Dynamic Voltage Scaling in Multi-tier Web Servers with End-to-end Delay Control *

Tibor Horvath, Tarek Abdelzaher, Kevin Skadron
Department of Computer Science, University of Virginia
Technical Report CS-2004-34

Abstract

Energy and cooling costs of web server farms are among their main financial expenditures. This paper explores the benefits of dynamic voltage scaling (DVS) for power management in server farms. Unlike previous work, which addressed DVS on individual servers and on load-balanced server replicas, this paper addresses DVS in multi-stage service pipelines. Contemporary Web server installations typically adopt a three-tier architecture in which the first tier presents a Web interface, the second executes scripts that implement business logic, and the third serves database accesses. From a user's perspective, only the end-to-end response across the entire pipeline is relevant. This paper presents an algorithm for minimizing the total energy expenditure of the multi-stage pipeline subject to soft end-to-end response-time constraints. A distributed power management service is designed and evaluated on a real 3-tier server prototype for coordinating DVS settings in a way that minimizes global energy consumption while meeting end-to-end delay constraints. The service is shown to consume as much as 30% less energy compared to the default (Linux) energy saving policy.

1 Introduction

Complex web services are commonly realized by multi-tier web server systems in order to distribute computation across several computers. The different tiers perform different parts of request processing. For example, an e-business service usually consists of an HTTP server tier, an application server tier, and a database server tier. Client requests to these systems generally have highly varying and unpredictable resource requirements at each tier. Requests for static content such as images or binaries are often served by the first tier alone, with no resource usage in the others. On the other hand, an online purchase transaction would likely have a large processing demand on the application server and the database server, with the HTTP server only transfering a trivial amount of data.

In this paper, we consider the energy efficiency of multi-tier web servers hosting soft real-time services with guaranteed end-to-end response times. These web servers are often significantly over-provisioned in order to meet target response delay constraints even under peak loads. This practice, however, leads to poor overall energy efficiency since such systems are typically under-utilized. The energy (and cooling) costs of large server farms are reported to be a significant part of their total upkeep and maintenance expenses [5, 2]. Excess power consumption not only hurts the operator economically, but it also limits the number of servers per unit volume (in the machine room) due to heat dissipation considerations [3]. Hence, there is an increasing need for solutions that reduce the system's energy consumption with as little effect on performance guarantees as possible.

Dynamic Voltage Scaling (DVS) is a powerful technique that allows significant energy savings by sacrificing some system performance. Reducing voltage requires a roughly proportional decrease in frequency, but power decreases quadratically with voltage. One of the key advantages of DVS (compared to other schemes, such as turning machines off) is that the overhead of performance adjustments is very low, and thus it allows for an agressive power saving policy. Previous research has studied DVS in a single web server with performance guarantees [16]. To our knowledge, no work has been done to address DVS in multi-tier web servers with end-to-end delay constraints.

In this paper, we design, implement, and evaluate a coordinated distributed DVS policy for a traditional three-tier web server system, based on distributed feedback control driven by a simple stage

*This work was supported in part by an NSF grant, no. CCR-0306404.

delay model. We also present the formulation of the problem of determining the optimal DVS policy for such systems. We show experimental results confirming that our solution is efficient and stable.

The rest of this paper is organized as follows. Related work is presented in Section 2. Section 3 presents the general system architecture and DVS solution. Section 4 details the implementation. Performance evaluation is presented in Section 5. The paper concludes with Section 6.

2 Related Work

The importance of reducing both energy and power consumption in server systems is now well-known, and has become a major research topic. Several papers [3, 5, 2] have made the case by pointing out negative environmental effects, high operating costs, power density problems, and expensive infrastructure requirements of large server sites.

Earlier DVS research primarily addressed standalone, battery-operated, embedded mobile devices, which still remains an active research area [13]. Families of DVS algorithms integrated with an RTOS scheduler are proposed for periodic hard real-time task sets in [14, 1, 10]. DVS algorithms assuming similar task sets and a continuous frequency setting model are presented in [22] for multi-processors. Recently, [23] presented the first feedback control-based DVS framework with EDF scheduling in hard real-time systems. A soft real-time energy-efficient scheduler for periodic tasks in embedded systems is presented in [21]. It employs a DVS algorithm similar to the most aggressive one in [14], but it is based on CPU cycle demand distribution histograms built online. It can save more energy while providing statistical performance guarantees.

Much of the previous literature is focused on multimedia task sets. The authors of [17] devise a DVS algorithm for portable systems, which relies on offline workload characterization and probabilistic online detection of arrival or service rate changes. Other DVS algorithms targeted at soft real-time systems predict near-future processing requirements (load) based on past history. PAST [20], one of the first such algorithms proposed by Weiser et al., simply assumes that the predicted (next) time window will have the same amount of idle time as the previous window had. Govil et al. presented and evaluated other prediction schemes [6], including AGED_AVERAGES, which uses a moving average of past samples with geometric decay, and PEAK, which expects short peaks in load, and was shown to outperform PAST. Recently, the authors of [19] applied control theory to predict the future workload. They designed an algorithm, nqPID, that outperforms the aforementioned ad-hoc algorithms, while its performance is also less dependent on parameter tuning. However, their results were validated only by simulation against a periodic task model. Feedback control techniques are used with DVS in [8, 9] to save energy while guaranteeing frame rate in multimedia workloads. The prediction is calculated based on a queueing model. In [8], similar energy savings are reported with reduced computation and improved quality of service over [17]. The authors of [9] use a dead-zone control method to provide strong real-time guarantees without requiring prior workload knowledge like many multimedia DVS schemes. However, since it controls buffer levels, it is not applicable to systems that do not tolerate buffering latency.

Several papers address DVS in standalone servers and server clusters. The authors of [5] present a soft real-time feedback control-based DVS policy combined with request batching. Simulation results show up to 42% savings of CPU energy in a standalone web server, when 90% of the response times are within the target deadline. They do not, however, validate their results by implementation in a real system, nor do they measure total *system* energy savings. A real DVS policy is implemented in [16] for standalone web servers with multiple QoS service classes, which have soft real-time deadlines. The system builds on a proven schedulability bound for aperiodic tasks, due to which it can sustain less than 2% deadline miss ratio. However, the work is restricted to a single tier server. Elnozahy et al. present and evaluate by simulation five different power management schemes for single-tier server clusters [4]. The schemes employ VOVO (*vary-on/vary-off*, i.e. turning nodes on and off depending on cluster load) and/or independent or coordinated (across the cluster) DVS. VOVO attempts to consolidate all workload to just as many nodes as necessary, leaving enough slack for load spikes. An independent DVS policy (IVS) is completely node-local, while a coordinated one (CVS) is constrained to a small frequency range around the cluster average. VOVO combined with CVS is shown to be superior. However, they do not address service pipelines.

A power-aware scheduler for distributed systems with hard real-time end-to-end delay constraints is proposed in [7]. It is capable of determining an optimal voltage schedule in a single task chain (such as a multi-tier web server), but it assumes periodic task chains and requires worst case execution times. Our work is different from the above literature in that we

present and evaluate the first system implementation of a feedback control-based DVS policy in a multi-tier (i.e. pipeline) service that is not restricted to periodic task instances.

3 Architecture

Our multi-tier web service architecture consists of a pipeline of several processing stages. The processing at each stage invokes services of the next stage in a request-response fashion. Requests from a client are addressed to the first stage. Depending on content, they may be processed by subsequent stages sequentially. Such processing is typically in response to calls to business logic scripts and database queries. Eventually, calls and queries return to their originating stage with a response to be sent back to the client. This is the common architecture of most current three-tier systems, where the first stage, an HTTP server, requests dynamically generated content from the second stage, an application server, which in turn requests data from the third stage, a database server.

The non-traditional element in our energy-efficient architecture is that the server machines in the aforementioned pipeline have DVS-capable processors. By employing our novel coordinated DVS policy, the servers minimize the overall power consumption of the web service while satisfying the (soft) real-time end-to-end delay constraints on request processing. The controlled variable is the end-to-end response delay, with the set-point being a target end-to-end delay value. To prevent frequent DVS changes in response to delay fluctuations, a dead-zone is imposed. In other words, no corrective action is taken as long as the measured end-to-end delay lies within an acceptable range between a low and a high threshold. If either threshold is violated, the feedback loop changes DVS settings in the pipeline to recover from the violation.

3.1 Delay Characteristics

End-to-end delays are continuously measured at the first stage, where client requests enter and responses leave. The average CPU utilization U_i is measured at each stage i with sampling period T. The measured end-to-end delay, D, can be broken into a delay component D_i for each stage i. Hence, for an N-stage system, $D = \sum_i^N D_i$. In turn, the delay D_i, on stage i, can be broken into a CPU processing delay, denoted D_i^{CPU}, and a blocking delay, such as I/O blocking, denoted D_i^{block}. This delay is incurred by a request when waiting on or using a resource other than the CPU.

The DVS mechanism manipulates CPU speed and voltage only. Thus, it can only control the CPU delay components, D_i^{CPU}. In contemporary multi-tier servers, significant non-CPU delay components, D_i^{block}, are typically present due to network latency and database I/O. This happens to be a fortunate circumstance from the perspective of DVS schemes, as opposed to a disadvantage. The reason is that DVS schemes opportunistically *increase* CPU delay D_i^{CPU} whenever possible (by slowing processors down) in order to save energy. If the end-to-end delay is primarily a function of D_i^{block} and not D_i^{CPU}, more aggressive energy savings can be accomplished without adverse effects on overall delay performance.

3.2 Dynamic Voltage Scaling

To design an optimal feedback-based DVS scheme in terms of energy savings, we make a couple of assumptions. First, we assume that CPU delay D_i^{CPU} at stage i is a convex function $f(U_i)$ of the CPU utilization, U_i, at that stage. In other words, stage delay increases progressively more steeply as CPU utilization increases. Formally, the second derivative $d^2 f(U_i)/d^2 U_i$ is positive. This assumption is generally true of busy servers according to queueing theory. For example, given a Poisson arrival process and exponentially distributed execution times, we know from queueing theory that $D_i^{CPU} = T/(1-U_i)$, where T is a constant. Hence, $d^2 f(U_i)/d^2 U_i = 2T/(1-U_i)^3$, which is positive for $U_i < 1$.

The convexity assumption leads to a simple set of rules for adjusting CPU speed to maximize energy savings subject to delay constraints. Namely, if the measured end-to-end delay, D, exceeds an upper threshold, step up the frequency of the most loaded machine. Similarly, if the delay drops below a lower threshold, step down the frequency of the least loaded machine.

Intuitively, when the end-to-end delay exceeds the desired value, some processor's frequency must be stepped up to decrease that processor's utilization and consequently decrease delay. The convexity of the utilization-delay function implies that stepping-up the frequency of the most utilized processor is a good rule-of-thumb, because it results in the maximum reduction in delay for the same reduction in utilization. Hence, hopefully, delay can be brought down to the set point with the least additional energy expenditure.

By the same token, when the end-to-end delay is below threshold, stepping-down the frequency of the least utilized processor is a good choice because it

results in the least impact on delay for the same increase in utilization. Hence, this processor can presumably be slowed down the most resulting in the most energy savings.

The main advantage of the above algorithm is simplicity. It uses two simple rules that require only per-machine total utilization measurements and a measurement of end-to-end delay. In particular, it does not need to know individual stage delays, task execution times, or processor power characteristics.

The algorithm does not actually lead to an optimal solution to the energy minimization problem because it implicitly assumes that energy savings are proportional to utilization changes. In general, this is not true. Fortunately, if the processors's power-frequency curve and the workload's utilization-delay function are known, the above optimization algorithm can be easily adapted to produce the optimum energy consumption as shown below.

3.3 Optimality Conditions

Let us assume that the power consumption P_i of stage i can be approximated by:

$$P_i = A_i f_i^n + B_i \tag{1}$$

where A_i and B_i are constants.[1] This assumption is accurately satisfied in realistic systems, with n ranging between 2.5 and 3. Second, assume that the delay D_i^{CPU} of a stage i is approximately related to its utilization U_i by the equation:

$$D_i^{CPU} = C_i U_i^m + G_i \tag{2}$$

where C_i and G_i are constants. In reality, this equation is not exact. However, it is general enough to approximate a large variety of functions in the region around the nominal operating point of the system. In general, it is possible to obtain the exponents n and m, and constants A_i, B_i, C_i, and G_i by curve fitting against empirical measurements obtained from profiling the system.

If the workload arrival rate at stage i is λ_i cycles/sec, the utilization U_i of that processor is λ_i/f_i, where f_i is the service rate or frequency in cycles/sec. Equivalently, $f_i = \lambda_i/U_i$. Substituting in Equation (1) and summing over the entire pipeline, the total power consumption P of the N-stage system can be expressed by:

$$P = \sum_{i=1}^{N} A_i \frac{\lambda_i^n}{U_i^n} + B_i \tag{3}$$

[1]The general rule of thumb is that $P \propto V^2 f \propto f^3$. In reality, $f \propto V$ is a simplification, hence our more general expression.

Our objective is to minimize that power consumption subject to the constraint $\sum_{i=1}^{N} D_i^{CPU} + D_i^{block} \leq L$, where L is the maximum desired latency. Taking the equality condition as the limiting case, and substituting from Equation (2), this constraint can be rewritten as:

$$\sum_{i=1}^{N} C_i U_i^m = K \tag{4}$$

where $K = L - \sum_{i=1}^{N} D_i^{block} - \sum_{i=1}^{N} G_i$, which we assume is a constant independent of frequency settings, since blocking delays are not affected by CPU speed.

To solve the aforementioned constrained optimization problem, we first solve Equation (4) for U_N, which yields:

$$U_N = \left(\frac{K - \sum_{i=1}^{N-1} C_i U_i^m}{C_N} \right)^{1/m} \tag{5}$$

Then, we substitute for U_N from Equation (5) into Equation (3), which yields:

$$P = \sum_{i=1}^{N-1} A_i \frac{\lambda_i^n}{U_i^n} + A_N C_N^{n/m} \frac{\lambda_N^n}{(K - \sum_{i=1}^{N-1} C_i U_i^m)^{n/m}} + B \tag{6}$$

where $B = \sum_{i=1}^{N} B_i$. Taking the derivative dP/dU_i, then substituting from Equation (5) back with U_N, we get for each i:

$$\frac{dP}{dU_i} = -nA_i \frac{\lambda_i^n}{U_i^{n+1}} + nA_N C_i U_i^{m-1} \frac{\lambda_N^n}{C_N U_N^{n+m}} \tag{7}$$

It can be easily shown that the second derivative in this case is positive, which means that equating all derivatives dP/dU_i to zero, we find the point at which power consumption is minimized. From Equation (7), $dP/dU_i = 0$ gives:

$$\frac{U_1}{W_1} = \frac{U_2}{W_2} = ... = \frac{U_N}{W_N} \tag{8}$$

where W_i is a weight given by $\left(\frac{A_i \lambda_i^n}{C_i}\right)^{1/(m+n)}$.

To minimize power consumption across the pipeline subject to the end-to-end delay constraint, a feedback loop is added to equalize the weighted utilizations of all stages. Utilization is manipulated by changing the CPU frequency settings.

3.4 Improved Algorithm

To converge on the condition expressed in Equation (8), average local stage CPU ulization measurements, U_i, are broadcast by each machine at each sampling period. Average end-to-end delay D is computed by the first stage and also broadcast to all

stages at each sampling period. Given this information, the DVS algorithm on each machine computes the weighted utilization, U_i/W_i for each stage i. It is desired to keep these values as equal as possible while observing that a given deadline miss ratio is not exceeded.

To ensure that a maximum tolerable miss ratio r is not exceeded, one can compute (from the expected workload distribution) the conditional probability that a deadline miss will occur in the next sampling interval given that the maximum delay observed in the current sampling interval is some fraction $\alpha_{hi} < 1$ of the actual deadline L. We denote this conditional probability by $P(D[k+1] > L|D[k] < \alpha_{hi}L)$, which is a function of α_{hi} (where $D[k]$ and $D[k+1]$ denote the delay measurements in the current and next samples respectively). If the maximum acceptable deadline miss ratio is r, we would like to ensure that $P(D[k+1] > L|D[k] < \alpha_{hi}L) \leq r$. Given an analytically derived or empirically measured conditional probability function, the equality condition, $P(D[k+1] > L|D[k] < \alpha_{hi}L) = r$ can be solved for α_{hi} simply by finding the point where the curve of this function reaches value r. The following two feedback rules are then applied:

- If $D > \alpha_{hi}L$ (overload), machine i with $\max_i\{U_i/W_i\}$ steps up its frequency.
- If $D < \alpha_{lo}L$ (underutilization), machine i with $\min_i\{U_i/W_i\}$ steps down its frequency (where $\alpha_{lo} < \alpha_{hi}$).

The first rule guarantees that the conditions for a sustained miss ratio of r or more are always corrected to reduce miss ratio. The second rule allows energy savings to be applied when the system is underutilized. The parameter α_{lo} can be selected to achieve a good compromise between power savings and actual deadline miss ratio. Finally, note that if W_i are equal for all stages, the algorithm reduces to the one described in Section 3.2.

3.5 Discussion

A few remaining issues are worth to point out regarding the proposed algorithm. First, note that when choosing the sampling period T, one major concern is to limit controller overshoot as much as possible. We use a small T value at overload because it provides high responsiveness. However, such a small period is not suitable during underload, because it leads to a small set of delay samples that makes their average not sufficiently representative. Therefore, the sampling period during underload is increased. Our results indicate that this yields a good compromise between soft real-time performance and energy savings.

We rely on single-step actuation as opposed to changing multiple CPU frequencies at the same time. This is especially significant since the number of DVS frequency settings is usually small, and multiple-step actuation could easily overreact to load variations. Having said so, the controller gain can be changed by changing the sampling period. Smaller periods result in higher gain since acuation is more frequent (which the actuation step remains the same). It is important to choose a sampling period that does not violate loop stability. A control-theoretic analysis of this loop is carried out for this purpose. Control-theoretic models of absolute delay control loops have been presented in [15].

The overall algorithm behavior can be effectively adapted to user requirements by adjusting the delay thresholds α_{hi} and α_{lo}. Decreasing these thresholds generally reduces both deadline misses and power savings.

Another architectural feature that impacts performance is the issue of agreement in our distributed coordination scheme. Although synchronous coordination should be capable of guaranteeing coherence and consistency, it is expensive to enforce. Therefore coordination (i.e., sharing of utilization values and end-to-end delay) is done asynchronously. Assuming that average utilization and average delay do not change abruptly from sample to sample (which can be ensured by an appropriate choice of the sampling period), asynchrony has very little effect since state is not very time-sensitive. However, asynchrony does give rise to the possibility that, in an overload or underload situation, there might be no agreement on which stage should react (albeit there is likely to be an agreement on whether the system is underloaded or overutilized). As long as *any* stage decides to react, lack of agreement can only increase the extent of system reaction (as two or more machines decide to perform a corrective action). In other words, lack of agreement increases controller gain, which can be easily accounted for in stability analysis.

Let us also remark that since we do not assume that stage clocks are synchronized, the exact actuation times may vary throughout the pipeline. We note, however, that in the worst case, any stage's reaction will be late by at most T since the last broadcast of end-to-end delay. Since we choose T to be small (compared to end-to-end deadlines) for fast system reaction, we argue that this delay is acceptable.

Finally, observe that while we described the algorithm for a single class of clients with the same dead-

line, it is straightforward to generalize to multiple classes. The only change is that the first stage now measures the end-to-end delay for each class separately. This delay vector is broadcast to other stages. Let the deadline of class i be L_i and its measured end-to-end delay be D_i. Each stage executes the following two rules:

- If $\exists i : D_i > \alpha_{hi} L_i$ (overload), machine i with $\max_i\{U_i/W_i\}$ steps up its frequency.
- Else, if $\exists i : D_i < \alpha_{lo} L_i$ (underutilization), machine i with $\min_i\{U_i/W_i\}$ steps down its frequency (where $\alpha_{lo} < \alpha_{hi}$).

In our evaluation, we experiment with one class only, since Apache servers and Linux socket queues do not support priority scheduling.

4 Implementation

In designing the structure of our implementation, our primary goal was to make our DVS policy as independent of the actual server software as possible. This is preferable because it is unobtrusive to the server software that we want to leave intact, and extensible because it needs not be modified to accomodate a new server software. There is no need to modify any existing server software on the source code level as long as we can measure the end-to-end processing delay on the first stage without doing so. This may be done by taking advantage of certain hooks the server software provides for plugin modules. The Apache web server [18], for example, does provide such hooks. But our controller could work with any alternative solution through the same interface that it provides.

Our prototype three-tier web server system is composed of three laptop computers with DVS-capable processors, each running Linux 2.6. The first two computers run Apache 1.3 as an HTTP server and as an application server, respectively. The third computer runs the MySQL database server [12]. Our DVS policy is implemented independently as a standalone daemon to be started on all servers. The daemons establish TCP connections to the ones running on their previous and next stages, they self-coordinate once started on all stages, then they start controlling the pipeline.

Measuring end-to-end delay in practice is a challenge, and for this reason it is also a serious limitation of all delay model-based approaches. True end-to-end delay could only be measured by the kernel. However, this would require correlating TCP/IP messages in the network-level protocol stack based on application-level (for example HTTP) message content, which is known to be computationally prohibitive, not to mention that it is heavily application dependent. Alternatively, measuring delay in user space is a feasible yet imprecise solution. We choose this latter solution because even though it may not be completely accurate, it is much more flexible, and it gives a reasonable approximation if the network is not the bottleneck resource on the first stage.

To obtain end-to-end delay samples, processing delays of the first stage (and thus the whole pipeline) are measured by our Apache extension module attached to the "post read-request" and the "logger" hooks. The time elapsed between the invocation of these two hooks for a given request is its measured end-to-end processing delay. The DVS daemon running on the first stage provides a local (System V) Message Queue IPC interface to gather these delay samples. The end-to-end delay sample statistics are periodically sent to all subsequent remote stages via TCP/IP messages, originating at the first stage.

At the end of each sampling period, average stage CPU utilization is measured by the DVS daemon on all stages. The utilization values are obtained from the the Linux kernel, by reading its clocktick accounting statistics from the virtual file "/proc/stat". Averages for each period are computed by subtracting the values collected at the end of the previous period from those at the end of the current period. The average stage CPU utilizations, along with the stage's current CPU frequency setting, are periodically sent from all stages to each other also via TCP/IP messages.

As discussed earlier, we select a short sampling period during overload for the sake of high responsiveness to deadline misses. Our choice is $T = 200\,ms$ because in a typical three-tier system only a small number of requests exit during this time, which means the system quickly reacts after observing a few samples. It also results in a low controller overhead, since coordination data will be measured and sent only five times per second. During underload, the sampling period is equal to the deadline. The reason is that this prevents the controller from decreasing system capacity before current request delays could be measured, as long as deadlines are met. This much longer sampling period does not mean, however, that the system becomes unresponsive during underload, because it is implemented in terms of the normal short periods by aggregating their samples.

The DVS algorithm is invoked at the end of each (short or long) sampling period. Through the coordination mechanisms described above, all stages ideally have a consistent view of current CPU frequencies, average CPU utilizations, and the end-to-end

delay samples, hence they can solve the current DVS problem instance independently. When a stage's solution indicates so, the stage applies the calculated frequency adjustment to itself. The actual CPU speed setting is implemented by invoking the standard "userspace" frequency scaling governor of the Linux CPUFreq device driver.

5 Evaluation

We evaluate our algorithm by comparison to a base case. In the base case, we set the CPU frequency to constant maximum on all stages. Let us point out that this does not necessarily mean that the CPUs will actually constantly run at that frequency. Linux (as most modern operating systems) attempts to save power by default when the CPU is idle, even without a DVS policy. The exact way is platform and parameter-specific, but usually the CPU is turned off until a hardware interrupt occurs. Our platform uses the default method for x86 platforms: it executes the "hlt" instruction, which halts the CPU. Our experimental DVS implementation is run on top of this policy. Thus, our reported power savings are those above the aforementioned baseline policy.

5.1 Workloads

To evaluate the expected real performance of our algorithm, we attempt to create a realistic server workload modeled after that of a typical three-tier web server. As most serious services rely on large volumes of data, we create a reasonably-sized database on the third stage. We have 500 tables, each table contains 1000 records, and each record consists of 20 fields. All records are initially filled with a key and 19 random fractional numbers. The physical size of the database prevents it from being entirely cached on our machine, making this stage I/O-intensive.

The second stage implements its application server functionality using CGI scripts, which perform data access and simulate data processing. The script first requests the database server to perform one of three different types of data manipulation actions: query record based on primary key; update record selected by primary key; and query records based on textual search pattern. The requested action is randomly chosen. In the first two cases, the key is randomly selected from the existing valid keys, and in the third case, the search pattern is a random 3-digit number as a substring. This randomization helps avoid invalid results due to disk caching by decreasing spacial locality of data accesses. These actions are, although minimal, representative of many real applications because they consist of both reads and writes, they involve both simple indexed lookups and complex non-indexed searches, and they can have highly varied execution times. Once the database access is finished, the script performs numeric calculations to simulate data processing. This processing, along with the processing done by the database client library (before sending a request to the database server), makes the second stage CPU-intensive, with the amount of CPU processing performed depending on the size of the data set received.

Finally, for the first stage, we create a small CGI script that sends an HTTP GET request to the second stage, and copies the response to the client. It models the non-CPU intensive mediator and response-assembler role the HTTP server tier typically has.

Test requests from the client are generated by the *httperf* [11] workload generator tool at various average rates. The request interarrival times are exponentially distributed. An individual TCP connection is created for each request. Figures 1 and 2 demonstrate the baseline (no-DVS) end-to-end response delay distribution for requests generated at average rates of 150 and 300 requests per minute, respectively. We can see that the distributions are realistic heavy-tailed ones even at light load, which verifies that our server workload is appropriate.

5.2 Measurement Setup

We place our three server laptops on one network segment, making sure that unintended traffic does not flood it. The workload generator is run on a dedicated client computer located in a separate network segment. To filter out possible measurement errors due to lack of client resources, we verify that close to 100% of system time is available for request generation on the dedicated computer during each test.

To measure the power consumption of the laptops, we use three custom measurement circuits that sense the current flowing from each laptop power supply (AC adapter). Since the adapters provide constant voltage (18.5V), we need not measure it. Observe that the adapter's voltage remains the same even when the CPU is performing DVS. Hence, our measurements reflect the true total power consumption of the laptop, including that of the CPU and other circuits. During power measurements, we remove the batteries from the laptops, since we do not want to measure power consumed to charge them, and we want the laptops to obtain power exclusively from the AC adapter. Also, since server systems usually

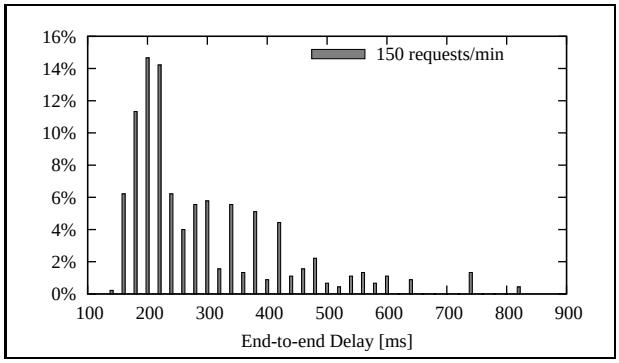

Figure 1: Workload End-to-end Delay Distribution at 150 requests/min

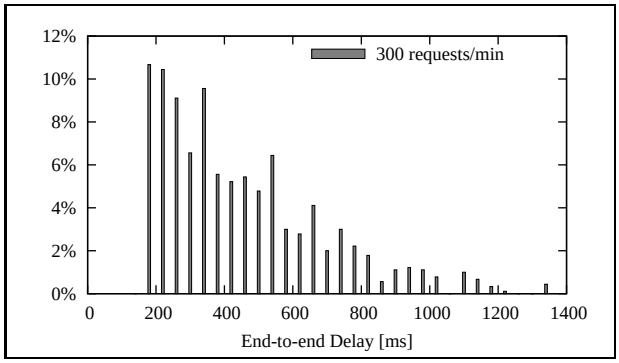

Figure 2: Workload End-to-end Delay Distribution at 300 requests/min

do not include a display, we turn off the LCD backlighting, which drains a significant amount of power. We do not, however, turn off the display adapter, by which our power savings could be improved even a little further without affecting performance.

Current readings for all three laptops are performed simultaneously at a rate of 2000 samples per second per channel, using three channels of a Texas Instruments PCI-6034E data acquisition card installed in a separate computer. The average stage power consumptions for the test duration are then calculated offline. Performance data, such as the deadline miss ratio, is collected from the output of the workload generator tool.

5.3 Choosing The Delay Thresholds

To evaluate the performance of our DVS algorithm, we must first make an appropriate choice of the upper and lower delay thresholds, α_{hi} and α_{lo}, described in Section 3.4. Violations of these thresholds trigger re-

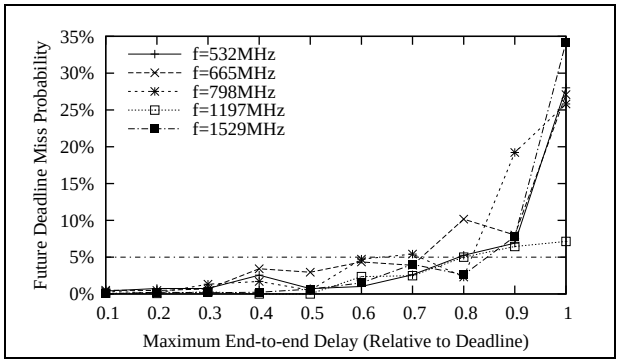

Figure 3: Chosing the Delay Threshold

actions to overload and underutilization respectively. As mentioned in Section 3.4, the upper threshold is chosen such that $P(D[k+1] > L | D[k] < \alpha_{hi} L) = r$, where L is the end-to-end deadline, $D[k+1]$ is the end-to-end delay in the next sampling period, $D[k]$ is the maximum end-to-end delay measured in the current sampling period, and r is the maximum tolerable deadline miss ratio. In other words, we would like the DVS algorithm to increase CPU speed when the conditional probability of a future deadline miss reaches the maximum tolerable miss ratio. Figure 3 plots the aforementioned conditional probability for our workload as a function of the delay threshold. This curve was obtained empirically by observing the delays in every two successive sampling times. The conditional probability of a future deadline miss depends on CPU speed because at lower speeds individual requests contribute more to server delay, hence causing a larger delay variability. We imagine that in high performance servers where individual requests are very small compared to server capacity, the granularity of individual requests will play a smaller role. Let us take 5% to be the largest tolerable miss ratio. From Figure (3), we see that a threshold of $\alpha_{hi} = 0.7$ guarantees that the maximum miss ratio will remain below 5%.

The choice of the lower threshold determines how aggressive power management will be. Empirical data shown in Figures 4 and 5 suggests that $\alpha_{lo} = 0.4$ is a good low threshold in the sense of saving the most power without impacting the miss ratio. Hence, we use $0.7L$ and $0.4L$ are the upper and lower delay thresholds respectively.

5.4 Performance Results

Next, we evaluate the energy savings and deadline miss ratio of the 3-tier service that runs our DVS al-

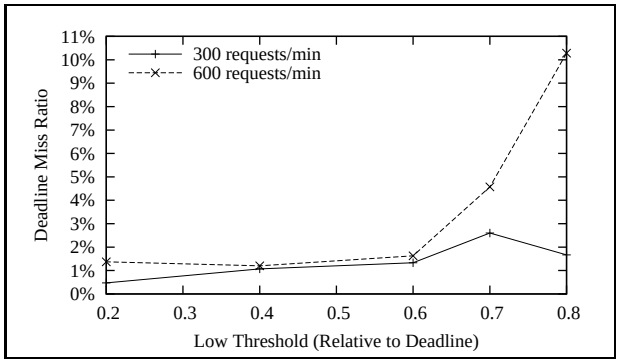

Figure 4: Deadline Miss Ratios of Different Low Thresholds (High Threshold=0.9, L=10s)

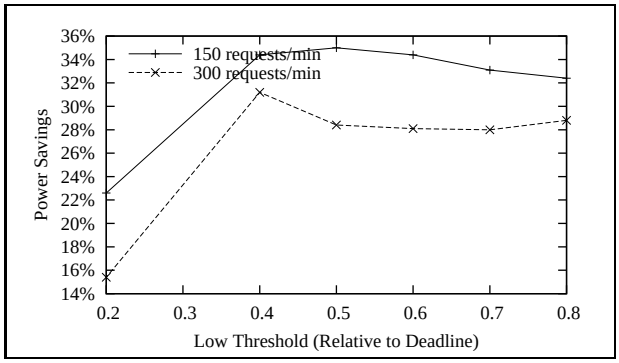

Figure 5: Power Savings of Different Low Thresholds (High Threshold=0.9, L=10s)

gorithm. Each data point in our results is obtained by running an experiment for 3–5 minutes, repeating and averaging as necessary. Since we want to show the stable behavior of the system, we eliminate transient cold-start effects by running a short (18–30s) lead-in workload prior to starting each experiment.

Figure 6 plots the deadline miss ratio as a function of the average request rate, which we vary from 0 (no load) to 700 requests/minute (severe overload). We perform several sets of experiments for different deadlines ranging from 4 to 10 seconds. We choose these deadlines because given that more than 20% of the response delays in our workload are inherently over 200ms, a deadline below 4 seconds would be hard to achieve while saving power. Obviously, on a faster machine shorter deadlines are possible. The eight-second rule [24] tells us that users are generally not willing to wait more than 8 seconds for a web page to load. Therefore we should not target much higher deadlines. The graph shows that the system begins to saturate at 450 requests/minute in each case, and

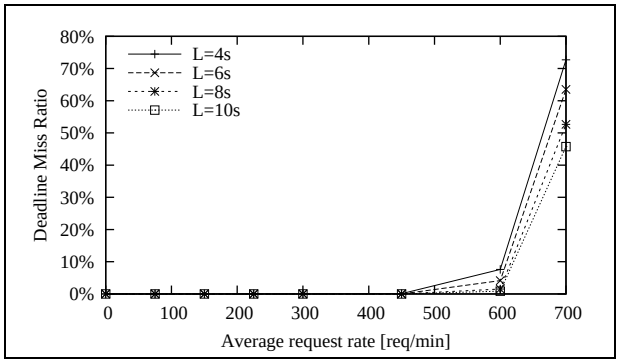

Figure 6: Baseline Performance

that saturation is naturally slower with higher deadlines.

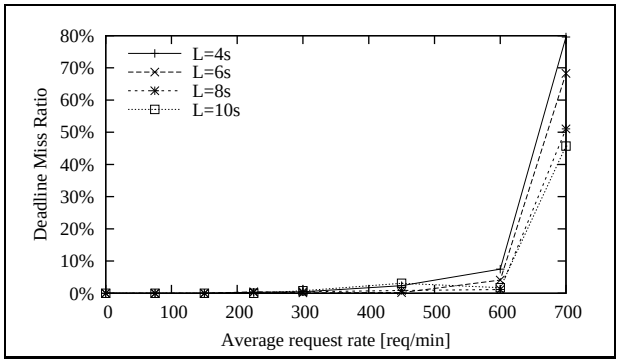

Figure 7: Feedback DVS Performance

Figure 7 presents the miss ratio of two versions of our DVS algorithms. The first version is one were all weights W_i as assumed to be equal (as an approximation). The advantage of this version is that it does not require knowledge of the power caharacteristics of the CPUs, and characteristics of machine workload. If such information is available, however, it is possible to compute the coefficients W_i derived in Section 3.4. Figure 8 shows the resulting improved (optimal) algorithm using weights derived from empirical measurements. We can see that both systems successfully control the end-to-end delays so that at least 95% of the deadlines are met when the system is underloaded. Even when it is overloaded, the deadline miss ratio still closely follows that of the base case. As we see next the improved DVS algorithm saves more power.

To illustrate what energy savings are achieved, in Figure 9, the total power consumption of the three servers using the feedback DVS policies is compared

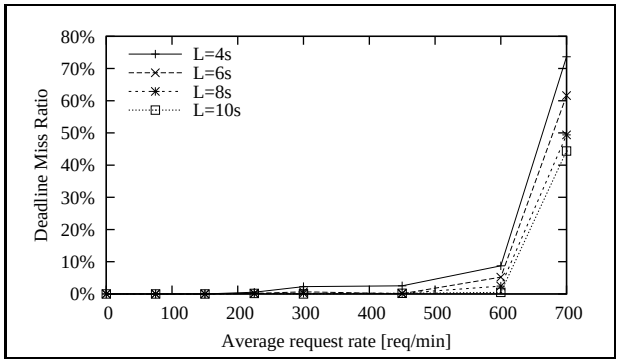

Figure 8: Weighted Feedback DVS Performance

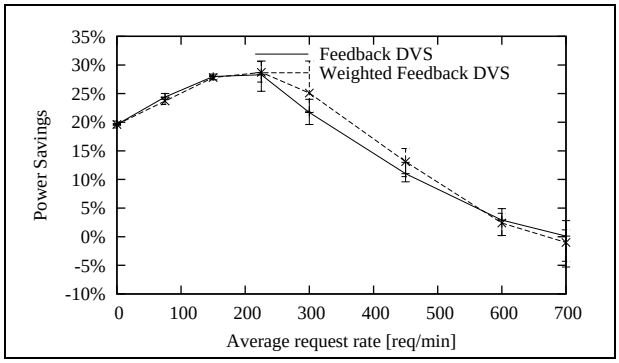

Figure 10: Feedback DVS Power Savings

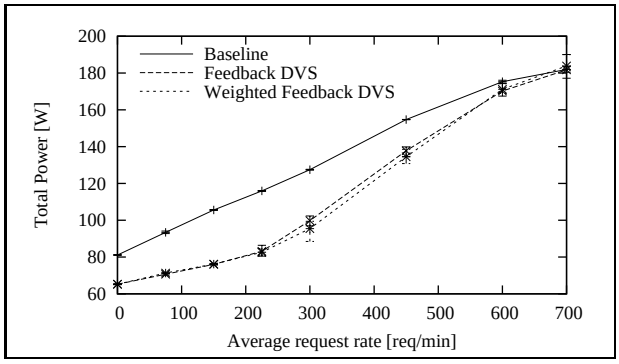

Figure 9: Total System Power Consumption

to the total power consumption using the baseline policy. For each load level, the power samples are obtained by performing individual measurements for each deadline. The lines connect the averages of these samples, while the error bars show the minimum and maximum values. We can verify that the baseline power saving policy in fact saves a considerable amount of power in itself when the system is underutilized.

Finally, Figure 10 displays the overall power savings attained by the two feedback DVS policies. We can see that both of our algorithms can achieve above 30% total power savings under medium load. The graph also demonstrates that the improved algorithm in fact slightly outperforms the original algorithm. Let us observe that approximately 20% power is saved even when the system is idle, because background processes and periodic kernel operations such as the timer handlers all run at lower frequency. The highest relative power savings are realized at medium load (150–225 requests/min). The shape of the curves is explained by the fact that in light load (0–75 requests/min), the CPU is often idle making the original Linux policy more efficient at power saving. As load increases, there is less chance for the hlt instruction to be performed. Our policy wins because it can run the processor at a lower frequency. Progressing towards heavier loads (above 300 requests/min), there is no longer much opportunity to lower processor frequencies. Therefore the power savings diminish. Since most server farms are normally overprovisioned, a substantial power reduction is possible using our schemes.

Four important points are made from the experimental results. First, non-trivial power savings can be achieved using our DVS scheme while maintaining the miss ratio at a low rate. Second, the optimal savings occur when the *weighted* utilizations of all machines are equal and *not* when utilizations are perfectly balanced. This interesting obsevation is confirmed both theoretically and experimentally. Third, balancing machine utilizations is an adequately good heuristic that is very easy to implement largely independently of load and machine characteristics. Finally, the scheme does not require any modifications to server code. We therefore believe that our algorithms are both practical and efficient, which makes them a good candidate for implementation in real-life systems.

6 Conclusions

In this paper, we presented a DVS control algorithm that mimimizes power consumption in a server pipeline subject to end-to-end latency constraints. While the algorithm was described for a single class of clients, straightforward extensions to multiple classes are possible. A formal derivation of optimality conditions was given, together with a feedback control architecture that drives the system to satisfy these

conditions. Interestingly, it was shown that the optimal power savings to do not always coincide with the load balanced condition of equal utilization on all servers. However, in practice such load balancing is a good approximation. A functional prototype of this system was implemented and experimentally evaluated. Emprical measurements confirm theoretical results and show that our system consumes up to 30% less energy than the default Linux power saving mode. These savings have a significant effect on the operation cost of large server farms. This work will be extended to larger server clusters with multiple machines per stage and multiple classes of clients with different timing constraint.

References

[1] H. Aydi, P. Mejia-Alvarez, D. Mosse, and R. Melhem. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, page 95. IEEE Computer Society, 2001.

[2] R. Bianchini and R. Rajamony. Power and energy management for server systems. In *IEEE Computer, 37(11)*, 2004.

[3] P. Bohrer, E. N. Elnozahy, T. Keller, M. Kistler, C. Lefurgy, and R. Rajamony. The case for power management in web servers. In *Power-Aware Computing*, 2002.

[4] E. Elnozahy, M. Kistler, and R. Rajamony. Energy-efficient server clusters. In *Proceedings of the Workshop on Power-Aware Computing Systems*, 2002.

[5] M. Elnozahy, M. Kistler, and R. Rajamony. Energy conservation policies for web servers. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, 2003.

[6] K. Govil, E. Chan, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power cpu. In *Proceedings of the 1st annual international conference on Mobile computing and networking*, pages 13–25. ACM Press, 1995.

[7] D.-I. Kang, S. Crago, and J. Suh. Power-aware design synthesis techniques for distributed real-time systems. In *LCTES/OM*, pages 20–28, 2001.

[8] Z. Lu, J. Hein, M. Humphrey, M. Stan, J. Lach, and K. Skadron. Control-theoretic dynamic frequency and voltage scaling for multimedia workloads. In *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 156–163. ACM Press, 2002.

[9] Z. Lu, J. Lach, M. R. Stan, and K. Skadron. Reducing multimedia decode power using feedback control. In *Proceedings. 21st International Conference on Computer Design*, pages 489–496, Oct. 2003.

[10] P. Mejia-Alvarez, E. Levner, and D. Mosse. Power-optimized scheduling server for real-time tasks. In *Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)*, page 239. IEEE Computer Society, 2002.

[11] D. Mosberger and T. Jin. httperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance*, pages 59–67. ACM Press, June 1998.

[12] MySQL AB. MySQL database server. http://www.mysql.com/.

[13] I. K. Osman S. Unsal. System-level power-aware design techniques in real-time systems. In *Proceedings of the IEEE, 91(7)*, 2003.

[14] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 89–102. ACM Press, 2001.

[15] L. Sha, X. Liu, Y. Lu, and T. Abdelzaher. Queuing model based network server performance control. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, Dec. 2002.

[16] V. Sharma, A. Thomas, T. Abdelzaher, K. Skadron, and Z. Lu. Power-aware QoS management in web servers. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, page 63. IEEE Computer Society, 2003.

[17] T. Simunic, L. Benini, A. Acquaviva, P. Glynn, and G. D. Micheli. Dynamic voltage scaling and power management for portable systems. In *Proceedings of the 38th conference on Design automation*, pages 524–529. ACM Press, 2001.

[18] The Apache Software Foundation. The Apache HTTP server. http://www.apache.org/.

[19] A. Varma, B. Ganesh, M. Sen, S. R. Choudhury, L. Srinivasan, and J. Bruce. A control-theoretic approach to dynamic voltage scheduling. In *Proceedings of the 2003 international conference on Compilers, architectures and synthesis for embedded systems*, pages 255–266. ACM Press, 2003.

[20] M. Weiser, B. Welch, A. J. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Operating Systems Design and Implementation*, pages 13–23, 1994.

[21] W. Yuan and K. Nahrstedt. Energy-efficient soft real-time CPU scheduling for mobile multimedia systems. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 149–163. ACM Press, 2003.

[22] D. Zhu, R. Melhem, and B. Childers. Scheduling with dynamic voltage/speed adjustment using slack reclamation in multi-processor real-time systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, page 84. IEEE Computer Society, 2001.

[23] Y. Zhu and F. Mueller. Feedback EDF scheduling exploiting dynamic voltage scaling. In *Real-Time and Embedded Technology and Applications Symposium*, pages 84–93, May 2004.

[24] Zona Research, Inc. The economic impacts of unacceptable web site download speeds, Apr. 1999.