform request batching on a virtualized server because the performance of all the virtual machines on a physical server is correlated due to sharing the same hardware platform. In this paper, we have proposed Virtual Batching, a novel request batching solution for virtualized web servers with primarily light workloads. Virtual Batching dynamically allocates the CPU resource such that all the virtual machines can have approximately the same performance level relative to their allowed peak values. Based on the uniform level, Virtual Batching then determines the time length for periodically batching incoming requests and putting the processor into Deep Sleep. When the workload intensity changes from light to medium, request batching is automatically switched to DVFS to increase processor frequency for performance guarantees. Empirical results based on a hardware testbed and real trace files show that Virtual Batching can achieve the desired performance with more energy conservation than several well-designed baselines, e.g., 63% more, on average, than a solution based on DVFS only.

References

[1] Credit Scheduler. http://wiki.xensource.com/xenwiki/CreditScheduler.

[2] Traces available in the Internet Traffic Archive. http://ita.ee.lbl.gov/html/traces.html.

[3] N. AbouGhazaleh, A. Ferreira, C. Rusu, R. Xu, F. Liberato, B. Childers, D. Mosse, and R. Melhem. Integrated cpu and l2 cache voltage scaling using machine learning. *SIGPLAN Notices*, 42:41–50, 2007.

[4] L. Barroso and U. Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40:33–37, 2007.

[5] L. Bertini, J. Leite, and D. Mosse. SISO PIDF controller in an energy-efficient multi-tier web server cluster for e-commerce. In *FeBID*, 2007.

[6] E. V. Carrera, E. Pinheiro, and R. Bianchini. Conserving disk energy in network servers. In *ICS*, 2003.

[7] Y. Chen, S. Iyer, X. Liu, D. Milojicic, and A. Sahai. SLA decomposition: Translating service level objectives to system level thresholds. In *ICAC*, 2007.

[8] K. Choi, W. Lee, R. Soma, and M. Pedram. Dynamic voltage and frequency scaling under a precise energy model considering variable and fixed components of the system power dissipation. In *ICCAD*, 2004.

[9] DOE. Electric power monthly, October 2009.

[10] M. Elnozahy, M. Kistler, and R. Rajamony. Energy conservation policies for web servers. In *USENIX*, 2003.

[11] EPA. Report to congress on server and data center energy efficiency. Technical report, US Environmental Protection Agency, 2007.

[12] J. Heo, D. Henriksson, X. Liu, and T. Abdelzaher. Integrating adaptive components: An emerging challenge in performance-adaptive systems and a server farm case-study. In *RTSS*, 2007.

[13] T. Horvath and K. Skadron. Multi-mode energy management for multi-tier server clusters. In *PACT*, 2008.

[14] Intel. Intel 5500 series datasheet vol. 1. 2009.

[15] G. Khanna, K. Beaty, G. Kar, and A. Kochut. Application performance management in virtualized server environments. In *NOMS*, 2006.

[16] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller. Energy management for commercial servers. *IEEE Computer*, 36, 2003.

[17] C. Lefurgy, X. Wang, and M. Ware. Server-level power control. In *ICAC*, 2007.

[18] D. Meisner, B. T. Gold, and T. F. Wenisch. Powernap: Eliminating server idle power. In *ASPLOS*, 2009.

[19] R. Nathuji, P. England, P. Sharma, and A. Singh. Feedback driven qos-aware power budgeting for virtualized servers. In *FeBID*, 2009.

[20] R. Nathuji and K. Schwan. Virtualpower: Coordinated power management in virtualized enterprise systems. In *SOSP*, 2007.

[21] 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 *EuroSys*, 2009.

[22] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *EuroSys*, 2007.

[23] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu. No power struggles: Coordinated multi-level power management for the data center. In *ASPLOS*, 2008.

[24] K. Rajamani and C. Lefurgy. On evaluating request-distribution schemes for saving energy in server clusters. In *ISPASS*, 2003.

[25] A. Verma, P. Ahuja, and A. Neogi. pmapper: Power and migration cost aware application placement in virtualized systems. In *Middleware*, 2008.

[26] A. Verma, G. Dasgupta, T. K. Nayak, P. De, and R. Kothari. Server workload analysis for power minimization using consolidation. In *USENIX*, 2009.

[27] X. Wang and M. Chen. Cluster-level feedback power control for performance optimization. In *HPCA*, 2008.

[28] X. Wang, M. Chen, C. Lefurgy, and T. W. Keller. SHIP: Scalable hierarchical power control for large-scale data centers. In *PACT*, 2009.

[29] X. Wang and Y. Wang. Co-con: Coordinated control of power and application performance for virtualized server clusters. In *IWQoS*, 2009.

[30] Y. Wang, K. Ma, and X. Wang. Temperature-constrained power control for chip multiprocessors with online model estimation. In *ISCA*, 2009.

[31] Y. Wang, X. Wang, M. Chen, and X. Zhu. Power-efficient response time guarantees for virtualized enterprise servers. In *RTSS*, 2008.

[32] J. Xu, M. Zhao, J. Fortes, R. Carpenter, and M. Yousif. On the use of fuzzy modeling in virtualized data center management. In *ICAC*, 2007.

[33] X. Zhu, D. Young, B. J. Watson, J. Rolia, S. Singhal, B. Mckee, C. Hyser, D. Gmach, R. Gardner, T. Christian, and L. Cherkasova. 1000 islands: An integrated approach to resource management for virtualized data centers. *Cluster Computing*, 12:45–47, 2009.