Energy-Efficient Real-Time Heterogeneous Server Clusters [*]

Cosmin Rusu, Alexandre Ferreira, Claudio Scordino[†] Aaron Watson,
Rami Melhem and Daniel Mossé
Department of Computer Science, University of Pittsburgh

Abstract

With increasing costs of energy consumption and cooling, power management in server clusters has become an increasingly important design issue. Current clusters for real-time applications are designed to handle peak loads, where all servers are fully utilized. In practice, peak load conditions rarely happen and clusters are most of the time underutilized. This creates the opportunity for using slower frequencies, and thus smaller energy consumption, with little or no impact on the Quality of Service (QoS), for example, performance and timeliness.

In this work we present a cluster-wide QoS-aware technique that dynamically reconfigures the cluster to reduce energy consumption during periods of reduced load. Moreover, we also investigate the effects of local QoS-aware power management using Dynamic Voltage Scaling (DVS). Since most real-world clusters consist of machines of different kind (in terms of both performance and energy consumption) we focus on heterogeneous clusters.

For validation, we describe and evaluate an implementation of the proposed scheme using the Apache *Webserver in a small realistic cluster. Our experimental results show that using our scheme it is possible to save up to 45% of the total energy consumed by the servers, maintaining average response times within the specified deadlines and number of dropped requests within the required amount.*

1 Introduction

Until recently, performance had been the main concern in server farms, but energy consumption has also become a main concern in such systems. Due to the importance of customer care service in commercial installations and the importance of timely responses for embedded server clusters, current clusters are typically designed to handle peak loads. However, peak load conditions rarely happen in practice, and clusters are most of the time underutilized. In fact, their loads often vary significantly depending on the time of the day or other external factors, therefore the average processor use of such systems may be even less than 50% with respect to their peak capacity [7].

Clusters with high peak power need complex and expensive cooling infrastructures to ensure the proper operation of the servers. With power densities increasing due to increasing performance demands and tighter packing, proper cooling becomes even more challenging: fans driving the cooling system may consume up to 50% of the total system power in some commercial servers [16, 18], and manufacturers are facing the problem of building powerful systems without introducing additional techniques such as liquid cooling. Electricity cost is a significant fraction of the operation cost of data centers [6]. For example, a Google 10kW rack consumes about 10MWh a month (including cooling), which is at least 10% of the operation cost [5], with this fraction likely to increase in the future.

These issues are even more critical in embedded clusters [28], typically untethered devices, in which peak power has an important impact on the size of the system, while energy consumption determines the device lifetime. Examples include satellite systems or other mobile devices with multiple computing platforms, such as the Mars Rover and robotics platforms.

Power management (PM) mechanisms can be divided into two categories: *cluster-wide* and *local* [6]. Cluster-wide mechanisms involve global decisions, such as turning on and off cluster machines, according to the load. Local techniques put unused (or underutilized) resources in low-power states, for example self-refresh, standby and off modes for DRAM chips, Dynamic Voltage Scaling (DVS) and low-power states for the CPU, disk shutdown, etc. A PM mechanism (local or cluster-wide) is *QoS-aware* if it reduces the power consumption while guaranteeing a certain amount of Quality of Service (QoS), such as average response times or percentage of deadlines met.

To the best of our knowledge, this is the first Power Management scheme that is *simultaneously* (a) cluster-wide

[*]Supported by NFS through the Secure-CITI project (ANI-0325353) and the Power-Autonomous Networks project (ANI-0121658).

[†]Claudio Scordino is a PhD student at the University of Pisa, visiting the University of Pittsburgh.

(i.e., turning on and off machines), (b) designed for heterogeneity, (c) QoS-aware and power-aware at the local servers (i.e., deadline-aware), (d) measurement-based (contrary to theoretical modeling, relying on measurements is the key to our approach), (e) implementation-oriented, and (f) performing reconfiguration decisions at runtime.

Our scheme is realistic because most clusters have one or more front-ends, are composed of different kind of machines, and need both local and cluster-wide QoS-Aware PM schemes. While the methodology and the algorithms proposed apply to any kind of cluster, we show their use in a web server context. Our measurements show a reduction of energy consumption equal to 17% using only the local PM, 39% using the On/Off scheme, and 45% using both schemes. With respect to delays, the local PM added $0.5ms$, while On/Off added about $4ms$; in all cases, the average delay was quite small with respect to deadlines.

The remainder of the paper is organized as follows. We first present related work in Section 2. The cluster model is given in Section 3. The cluster-wide PM scheme is explained in Section 4, while the local real-time DVS scheme is presented in Section 5. Both schemes are then evaluated in Section 6. In Section 7 we state our conclusions.

2 Related Work

With energy consumption emerging as a key aspect of cluster computing, much recent research has focused on PM in server farms. A first characterization of power consumption and workload in real-world webservers was made in [7]. DVS was proposed as the main technique to reduce energy consumption in such systems. DVS and request batching techniques were further evaluated in [10]. Software peak power control techniques were investigated in [11]. However, these studies considered only power consumption of processor and main memory in single-processor systems.

The problem of *cluster configuration* (i.e., turning on and off cluster machines) for homogeneous clusters was first addressed in [20]. An offline algorithm determines the number of servers needed for a given load. Cluster reconfiguration is then performed online by a process running on a server, using thresholds to prevent too frequent reconfigurations, even though there is no explicit QoS consideration. The authors have extended their work to heterogeneous clusters in [14]. Models have been added for throughput and power consumption estimation. Reconfiguration decisions are made online based on the precomputed information and the predicted load. The authors also proposed to add request types to improve load estimation in [15].

Our work differs from the above previous studies in the following ways: we consider QoS directly; individual servers are both power-aware and QoS-aware; we rely on offline measurements instead of using models; reconfiguration decisions (i.e., number of active servers and load distribution) are not expensive and are performed online; and reconfiguration thresholds are based on the time needed to boot/shutdown a server.

One of the first attempts to combine cluster-wide and local PM techniques [9] proposed five different policies combining DVS and cluster configuration. However, the theory behind this work relies on (a) homogeneous clusters, and cannot be easily extended to heterogeneous machines; and (b) the often-incorrect assumption that power is a cubic function of the CPU frequency. Another work proposed to use the cluster load (instead of the average CPU frequency) as the criteria for turning on/off machines [28]. However, this study assumed homogeneous clusters as well. To the best of our knowledge, this work is the first attempt to combine cluster-wide and local techniques in the context of heterogeneous clusters.

In real-time computing, dynamic voltage (and frequency) scaling has been explored to reduce energy consumption. DVS schemes typically focus on minimizing CPU energy consumption while meeting a performance requirement [29]. DVS work for aperiodic tasks in single processors includes: offline and online algorithms assuming worst-case execution times [29, 24], automatic DVS for Linux with distinction between background and interactive jobs [12], and use of knowledge about the distribution of job lengths for voltage scaling decisions [17, 21]. However, these techniques typically aim at reducing the energy consumed only by the CPU [17, 22, 24, 23] and do not take into account other devices (such as memory, power supplies, or disk) that contribute with an important fraction to the total energy consumed by the system. In our model, instead, servers can put their resources in low-power states, and no assumption is made about their local PM schemes.

Most related to our *local scheme* is Sharma et al.'s work on adaptive algorithms for DVS for a QoS-enabled web server [26]. Their scheme uses a theoretical utilization bound derived in [3] to guarantee the QoS of web requests. However, they take into account only local PM, assuming that a good load balancing algorithm is used at the front-end. In that sense, our works are complementary, since we describe how to achieve such load balancing.

3 Cluster Model

This section introduces the cluster model that we consider (see Figure 1). A front-end machine receives requests from clients and *redirects* them to a set of processing nodes, henceforth referred to as *servers*. The front-end is not a processing node and has three main functions: (a) accepting aperiodic requests from clients, (b) distributing the load to servers, and (c) reconfiguring the cluster (i.e., turning servers on/off) to reduce the global energy consumption while keeping the overall performance within a prespecified QoS requirement. After receiving a request, the front-end communicates to the client to which server the request must be sent using *HTTP redirection* [8]. Then, the client sends its request directly to the server.

In our cluster scheme, each request is an aperiodic task (i.e., no assumptions are made about task arrival times) and is assigned a deadline. The specification of the QoS is system-wide and is, in our case, the percentage of deadlines met. The way to achieve the soft-real-time properties will be presented in detail in the next sections.

Each server in the heterogeneous cluster performs the same service (i.e., all servers can process all requests). No restriction is imposed regarding any aspect of their computation: process scheduling, CPU performance, memory speed/bandwidth, disk speed/bandwidth, power consumption, network bandwidth, etc. In addition, servers periodically inform the front-end about their current load, to aid the front-end in load distribution and cluster configuration decisions. After a request has been processed by a server, the result is returned directly to the client, without the front-end as intermediary.

Note that a more common cluster design is with the front-end acting as a proxy (i.e., acting as intermediary between clients and servers). In our webserver example, choosing one configuration or the other (i.e., proxy versus no proxy with redirection) is simply a configuration option, and the proposed scheme in this paper works equally well with either type of front-end. In our experiments, for high loads (above 1Gbps), we had to use the no-proxy architecture shown in Figure 1, as a proxy front-end cannot fully utilize the cluster in our experimental setup (our front-end has only one GbE network interface card).

When using redirection instead of proxying, the links and internal references should use the full URL to guarantee that all the requests are sent to the front-end. This way, redirection works with either the *HTTP/1.0* or the *HTTP/1.1* protocols. In *HTTP/1.1* the client may keep multiple connections open to the front-end and the servers it was redirected to, but all the requests will be sent to the front-end first.

The aspects related to cluster configuration, PM and load distribution performed by the front-end will be presented in detail in the next section. Local PM is performed independently by each server, without front-end control, and will be presented in Section 5.

4 Front-end Power Management

Our proposed front-end follows a very general framework that is applicable to *any* heterogeneous cluster. To achieve this goal, we cannot impose any restriction on server characteristics. However, for ease of presentation, definitions and examples emphasize web server clusters.

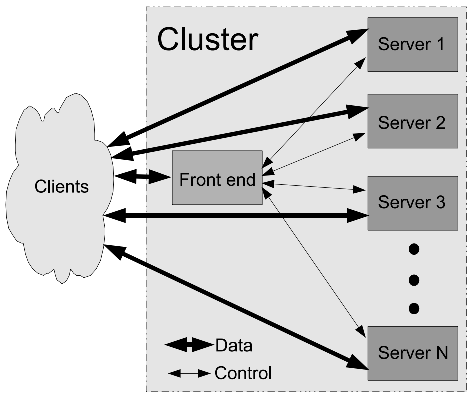

Figure 1. Cluster architecture

4.1 Load Definition and Estimation

The front-end determines the number of active servers to meet the desired level of QoS while minimizing cluster energy consumption. The number of servers is computed (offline or online) as a function of the system load. Thus, defining load correctly is a crucial step. A measure of the load for clusters is the number of requests received per second, measured over some recent interval. Clearly, depending on the kind of service under consideration, other definitions of load may be more appropriate (such as the bandwidth for a file server).

At runtime, the front-end needs to correctly estimate (or observe) the load, in order to make PM decisions and to perform load distribution. The load estimation can be further improved by using feedback from the servers.

As observed in previous work [21, 28, 15], load estimation can be greatly improved by considering request types. The type of a request may be conveniently determined only by the header (e.g., the name of the requested file). Notice that the number of types is a design issue. On one hand, different types may not be necessary (if the variability of the time to service a request is low). On the other hand, each request could be of a different type, leading to an improved estimation but also to a higher overhead (to measure all different types of requests and update statistics tables).

In the case of a web server there are two main types of requests, with different computational characteristics: static and dynamic pages. Static pages reside in server's memory and do not require much computation. Dynamic pages, instead, are created on-demand through the use of some external language (e.g., Perl or PHP). For this reason, dynamic pages typically require more computation than static ones.

Consider a generic server, and let A_{static} and $A_{dynamic}$

be the average execution times to serve a static and a dynamic page, respectively, at the maximum CPU speed. For example, for one server in our cluster we measured an average execution time $A_{static} = 438\mu s$ for static pages and $A_{dynamic} = 24.5ms$ for dynamic pages. On average, the time needed by the CPU to serve N_{static} static requests and $N_{dynamic}$ dynamic requests is thus $N_{static}A_{static} + N_{dynamic}A_{dynamic}$ seconds. If the number of requests is observed over a period of $monitor\_period$ seconds, then the load of the machine serving the requests is

$$Load = \frac{N_{static}A_{static} + N_{dynamic}A_{dynamic}}{monitor\_period} \quad (1)$$

Notice that this definition of load assumes a CPU-bound server. This is normal for most web servers because much of the data are already in memory [7, 27]. In fact, on all our machines we have noticed that the bottleneck of the system was the CPU. However, for systems with different bottlenecks (e.g., disk I/O or network bandwidth) another definition of load may be more appropriate. In fact, the definition of load should account for the bottleneck resource. Note that even though web requests may exhibit a large variation in execution times, using the average values (A_{static} and $A_{dynamic}$) in Equation 1 results in a very accurate load estimation. This is because web requests are relatively small and numerous.

We define the *maximum load* of a server as the maximum number of requests that it can handle meeting the 95% of deadlines. The front-end never directs more than the maximum load to a server. The cluster load is defined as the sum of the current loads of all active servers. Therefore, the maximum load that the cluster can handle is the sum of the maximum loads of all servers. At runtime, the cluster load (i.e., both variables N_{static} and $N_{dynamic}$) is observed every $monitor\_period$ seconds. The value of $monitor\_period$ is a design issue, related to the tradeoff between response time and overhead. In our cluster, values in the order of a few seconds were found suitable.

4.2 Server information

In order to reduce the global power consumption at runtime, we furnish the front-end with information about the average power consumption of each server for any different value of its load. Servers can reduce their own power consumption in a number of different ways, such as using DVS and low-power states for the CPU, self-refresh modes for memory, stopping disk spinning after some time of idleness, etc. Moreover, each server may use a different OS or a different scheduling policy (such as a standard round robin, or a real-time policy to give higher priority to static pages with respect to dynamic ones). No assumption is made at the front-end about local PM schemes.

Once the local PM scheme, the OS, and the scheduling policy have been decided for a server, the power consumption as function of the load and the maximum load can be determined through simple measurements.

In our experiments, after choosing the local PM scheme (see Section 5), we measured the average power consumption for a load in 5% increments. Then, we interpolated the points to have values in 1% increments. We measured the total power consumed by the whole machines, not only by their CPUs. In our case recording the average power consumption for a given load over a period of 10 minutes was sufficient to obtain a good average. We measured AC power directly, with a simple power meter with 2% accuracy [25]. Hence, the whole process required at most few hours for each machine. Clearly, identical machines need not to be measured twice. The curve representing the power consumption of each server of our cluster is shown in Figure 2. The last point on each curve represents the maximum load that meets our QoS specification (i.e., 95% of deadlines met), normalized to the fastest machine in the cluster. The parameters for each machine are reported in Table 2 (on Page 9).

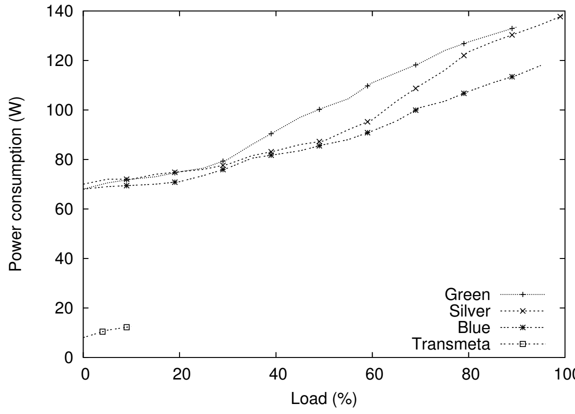

Figure 2. Power consumption vs. load for servers

The information about power consumption of servers can then be used by the front-end at runtime. Since the front-end controls the load for each server, and the power consumption of each server for a given load is known, the front-end has enough information for PM decisions. Notice that the cooling costs of the room have not been taken into account. However, since these costs are expected to be proportional to the AC power drawn, they are automatically reduced by minimizing cluster power consumption.

We now present all the information (and the corresponding notation) about each server needed at the front-end level. $boot\_time_i$ and $shutdown\_time_i$ represent the time to boot and to shutdown server i, including the time to start and finish the (webserver) process of the server. max_load_i is the maximum load of server i that can satisfy the 95% QoS requirement. off_power_i is the power consumed when the server is off (some components, such as the Wake-On-Lan interface used to power up the machine, may not be completely off). Finally, $power\_vs\_load_i$ is an array with $\lceil \frac{max\_load_i}{load\_increment} \rceil$ entries recording the measured power consumption of server i for each value of the load in $load\_increment$ percents (we used 1%). The first entry of the array denotes the idle power (i.e., no load).

4.3 On/Off Policy

This section describes the key idea behind our cluster-wide PM scheme. The front-end, besides distributing the load to servers to minimize global power consumption, determines the cluster configuration by turning on/off servers. Below is the algorithm used by the front-end to decide which servers will be turned on/off.

The algorithm turns machines on and off in a specific order, which is based on the power efficiency of servers (i.e., the integral of power consumption versus load). In our case, according to the values of Figure 2, the ordering for our cluster is: *Transmeta*, *Blue*, *Silver*, *Green*. In some situations we may need to change the order at runtime, as explained later.

The front-end turns on servers as the cluster load increases. However, since the boot time is not negligible, we need to turn machines on *before* they are actually needed. For this reason, the front-end maintains a variable, called $max\_load\_increase$, which specifies the maximum load variation that the cluster is prepared to sustain during $monitor\_period$. This is essentially the maximum slope of the load characterization for the cluster.

The on/off policy relies on two tables computed offline. The first table, called $mandatory\_servers$, keeps the load at which to turn servers on and is used to determine the lowest number of servers needed at a certain load to satisfy the QoS requirement. For example, consider a cluster with three servers having maximum loads $max\_load_0 = 0.5$, $max\_load_1 = 1.5$ and $max\_load_2 = 1.0$, respectively. Suppose that $monitor\_period$ is 5 seconds, $max\_load\_increase$ is equal to 0.05, and the boot time is 10 seconds for every machine. Ideally, we need only one server when the cluster load is less than 0.5, two servers when load is between 0.5 and 2, and all servers when load is higher than 2. However, if we account for the time to boot a new machine and we suppose that the cluster load is checked periodically every $monitor\_period$ seconds, the table becomes $mandatory\_servers = \{0, 0.35, 1.85\}$. Thus, the first server is always on, whereas the second and third servers are turned on when the cluster load reaches 0.35 and 1.85, respectively. In fact, if we consider the boot time of a new server, we have to account for a potential load increase equal to $\frac{boot\_time}{monitor\_period} max\_load\_increase$. Moreover, if we suppose that the load is checked periodically every $monitor\_period$ seconds, we have to introduce an additional interval of time to account for the error when measuring the current load. In general, server i is turned on when the cluster load reaches

$$\sum_{j=0}^{i-1} max\_load_j - (\frac{boot\_time_i}{monitor\_period} + 1) max\_load\_increase$$

The second table, called $power\_servers$, precomputes the number of servers needed to minimize the power consumption for a given load. Unlike the previous table, this table is computed considering the power consumption of servers, and is used to distribute the current load among active servers. For a given value of N, we compute the power consumption of the cluster as follows. We start considering a load equal to zero, and we increase its value in $load\_increment$ increments. For any increment of the load, we evaluate which server can handle it in order to minimize the overall energy consumption.

To determine the load at which N servers become more power efficient than $N-1$, we follow this procedure considering both cases of $N-1$ and N machines, respectively. The load at which N servers consume less power than $N-1$ servers is the value after which the N^{th} server is turned on. The server to be turned on is the next one according to the power efficiency order.

The complexity of computing the two tables is $O(N)$ (where N is the number of servers) for $mandatory\_servers$ and $O(NM)$ for $power\_servers$, where $M = \sum_{i=1}^{N} \lceil \frac{max\_load_i}{load\_increment} \rceil$. In our cluster, the time to compute these two tables was less than $1msec$, which was negligible compared to $monitor\_period$ (that is in the range of seconds). Thus, this computation can also be performed online. For example, a new ordering of the servers and an online recalculation of the tables become necessary when a server crashes.

A high-level view of the front-end on/off policy is presented in Figure 3. Every $monitor\_period$ seconds the load is estimated according to Equation 1, then the request counters are reset. The number of mandatory servers $N_{mandatory}$ is determined by a lookup in the $mandatory\_servers$ table. If $N_{mandatory}$ is higher than the current number of active servers $N_{current}$, all needed servers are immediately turned on.

Each server can be in one of the following states: Off, Boot, On, or Shutdown. After receiving the "boot" command (such as a Wake-On-Lan packet), the server i moves from the Off to the Boot state. It stays in this state for $boot\_time_i$ seconds (i.e., until it starts the server process), then informs the front-end that it is available for processing, moving to the On state. When server i is shutdown, it stays in the Shutdown state for $shutdown\_time_i$ seconds, after that the front-end changes its state to Off.

The variable Cmd in Figure 3 can have three different values: None, Boot or Shutdown. This variable allows to

```
1 Every $monitor\_period$ seconds
    1.1 Compute the load according to Equation 1
    1.2 Reset the counters:
        $N_{static} = 0 \quad N_{dynamic} = 0$
    1.3 Compute the minimum number of servers that can
        handle the load:
        $N_{mandatory}$=mandatory_servers($Load$)
    1.4 if ($N_{mandatory} > N_{current}$)
        turn on the servers
        set $N_{current} = N_{mandatory}$
        return
    1.5 Compute the number of servers needed to reduce
        the energy consumption:
        $N_{power} = power\_servers(Load)$
    1.6 if ($N_{power} > N_{current}$) and ($Cmd \neq$ Boot)
        Set $Cmd$ = Boot
        Find the next server $i$ to boot
        Set $N_{current} = N_{current} + 1$
        return
    1.7 if ($N_{power} < N_{current}$) and ($Cmd \neq$ Shutdown)
        Set $Cmd$ = Shutdown
        Find the next server $i$ to shutdown
        Set $N_{current} = N_{current} - 1$
        return
2 If $Cmd$=Boot for a period of time equal to $time\_boot_i$
    2.1 Turn on server $i$
    2.2 Set $Cmd$ = None
    2.3 return
3 If $Cmd$=Shutdown for a period of time equal to
  $time\_boot_i + time\_shutdown_i$
    3.1 Turn off server $i$
    3.2 Set $Cmd$ = None
    3.3 return
```

Figure 3. On/Off policy

describe the use of thresholds when turning on/off servers. If no server is in transition (i.e., all servers are in the On or Off states) a server may be turned on or off, as decided after a lookup in the $power\_servers$ table. To be conservative, only one server at a time is turned on or off. Server i is turned off if the system is in state Cmd = Shutdown for at least $time\_shutdown_i + time\_boot_i$ consecutive seconds, which is the rent-to-own threshold (see Step 3, Figure 3). Similarly, server i is turned on if Cmd = Boot for $time\_boot_i$ consecutive seconds (see Step 2, Figure 3). Notice that these thresholds do not apply to the mandatory servers, which are started immediately. The running time of the online part of the algorithm (every $monitor\_period$ seconds) is negligible because it is in the microsecond range; the complexity is $O(N)$, but can be improved to $O(1)$ by increasing the table size from N to M (that is, storing all entries in an array).

For convex and linear power functions, tables $mandatory\_servers$ and $power\_servers$ contain the optimal transition points (in the discrete space; for continuous space, see [23]). In practice, however, power functions may have concave regions. This means that a server with an abrupt power increase at some load x may not be allocated more than x load, even though the power may become flat above $x + \epsilon$, making it a good target for load allocation. A simple fix to the problem is to consider the average power consumption over a larger interval, rather than the exact value at each load. This effectively results in smoothing the power functions. In our case, although the measured power functions have concave regions, we have found that no smoothing was necessary.

4.4 Request Distribution Policy

The front-end distributes the incoming requests to a subset of the current servers that are in the On state. $load\_allocation$ is a table containing the estimated load allocated to each server and is computed with the same procedure used to determine the $power\_servers$ table, in $O(MN)$ time. The load allocation is computed every $monitor\_period$ seconds, after the on/off decisions.

Another table, called $load\_accumulated$, stores the accumulated load of each server, and is reset after computing $load\_allocation$. The server i with the minimum weight

$$w_i = \frac{load\_accumulated_i}{load\_allocation_i} \qquad (2)$$

gets the next request. Notice that w_i can be higher than 1 when the load is underestimated. The server that receives the request updates its accumulated load (and thus increases its weight), by adding $A_{static}/monitor\_period$ or $A_{dynamic}/monitor\_period$, depending on the request type. The complexity to find the server with minimum weight is $O(N)$ with a straightforward implementation, but can be improved to $O(logN)$ using a tree.

4.5 Implementation Issues

We implemented our PM scheme in the *Apache* 1.3.33 Web server [4]. We created an Apache module, called *mod_on_off*, which makes on/off decisions. Moreover, we extended an existing module, *mod_backhand* [2], to support our distribution policy.

mod_backhand is a module responsible for load distribution in Apache clusters. It allows servers to exchange information about their current usage of resources. It also provides a set of *candidacy* functions to forward requests from an overloaded server to other less utilized servers. Examples of such functions are *byLoad*, which selects as candidate the least loaded server, and *byCost*, which considers a cost for each request.

We added a new candidacy function, called *byEnergy*, to implement our request distribution policy. Notice that only front-end machines use this function. In addition, servers provide some feedback about their current real-time utilization (as explained in Section 5) to front-ends. We used this feedback to prevent the overloading of the servers. In particular, the server with the minimum w_i is selected, providing that it is not overloaded.

The *mod_on_off* module communicates with *mod_backhand* through shared memory. On initialization, *mod_on_off* acquires server information and computes both $mandatory\_servers$ and $power\_servers$ tables. *mod_on_off* executes periodically every $monitor\_period$ seconds. On each invocation it performs the following tasks: (a) computes the current load based on the counters N_{static} and $N_{dynamic}$ (that are incremented in the "Apache *post read* request" phase), (b) looks up in the table to determine the number of servers needed for the next period, (c) computes the $load\_allocation$ table for the active servers (not shown in Figure 3), (d) turns on (by sending Wake-On-Lan packets) and off (by invoking special CGI scripts) servers, and finally (e) resets the counters N_{static}, $N_{dynamic}$ and $load\_accumulated$. In addition, it displays at runtime the estimated power and energy consumption of each server, based on the $power\_vs\_load$ and $load\_accumulated$ tables.

5 Server Power Management

In addition to front-end directed cluster reconfigurations (i.e., turning on/off machines), the servers perform their own local PM to reduce power consumption of unutilized or underutilized resources. We present an example of a QoS-aware DVS scheme and we discuss an implementation using the Apache Webserver [4].

5.1 Local DVS Policy

We rely on a local real-time scheme, where each request is an aperiodic task and is assigned a deadline. Each request *type* [21, 28, 15] has a deadline to allow for more accurate load estimation.

We consider a *soft* real-time system, in which the schedule is not generated by a real-time scheduler and the computation time C_i is the average execution time (i.e., A_{static} or $A_{dynamic}$), not the worst-case. Let D_i be the time remaining to the deadline, then the real-time utilization of a server is defined as $U = \sum_i \frac{C_i}{D_i}$.

If the CPU is the bottleneck of the system (as in our case), the CPU frequency to handle this rate of requests is $U f_{max}$, where f_{max} is the highest possible frequency of the CPU. Each server periodically computes its utilization U and sets the CPU frequency to the closest value higher than $U f_{max}$.

Note that DVS architectures may have inefficient operating frequencies [22], which exist when there are higher frequencies that consume less energy. A simple online tool for inefficient frequency elimination has been provided in [19]. Removal of inefficient operating frequencies is the first step in any DVS scheme. This was not necessary in our servers, because surprisingly all frequencies were efficient, although we had a different experience with other systems we tested [28].

5.2 Implementation Issues

We implemented an Apache module, called *mod_cpufreq*, responsible for CPU speed settings at the user level. On Athlon machines, the CPU speed was changed by writing to the `/sys/` file system, using the *CPUfreq* interface [1]. On the Transmeta machine the speed was changed by writing to a model-specific register (MSR). Since the register cannot be written from user-level we added two system calls for setting and reading its value [13]. After detecting the available frequencies, our module creates an Apache process that periodically sets the CPU frequency according to the current value of U. We chose as period $10ms$ to match *any* default Linux kernel; the measured overhead for changing voltage/frequency in the Athlon64 machines is approximately $50\mu s$.

To compute U, the module needs to know the type (i.e., static or dynamic) and the arrival time of each request. At every request arrival (called "Apache *post-read* request" phase), the arrival time and the deadline are recorded with μs accuracy and stored in a hash table in shared memory. The request type is determined from the name of the requested file. Thus, a single queue traversal is necessary to compute U. In fact, the current value of U depends on all queued requests, therefore the complexity is $O(R)$ where R is the number of requests queued; the overhead is negligible. Requests are removed from the queue after being served (called "Apache *logging* request" phase).

A problem we encountered during the implementation was that our scheme worked very well except for fast machines serving a large amount of small static pages. In this case, those machines were not increasing their speed, resulting in a large number of dropped requests. A further investigation revealed that the value of U was close to zero. We did not see this phenomenon on slower machines (such as Transmeta) nor using bigger pages. The problem was that the requests were served too fast (in approximately 150 μs). Such short requests were queued, served, and removed from the queue before other requests were added to the queue. Thus, at any time only a few requests (usually just one) was in the queue, and when *mod_cpufreq* recomputed the utilization, it resulted in an underestimation of U. In other words, even though the requests were received and queued at the OS-level, Apache was not able to see them because it is a user-level server and it has no information about requests stored at the OS level. We called this problem the "*short request overload problem*" phenomenon.

A simple fix was to compute the utilization also over a recent interval of time $interval\_size$ (we used $200ms$):

$$U_{recent} = \frac{(N_{static}A_{static} + N_{dynamic}A_{dynamic})}{interval\_size}$$

We would like to keep the server utilization U_{recent} below a certain threshold (we used $threshold = 80\%$). The minimum frequency that does that is $\frac{U_{recent}}{threshold} f_{max}$. Thus, our module sets the CPU speed to $max(U, \frac{U_{recent}}{threshold}) f_{max}$. Note that Sharma et al.'s work with a kernel webserver (kHTTPd [26]) aware of small requests at the OS-level has a nice synergy with our approach and could be used in lieu of our scheme. Exploring the composition of our cluster configuration and Sharma's (or other similar DVS) work is left for future work. The problem with including such work in our scheme is exactly the reason why the authors discontinued the development of kHTTPd: the difficulty of maintaining, developing and debugging a kernel-level server.

6 Evaluation

To evaluate our QoS-aware PM scheme we used a small cluster composed by one front-end and 4 different servers. Every machine ran *Gentoo Linux 2.6* as operating system and *Apache* 1.3.33 servers. The parameters of the machines are shown in Tables 1 and 2.

The cluster has been tested using 2 clients connected to the cluster with a GbE interface and Gbps switch; the clients generate up to 3,186 requests per second, which corresponds to a total maximum cluster load equal to 2.95 (all loads were normalized to that of *Silver* machine). A total cluster load of 0.05 (or 5%) corresponds on average to 54 requests/second. Considering request types, however, greatly improves the prediction, as 54 requests/second may correspond to a load ranging from 0.02 (if $N_{dynamic} = 0$) to 1.32 (if $N_{static} = 0$). We assigned deadlines of $50ms$ and $200ms$ for requests of static and dynamic pages.

We set $max\_load\_increase = 0.005$, therefore we had $mandatory\_servers = \{0.000, 0.062, 1.012, 2.012\}$ and $power\_servers = \{0.000, 0.100, 1.050, 2.040\}$.

6.1 DVS policy

As first experiment, we evaluated the effectiveness of our local DVS scheme. We compared our *mod_cpufreq* module with the default PM in Linux (i.e., HALT instruction when idle) and with Sharma's DVS scheme for QoS-aware web servers proposed in [26] (which we implemented at user level in our *mod_cpufreq* module). This scheme adjusts the speed of the processor to the minimum speed that maintains a quantity called *synthetic utilization* below the theoretical utilization bound ($U_{bound} = 58.6\%$) that ensures that all deadlines are met [3].

Transmeta

Frequency (MHz)	333	400	533	667	733
Idle (W)	8	8.5	8.5	9	9
Busy (W)	9	9.5	10.5	12	12.5

Blue

Frequency (MHz)	800	1800	2000	2200
Idle (W)	68	73	76	80.5
Busy (W)	74.5	93.5	105.5	120.5

Silver

Frequency (MHz)	1000	1800	2000	2200	2400
Idle (W)	70	74.5	78.5	83.5	89.5
Busy (W)	80.5	92.5	103.5	119.5	140.5

Green

Frequency (MHz)	1000	1800	2000
Idle (W)	68	79	87
Busy (W)	77	108	131

Table 1. Idle/busy power consumption (in Watts) for each server at each frequency

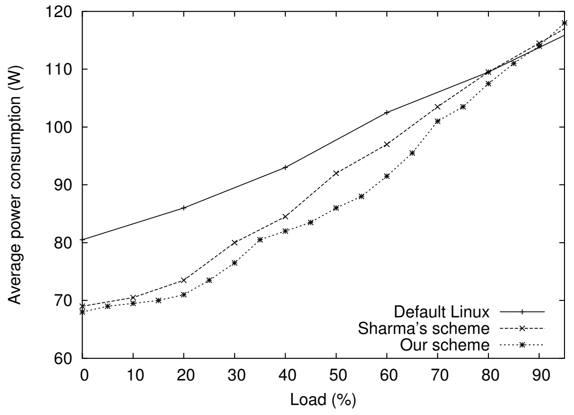

Figure 4. Comparison of DVS policies

The measured power consumption of each scheme on the *Blue* machine is shown as function of the load in Figure 4. The graph shows that our scheme outperforms the other schemes, especially for the mid-range load values. Higher savings are obtained on machines with a more convex power function (the power function of the *Blue* machine is rather linear — see Figure 2). In fact, for a rate of 300 requests/sec (approximately 28% load) the average processor frequency is 1.25GHz using our scheme and 1.5GHz using Sharma's scheme, but the amount of energy saved is only 3%. Importantly, we observed that both schemes maintained the QoS level above 99% even at the highest load.

Machine name	Processor model	RAM memory size	Cache size	Wake-On-Lan support	Boot time (sec)	Shutdown time (sec)	Off power (W)	Max load
Transmeta	Transmeta Crusoe TM5800	256 MB	512 KB		100	60	1	0.10
Blue	AMD Athlon 64 Mobile 3400+	1GB	1 MB	√	33	11	8	0.95
Silver	AMD Athlon 64 3400+	1GB	512 KB	√	33	12	8	1.00
Green	AMD Athlon 64 3000+	1GB	512 KB	√	33	11	8	0.90
Front-end	AMD Athlon 64 Mobile 3400+	1GB	1 MB	Not applicable				

Table 2. Parameters of the machines of the cluster

6.2 Overall scheme

To evaluate the overall scheme, we performed many experiments with and without the cluster-wide PM scheme (On/Off scheme), and with and without the local PM scheme (DVS scheme). For each load value, we measured the power consumption of the entire machine (not only CPU) for each scheme independently (see Figure 5). For fairness, we used the load balancing policy in Section 4.4 for all the schemes.

The On/Off policy allows a striking reduction of the energy consumption for low values of the load, because (obviously!) it allows to turn off unutilized servers. In Figure 5 we can see that when load = 0, the cluster consumption is around 32W because each Athlon server consumes 8W when in the Off state, and the Transmeta also consumes 8W when in the On state. The DVS technique, instead, has its biggest impact whenever a new server is turned on, since not all active servers are fully utilized. However, its importance decreases as the utilization of the active servers increases. For high values of the load (in our case, at 70% or higher) all servers are on, therefore the On/Off technique does not allow to reduce energy consumption. In those situations, however, there is still room for the DVS technique, that becomes more important than the On/Off technique.

The energy consumption of all servers without any power management scheme was $1.32KWh$. On average, we measured energy savings of 17% using DVS, 39% using On/Off, and 45% using both schemes.

It is worth noting that the front-end estimation of the total energy consumed when using DVS was extremely accurate: the difference from the actual values was less than 1%. For example, when using the on-off scheme, the measured value was $0.72KWh$, while the front-end estimated value was $0.725KWh$ (the resolution of our power/energy meter [25] is $0.01KWh$).

To measure the impact of cluster-wide and local PM schemes in the loss of QoS, we ran many four-hour experiments with workloads derived from actual webserver traces, and generated with the same shape of statistics taken from our *cs.pitt.edu* domain (see Table 3). The average delay (observed at the client side) without any PM scheme was $8.29ms$; a small response time is due to all machines being on at all times, and running at maximum frequency. Adding DVS (local PM) had a very small impact on the delay, with the average delay measured at $8.77ms$. However, with On/Off scheme, we measured an average delay equal to $12.29ms$ without DVS and $12.83ms$ with DVS. In both cases, the average delay was not higher than 50% of the no-PM delay and was quite small with respect to deadlines.

Request type	%	Request type	%
4 ms (CGI)	0.10	6-7 KB (html)	2.84
7 ms (CGI)	0.71	7-8 KB (html)	1.58
23 ms (CGI)	0.98	8-9 KB (html)	1.80
40 ms (CGI)	0.23	9-10 KB (html)	1.87
200 ms (CGI)	0.06	**10-20 KB (html)**	**10.74**
0-1 KB (html)	**37.78**	20-30 KB (html)	3.62
1-2 KB (html)	8.86	30-40 KB (html)	1.17
2-3 KB (html)	6.56	40-50 KB (html)	0.67
3-4 KB (html)	4.58	50-60 KB (html)	0.80
4-5 KB (html)	4.94	60-70 KB (html)	1.46
5-6 KB (html)	3.38	above 70 KB (html)	5.27

Table 3. Web server statistics: percentage of accesses, approximate size (for static pages), and running time (for dynamic pages)

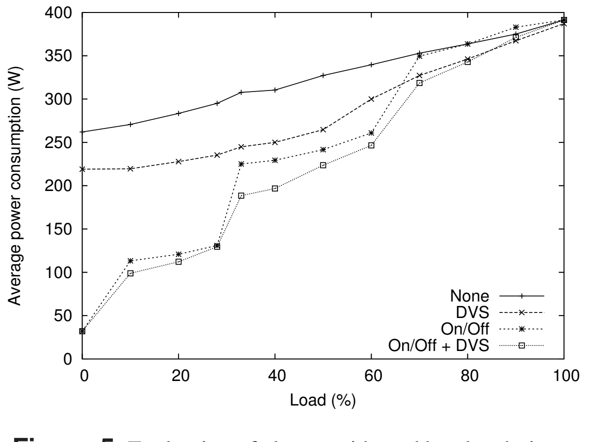

Figure 5. Evaluation of cluster-wide and local techniques

7 Conclusions and Future Work

We have presented a new QoS-aware power management scheme that combines cluster-wide (On/Off) and local (DVS) power management techniques in the context of heterogeneous clusters. We have also described and evaluated an implementation of the proposed scheme using the *Apache* Webserver in a small realistic cluster.

Our experimental results show that: (a) our load estimation is very accurate; (b) the On/Off policy allows a striking reduction of the power consumption; (c) DVS is very important whenever a new server is turned on or, as shown before, when all servers are on; (d) as expected, for high values of the load the On/Off technique does not help to reduce energy consumption but there is still room for DVS.

Using both techniques we saved up to 45% of the total energy with a limited loss in terms of QoS. In the worst case, the average delay was increased by at most 50%, and was still very small when compared to the deadlines.

As immediate future work we plan to investigate the use of both suspend-to-disk and suspend-to-RAM techniques to reduce the time to boot and shutdown a server. We also plan an integration of our cluster PM schemes with other grid-like or cluster (e.g., Condor) load balancing schemes.

References

[1] Linux kernel CPUfreq subsystem. `http://www.kernel.org/pub/linux/utils/kernel/cpufreq/cpufreq.html`.

[2] The Backhand Project. `http://www.backhand.org/`.

[3] T. F. Abdelzaher and C. Lu. Schedulability Analysis and Utilization Bounds for Highly Scalable Real-Time Services. In *Proceedings of the 7th IEEE Real-Time Technology and Applications Symposium (RTAS'01)*, Taiwan, June 2001.

[4] Apache. HTTP Server Project. `http://httpd.apache.org/`.

[5] L. A. Barroso, J. Dean, and U. Holzle. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, 23(2):22–28, 2003.

[6] R. Bianchini and R. Rajamony. Power and Energy Management for Server Systems. *Computer*, 37(11):68–74, 2004.

[7] P. Bohrer, E. N. Elnozahy, T. Keller, M. Kistler, C. Lefurgy, C. McDowell, and R. Rajamony. *The case for power management in web servers*. Kluwer Academic Publishers, 2002.

[8] V. Cardellini, M. Colajanni, and P. S. Yu. Redirection Algorithms for Load Sharing in Distributed Web-server Systems. In *19th IEEE International Conference on Distributed Computing Systems (ICDCS'99)*, Austin, TX, June 1999.

[9] M. Elnozahy, M. Kistler, and R. Rajamony. Energy-Efficient Server Clusters. In *Workshop on Power-Aware Computer Systems (PACS'02)*, 2002.

[10] M. Elnozahy, M. Kistler, and R. Rajamony. Energy Conservation Policies for Web Servers. In *4th USENIX Symposium on Internet Technologies and Systems*, Seattle, Mar. 2003.

[11] W. Felter, K. Rajamani, T. Keller, and C. Rusu. A performance-conserving approach for reducing peak power consumption in server systems. In *International Conference on Supercomputing (ICS'05)*, pages 293–302, Cambridge, Massachusetts, June 2005.

[12] K. Flautner and T. Mudge. Vertigo: Automatic Performance-Setting for Linux. In *5th Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[13] S. Gleason. Power Aware Operating Systems: Task Specific CPU Throttling. `http://www.cs.pitt.edu/PARTS/implementation/`.

[14] T. Heath, B. Diniz, E. V. Carrera, W. M. Jr., and R. Bianchini. Self-Configuring Heterogeneous Server Clusters. In *Workshop on Compilers and Operating Systems for Low Power (COLP'03)*, September 2003.

[15] T. Heath, B. Diniz, E. V. Carrera, W. M. Jr., and R. Bianchini. Energy Conservation in Heterogeneous Server Clusters. In *10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2005.

[16] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller. Energy management for commercial servers. *IEEE Computer*, 36(12):39–48, Dec. 2003.

[17] J. Lorch and A. Smith. Improving Dynamic Voltage Scaling Algorithms with PACE. In *ACM SIGMETRICS*, June 2001.

[18] J. Moore, R. Sharma, R. Shih, J. Chase, C. Patel, and P. Ranganathan. Going Beyond CPUs: The Potential of Temperature-Aware Data Center Architectures. In 1st *Workshop on Temperature-Aware Computer Systems*, 2004.

[19] PARTS. Power Efficiency Test. `http://www.cs.pitt.edu/PARTS/demos/efficient`.

[20] E. Pinheiro, R. Bianchini, E. V. Carrera, and T. Heath. Load Balancing and Unbalancing for Power and Performance in Cluster-Based Systems. In *Workshop on Compilers and Operating Systems for Low Power (COLP'01)*, Sept 2001.

[21] C. Rusu, R. Xu, R. Melhem, and D. Mossé. Energy-Efficient Policies for Request-Driven Soft Real-Time Systems. In *Euromicro Conference on Real-Time Systems (ECRTS'04)*, Catania, Italy, July 2004.

[22] S. Saewong and R. Rajkumar. Practical Voltage-Scaling for Fixed-Priority RT-Systems. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'03)*, May 2003.

[23] C. Scordino and E. Bini. Optimal Speed Assignment for Probabilistic Execution Times. In 2nd *Workshop on Power-Aware Real-Time Computing (PARC'05)*, NJ, Sept. 2005.

[24] C. Scordino and G. Lipari. Using resource reservation techniques for power-aware scheduling. In *4th ACM International Conference on Embedded Software*, Pisa, Italy, 2004.

[25] Seasonic. Power Angel. `http://www.seasonicusa.com/products.php?lineId=8`.

[26] V. Sharma, A. Thomas, T. Abdelzaher, K. Skadron, and Z. Liu. Power-aware QoS Management in Web Servers. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS'03)*, Cancun, Mexico, December 2003.

[27] M. Xiong, S. Han, and K.-Y. Lam. A Deferrable Scheduling Algorithm for Real-Time Transactions Maintaining Data Freshness. In *IEEE Real-Time System Symposium*, Miami, Florida, Dec. 2005.

[28] R. Xu, D. Zhu, C. Rusu, R. Melhem, and D. Mossé. Energy-Efficient Policies for Embedded Clusters. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05)*, June 2005.

[29] F. Yao, A. Demers, and S.Shankar. A Scheduling Model for Reduced CPU Energy. In *IEEE Annual Foundations of Computer Science*, pages 374–382, 1995.