ENERGY MANAGEMENT IN REAL-TIME MULTI-TIER INTERNET SERVICES

A Dissertation

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy

Computer Science

by

Tibor Horvath

May 2008

Approvals

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Computer Science

Tibor Horvath

This dissertation has been read and approved by the examining Committee:

Kevin Skadron (Advisor)	Marty Humphrey (Chair)
Sudhanva Gurumurthi	Tarek F. Abdelzaher
John C. Lach	

Accepted for the School of Engineering and Applied Science:

James H. Aylor (Dean)

May 2008

Abstract

Energy management only recently emerged as a major consideration in the design of high-performance Internet services. As a result of requirements for continual performance scaling, the energy required to provide these services also massively increased—an unfortunate corollary of Moore's Law. At the same time, the high scalability of Internet services is commonly achieved by employing multi-tier (functionally distributed) clustered architectures. At high-demand sites, the number of server computers in such clusters can be very large (on the order of thousands or above). Due to the high overall power consumption and related heat dissipation that these server farms exhibit, severe operational challenges arise such as high energy costs, high cooling costs (operation and maintenance), costly infrastructure requirements (such as sophisticated power-delivery and cooling systems), increased space demands (server unit density is limited by heat dissipation), and decreased system reliability (heat-related failures). These increased operational costs constitute a significant part of total upkeep and maintenance expenses of large sites. Advances in networking technology have and will continue to cause demand and reliance on Internet-based services to accelerate, exacerbating the above-mentioned issues.

By 2002, researchers already identified that there is significant potential for reducing energy use in Web servers. Because of observed daily and weekly demand fluctuations due to the natural cycle of human activity levels, Internet server workloads tend to show extended periods of low-load or even near-idle operation. Furthermore, newer server-class hardware started to adopt power saving modes previously only supported on mobile systems, which allows judicious reduction of server capacity (and the corresponding energy use) during those off-peak load conditions. Since then, several techniques appeared to manage power dissipation of individual machines. However, since most techniques incur some performance penalty, minimizing the global energy expenditure of a cluster with minimal effect on its performance remains a challenge.

This dissertation presents the theoretical analysis of several aspects of this energy minimization problem, starting out with a basic multi-tier server model, which is then extended to deal with multiple priorities and reconfigurable clusters; it discusses the optimization of spare capacity in such clusters when multiple machine sleep states are available; and finally, it analyzes how external disturbances, such as those triggering dynamic thermal management actions, affect the performance of cluster power management algorithms.

Acknowledgments

This research was supported in part by NSF grant no. `CNS-0306404`, NSF grant no. `CNS-0615277`, Army Research Office grant no. `W911NF-04-1-0288`, and a grant from Intel MRL.

The TPC-W server I use as a workload in experiments throughout the dissertation was based on the implementation from the Parallel and Distributed Systems Research Group at the Computer Science Department, New York University. The client was adapted from the implementation originally done at the University of Wisconsin-Madison. Our ClusterControlWare framework was developed from a complete rewrite of initial code contributed by Jin Heo from the Department of Computer Science at the University of Illinois at Urbana-Champaign. Xue Liu, from the same department, helped with the final version of the derivation of the optimality condition in section 2.2.3.

I wish to thank my advisors, Kevin Skadron and Tarek Abdelzaher, for their invaluable guidance during my program, as well as the rest of my advisory committee, Sudhanva Gurumurthi, Marty Humphrey, and John Lach, for their helpful suggestions for this work. I am also truly grateful to my family for their constant support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

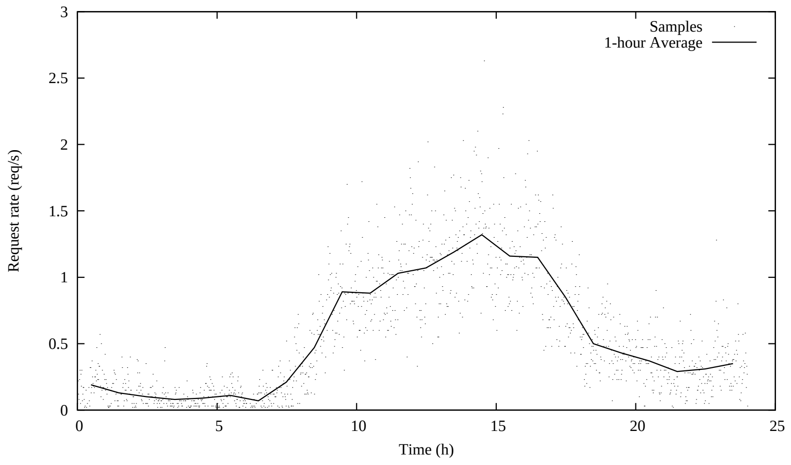
1.1 Motivation

Data centers hosting high-performance Internet services consist of large-scale, heterogeneous, multi-tier server clusters. Thousands of server machines can be densely packed in the machine rooms of such data centers. The power consumption of these machines alone is a great problem, first because of high power delivery (infrastructure) expenses, and more importantly because the daily energy costs constitute a significant part of operational costs. High availability requirements that are natural for Internet services also call for expensive investments in backup power supplies. Another serious problem in a data center is the heat generated by the servers. Even with dynamic, temperature-aware request distribution, expensive cooling infrastructure (including backup cooling) is necessary to keep the temperature of the machine room safe. These cooling systems consume significant amounts of energy too, raising operational costs even further. A rule of thumb is that for every watt consumed by the computer systems, another watt of electricity is consumed by the climate control. Even with good climate control, local hotspots can still form during periods of higher demand, which has a negative impact on equipment reliability and lifetime. Since the number of machine components is huge, the overall probability of temperature-related failure can be high. This, again, causes high maintenance costs. In short, energy requirements—both for the computer systems and their cooling—are becoming a major cost component in data center operation, presenting a threat to the competitiveness of service providers and a barrier to the deployment of new Internet services. Furthermore, the energy requirements of data centers are large enough to present environmental concerns.

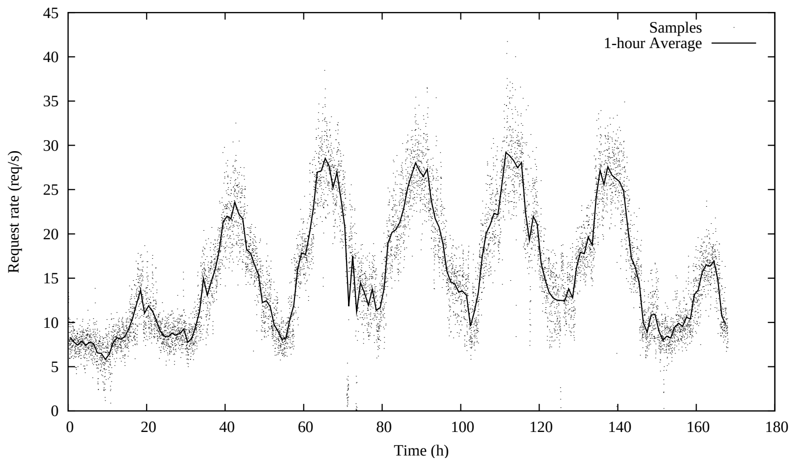
The potential for significant energy usage reduction exists because of the uneven utilization characteristics of high-performance Internet services. Studies of real-life Web services show that the number of requests received by a service is highly time-varying [11]. Depending on the nature of the service, there can be more and less active periods on different time scales (e.g. hour, day, or month). Other sources of fluctuation include, for example, high-profile events or breaking news, which can abruptly and drastically raise the demand for news websites. However, even during regular operation, cyclical load variations can be observed, as exemplified in Figure 1.1 showing two typical Web server traces.

To meet peak demands, data centers are typically overprovisioned. This means that during capacity planning, the maximum expected (or accepted) load is determined, and hardware resources sufficient to comfortably pass a corresponding stress test are provided. If the load estimates are correct, the full capacity of the server farm is utilized only for short time periods, with a large portion of the cluster otherwise idle. Ideally, all hardware would operate completely energy-proportionally, i.e. the energy used by a machine would be proportional to its utilization, in which case the idle portion of the cluster would not consume any electricity. However, in reality idle machines do consume significant energy, mainly due to conversion losses, leakage, and standby power. Hence, overprovisioning results in large amounts of wasted energy.

One option to reduce this waste is consolidating several services on one cluster. This may improve overall utilization (translating to reduced idleness), yet the periodic trends are unlikely to be eliminated. A more promising alternative is the application of dynamic power management (DPM), the essence of which is to adjust the power states of the machines over time to most closely match the actual demand, eliminating a large part of the idle power dissipation. DPM encompasses several techniques, such as putting unused components (e.g., network interfaces, disks, or even DRAM modules) to sleep, slowing down the CPU, or shutting down entire machines in a cluster. Naturally, the biggest challenge in DPM is providing an algorithm (referred to as a *policy*) that decides which components or machines should be in which power states at any given time. In



(a) Daily load fluctuation of a busy EPA Web server.



(b) Weekly load fluctuation of the 1998 World Cup Web servers (Sun–Sat).

Figure 1.1: Web server traces showing cyclical load variations. (Data source: The Internet Traffic Archive [39].)

fact, finding optimal policies appears to be a hard problem using even relatively simple system models [38].

It is important to point out that all of the above-mentioned power management techniques impact performance. Sleep modes entail both shutdown and wakeup latency, during which processing cannot be performed (but energy is still consumed). Slowed-down CPUs effectively reduce server throughput and increase delays. Clearly, power management for servers must save energy while maintaining performance requirements. The requirements are usually specified in a service level agreement (SLA), which is a contract describing minimum acceptable values for several performance metrics that the service provider (e.g., hosting center) must meet, along with associated penalties for violations. Of particular interest are service latency requirements, since latency is directly observable by each individual client.

1.2 Research Approach

This dissertation addresses the question of how to optimize energy savings on large-scale, scalable, multi-tier server clusters with bursty, aperiodic workloads, while still providing soft real-time performance. This entails finding the optimal global operating point of a system with given load and power characteristics and performance constraints, in the space of all allowed power states. The expected workload of such a system is aperiodic, with highly varying execution demand and possibly intensive I/O activity. Given the importance of service latency, we assumed performance constraints imposing limits on total end-to-end server delay. A significant challenge in this environment is to make energy-aware decisions in the face of power-performance tradeoffs that vary among applications and from tier to tier. For instance, in a tier running a disk I/O intensive application, little or no performance is gained by speeding up the CPU.

Multi-tier servers are composed of groups of machines (tiers), where each tier handles some aspect of each request. For example, one tier might serve static content (e.g., static parts of web pages and images), another handles business logic, while a third performs database operations. The tiers are typically organized in a pipelined fashion: each stage sends sub-requests to the next stage (if any), performs some processing, and sends its response to the previous stage (or ultimately to the client), as depicted in Figure 1.2. Because the tasks of each tier are highly heterogeneous, so are their performance characteristics. The importance of multi-tier configurations lies in their ability to substantially increase the scalability of a service.

Our main approach to this problem is as follows. We first estimate the optimal settings based on a system model, and then, because that estimate may not be sufficiently precise, feedback control is used to converge to more optimal final settings. Neither a purely model-based, nor a purely

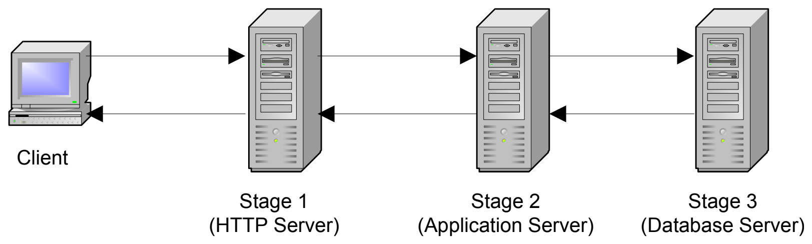


Figure 1.2: Example multi-tier server.

feedback-based approach using just service delay as the setpoint would give comparable results. The first one could not correct for inevitable modeling and estimation errors, while the second one would need to use heuristics, which might never achieve optimal power savings and might take a long time to converge.

Several different cluster models were considered. First, we assumed a basic model with a single machine in each tier (i.e., a *thin pipeline*). Since in this model each server must be awake to process every request, sleep states are not allowed and only CPU speed can be manipulated to manage power. Next, we extended this model to include prioritized service classes. Finally, we generalized the basic model to allow reconfigurable (i.e. variable-sized) clustered tiers, where sleep states become available in addition to variable CPU speeds. Using these models, we theoretically formulated and rigorously analyzed the stated problem as constrained optimization problems, and formulated mathematical solutions. This enabled the development of mathematically rigorous methods to dynamically tune all power states globally in the cluster, while satisfying total (end-to-end) request processing delay constraints.

Leveraging the analytical results, several algorithms and power management policies were designed in order to achieve the target performance with the least amount of power. In the most basic model, our policy merely effects optimized CPU speeds in all machines in the thin pipeline. The policy for the most complex model, however, keeps an optimized number of machines awake at optimized CPU speeds, and puts the remaining machines in low-power sleep states in order to balance energy savings against the ability to accommodate future bursts without violating performance requirements. The optimal balance is some function of both the future workload and the performance specifications of the user. Constructing such policies is non-trivial even in single-tier (i.e. non-tiered) systems because of several practical limitations. For example, these limitations include that transitioning between power states has both latency and energy overheads, and that the effects of switching performance states on service latency and power dissipation non-linearly depend on the workload. Our policies observe these limitations while considering multi-tier clusters, which added another dimension of complexity.

In order to experimentally validate and quantify the benefits of the discovered algorithms, we created several actual system implementations on our real testbeds. The basic model policy was validated on a small testbed of 3 laptops with variable-speed processors, while the more complex policies were verified on a larger testbed of 12 PCs, also with variable-speed processors and supporting multiple machine sleep states. Whereas much of the related literature is based on calculated or simulated quantification of benefits, in contrast, we took real measurements on actual hardware with algorithm implementations applicable in practical systems with realistic assumptions and limitations.

In summary, this dissertation includes the following main contributions:

- CPU speed-based energy minimization across thin resource pipelines whose tiers exhibit different power-performance tradeoffs, while maintaining end-to-end delay guarantees, with established superiority to both local policies that do not coordinate their actions across the machine pipeline and traditional heuristics that are not based on an analytical foundation. (Published [35].)

- Extension to accommodate multiple request classes with different performance requirements, in order to accommodate different classes of clients (users) or different service level agreements. (Published [36].)

- Generalization of the theory and algorithms to reconfigurable multi-tier clusters, where each tier consists of a variable number of machines, and therefore allows more complex energy management through utilizing machine sleep states.

- A spare capacity optimization algorithm for the general system model, assuming multiple sleep states with transition latencies and power dissipation progressively increasing and decreasing respectively with the depth of the state. Energy use is minimized subject to responsiveness constraints.

- Analysis of the cluster DPM policies with respect to their ability to tolerate thermal overload, machine failures, or other aberrant behavior.

- Finally, a major part of this project's contribution is the extensive experimental evaluation of the analytical solutions developed. A prototype implementation on a real experimental testbed running a complex multi-tier Web service (running workloads that are acceptable models of real-life commercial applications) was constructed. Physical measurements of total system power were taken on the main power supply lines.

1.3 Background

1.3.1 Escalation of the Power Problem

Traditionally, power management was mainly a concern in embedded, battery-operated systems, with the goal of maximizing battery life. Hence, algorithms focused on typical workloads of this domain (e.g., periodic, real-time task sets such as those in multimedia applications), and on taking advantage of special non-linear properties of battery technology, for example the phenomenon that batteries may recover some charge if left idle.

Due to heavy performance scaling of desktop PCs and servers, which led to increasingly power-hungry CPUs and other components, the energy use and heat generation of these systems became problematic. While for an individual desktop PC user the concerns may be more limited (e.g., annoying fan noise), for a server farm operator the rampant power demands could mean huge increases in costs. To see the trend, consider the exponential growth of the size of the Google server farm over the last several years. The total number of computers in 2000 was about 6,000 (at four sites) [34]. In 2001, at each site there were about 3,200 servers [10]. Finally, the latest published estimate places the total size of the farm in 2004 at 63,272 machines [50]. If we also observe that a *single site* in 2001 spent at least 1560 MWh of energy (a more than $100 K expense) a year [10], it becomes clear that the kind of growth expected demands advanced energy management solutions. In fact, providers like Google argue that operating costs like energy dominate over fixed costs like hardware. Providing enough hardware to accommodate bursts is often not the challenge: affording the operating costs of this hardware is [6].

The capacity of future networks will increase dramatically. As networked devices become ubiquitous, demands on Internet services will grow by orders of magnitude. In addition, the complexity of the services offered will increase further, requiring ever greater computational capacity for each client. This growth of demand will generate a similar growth in service provision, requiring large increases in both the number and the size of data centers and creating a huge energy management problem. Strong evidence of this growth is already seen: in 2005, already 1.2% of U.S. electricity use was attributed to servers and associated cooling [43]. While there are long-term initiatives in the direction of energy-proportional hardware designs [7], the reality for the foreseeable future is that idle machines do consume significant energy. Therefore DPM techniques will remain critical in attacking the energy problem.

1.3.2 Power Management Techniques

Several effective techniques exist for power management in servers today. The low-level design and implementation details of power states and transitions for modern systems are given in the ACPI specification [21].

On the granularity of entire computers, power savings can be attained by putting to sleep computers which are unnecessary to serve the current workload. Various sleep states, with different degrees of power savings and wakeup latency, are available. These system sleep states are defined by ACPI S-states. Sleep states are highly effective for managing energy in a server farm because they address the static power problem of idle servers. However, using sleep modes requires reconfigurable capability in the clusters (in each tier), including power-aware request scheduling algorithms in the cluster load balancers, ensuring consolidation of requests to only the necessary number of servers, as well as some remote wake-up capability (e.g. Wake-On-LAN, a technique allowing a special network packet to wake up a machine). Depending on the application's characteristics and resource needs, reconfigurability may be hard (or even impossible) to implement, which slows the adoption of this technique.

Within individual machines, the basic method is to put the CPU into a low-power state while it is idle. The different low-power CPU states are referred to as ACPI C-states. The benefit from these sleep states can be enhanced by short-term request batching [10], which delays processing of incoming requests for a short time so that several can be processed in one active cycle, allowing for slightly longer sleep periods. Additionally, CPU power can be lowered by the CPU throttling (i.e. clock gating or modulation) technique, which stops the CPU for very short periods amounting to a user-specified portion of time on average (e.g. 12.5–87.5% in increments of 12.5%). This provides for an approximately proportional decrease in CPU power dissipation, since it draws very little power when stopped. At the same time it also results in a proportional slow-down for CPU-bound tasks.

More recently, dynamic voltage scaling (DVS), also known as dynamic voltage and frequency scaling (DVFS), also became available in commercial server CPUs. This technique reduces power usage by slowing down the CPU (lowering its clock frequency) and reducing its core voltage. The discrete frequency-voltage combinations available in a specific CPU are known in ACPI terms as its performance states or P-states. The principal advantages of DVS are that the approximately linear slow-down caused by the frequency reduction allows approximately near cubic savings in power consumption, and that the transition between these different performance set-points has a fairly low overhead. The first advantage is in contrast with CPU throttling, which only achieves linear power reduction and therefore may not actually conserve energy. It is important to point out however, that as the voltage range available is shrinking, with many in industry suggesting that voltage cannot be reduced by more than about 25%, sleep modes are gaining additional importance in complementing DVS policies. Moreover, as the leakage power of CPUs is approaching 50% or more of their total power, leaving CPUs active with low utilization becomes extremely costly. Still, DVS retains its importance in applications where entering sleep modes might not be possible because the machine's other resources are actively used, even though its CPU capacity might not be required. In these cases DVS can help reduce leakage power, because leakage is proportional to the supply voltage affected by voltage scaling. Therefore, effective energy management requires selecting the proper balance of the available techniques, based on all requirements of the provided services.

Although effective energy management policies have been devised for simpler (single-server or single-tier cluster) architectures or for more restricted sets of available techniques, no straightforward extensions exist to make them applicable to multi-tier clusters supporting a combination of the aforementioned power management techniques while preserving response-time guarantees. This work presents the first comprehensive effort to address this more general optimization problem.

1.3.3 Distinction Between Power and Energy Management

It is very important to distinguish energy management and power management and their contrasting implications, especially since power and energy are sometimes loosely used in a synonymous meaning throughout the literature. While both power and energy management algorithms are based on the aforementioned power management techniques, they aim to solve different problems: the goal of power management is to reduce instantaneous power draw at any point in time, while energy management intends to reduce total energy consumed to perform a task. Note that power management does not necessarily improve energy efficiency—for instance, CPU throttling is effective in reducing instantaneous power, but at the same time it proportionally increases execution times, possibly resulting in *increased* energy use. Conversely, achieving better overall energy efficiency (for example, by consolidating work on fewer machines) may actually lead to increased peak power in parts of the system.

One major problem with overprovisioning in a data center is that the excess capacity of the machines themselves must be matched by similar excess capacity in power delivery and cooling infrastructure as well. This translates to serious initial investment costs. To make matters worse, computers are traditionally rated at their worst-case power draw, which is hardly realistic in powerful server-class machines running real applications—in fact, normally only specialized workloads specifically designed to stress hardware components to their power limits (such as the Intel Maximum Power Program or `P4MAX` [48]) are able to approach worst-case power dissipation. Hence, great investment saving potential exists in provisioning power delivery and cooling infrastructure for the actual workload's power draw instead of the machines' ratings. This in turn requires proactive power management with a policy that prevents exceeding infrastructural limits, while having as little performance impact as possible. Such DPM is referred to as power capping or power budget management, or, if performed at the level of multiple machines, as power shifting or ensemble-level power management [47, 70].

Energy management, on the other hand, seeks to reduce operational costs by reducing the energy use of the (overprovisioned) infrastructure. By eliminating part of the energy consumption from the computers, the energy needs of the cooling infrastructure automatically also diminish, adding to the significance of energy management. Similarly as in power management, the main requirement for energy management policies is having an acceptable performance impact. However, since in this case there are two metrics describing the performance of a given policy, namely how much energy the machine uses during the execution of a task (energy, E) and the execution time of the task (delay, D), a single derived metric must be defined to allow comparing policies. Such metrics are the widely used $E \times D$ energy-delay product [26], the $E \times D^2$ product, which is more appropriate for microarchitectural evaluations due to its advantage of being voltage independent [12], and other similar metrics with different weighing of energy and performance. However, in the context of real-time Internet services, the focus of this dissertation, these metrics are not directly applicable because instead of the actual service delay of each request, we are interested in whether deadlines were met. A good metric of energy efficiency analogous with the energy-delay product in this domain is $\text{power}/\text{throughput}^2$, or equivalently, the product of energy per successful request and the inverse of throughput. Further, such metrics are not sufficient since for throughput-oriented workloads, the energy spent per successful request or over some time period is a very important metric in itself due to its direct correspondence to cost savings.

Finally, it should be noted that, despite the seemingly contradicting goals of each, power and energy management can be combined to gain the benefits of both strategies. Since power management policies determine power states that are safe with respect to the auxiliary infrastructure, those power states can be viewed as a limiting envelope for energy management policies, which are then allowed to direct hardware into states within these limits. Nonetheless, joint optimization of power and energy management is non-trivial and it is out of the scope of this dissertation.

1.4 Related Work

1.4.1 Traditional Power Management

Prior to servers, DPM policy investigation efforts were mainly directed at embedded and desktop systems. For instance, earlier DVS research primarily addressed standalone, battery-operated, embedded mobile devices, which still remains an active research area (see survey by Unsal and Koren [83]). Subsequently, DVS was also exploited in desktop systems. Flautner et al. proposed a DVS policy that manages energy with the focus of maintaining the quality of interactive performance of applications [24]. Their algorithm is based on interactive-episode detection and it was validated using trace-driven simulation showing 75% CPU energy savings. However, no experiments on actual hardware were performed or system energy savings measured. DPM using sleep states, based on complex system modeling and learning, was investigated by Chung et al. [16, 17]. These works were primarily concerned with power-manageable devices within a computer (such as hard disk drives), addressing device sleep states. With similar focus, AbouGhazaleh et al. addressed DPM for wireless network cards by increasing sleep durations and utilizing deeper sleep states, evaluating the power-performance tradeoffs [2]. However, their methodology was highly specific to this domain and not directly applicable to server DPM.

In their survey of DPM design aspects, Benini et al. address the modeling of power-managed systems and present an overview of techniques and policies [8]. The policies presented target individual machines as opposed to multi-tier servers or clusters. Irani and Pruhs provide an overview of the algorithmic theory relating to these methods [38]. They discuss the Speed Scaling with Power-Down (SS-PD) scheduling problem, which is similar to our central problem of finding the optimal combination of power states, including CPU performance states and system sleep states, and note that it is unknown whether the offline version of this problem (for a defined task set) is NP-hard. However, the theories presented do not address server workloads.

1.4.2 Energy-aware Real-time Systems

Families of DVS algorithms integrated with an RTOS scheduler were proposed for periodic hard real-time task sets in several papers [5, 55, 66]. DVS algorithms assuming similar task sets and a continuous frequency setting model were presented by Zhu et al. [91] for multi-processors. More recently, Zhu and Mueller presented a feedback control-based DVS framework with EDF scheduling in hard real-time systems [92]. A soft real-time energy-efficient scheduler for periodic tasks in embedded systems is presented by Yuan and Nahrstedt [88]. It employs a DVS algorithm similar to the most aggressive one proposed by Pillai and Shin [66], 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. Simunic et al. devise a DVS algorithm for portable systems, which relies on offline workload characterization and probabilistic online detection of arrival or service rate changes [76]. Other DVS algorithms targeted at soft real-time systems predict near-future processing requirements (load) based on past history. `PAST`, 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 [85]. Govil et al. presented and evaluated other prediction schemes, including `AGED_AVERAGES`, which uses a moving average of past samples with geometric decay, and `PEAK`, which expects short peaks in load [28]. The latter was shown to outperform `PAST`.

Recently, Varma et al. applied control theory to predict the future workload to guide their DPM policy [84]. The authors designed an algorithm, `nqPID`, that outperforms the aforementioned algorithms, while achieving performance that is less dependent on parameter tuning. However, their results were validated only by simulation against a periodic task model. Feedback control techniques were also employed with DVS by Lu et al., in order to save energy while guaranteeing frame rate in multimedia workloads [51, 52]. Their load predictions are calculated based on a queueing model. In their first paper [51], they report similar energy savings as Simunic et al. [76], but with reduced computation and improved quality of service. The authors also used a dead-zone control method to provide strong real-time guarantees without requiring prior workload knowledge like many other multimedia DVS schemes [52]. However, since it controls buffer levels, it is not applicable to systems that do not tolerate buffering latency (e.g., Internet servers).

None of the efforts mentioned above address energy minimization in resource pipelines where savings at different tiers can be traded off against each other and where end-to-end latency requirements must be achieved. Ironically, most Internet server installations today are multi-tier, giving rise to pipelines of processing stages and hence a much richer space for energy optimization across the entire pipeline. Slightly related is the work by Kang et al., proposing a power-aware scheduler for distributed systems with hard real-time end-to-end delay constraints [41]. 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, which is not true of Web services.

1.4.3 Server Power and Energy Management

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 have made the case by pointing out negative environmental effects, high operating costs, power density problems, and expensive infrastructure requirements of large server sites [10, 11, 20]. Lefurgy et al. present an general overview of the motivation, potential, and mechanisms related to energy management in servers and clusters [46]. They specifically point out the power advantage of heterogeneous configurations such as multi-tier servers, which our work takes advantage of.

Several papers address DVS in standalone servers and server clusters. Elnozahy et al. presented a soft real-time feedback control-based DVS policy combined with request batching [20]. Their simulation results showed up to 42% savings of CPU energy in a standalone web server, when 90% of the response times were within the target latency. They did not, however, validate their results by implementation in a real system, nor did they measure total *system* energy savings. A real DVS policy was implemented by Sharma et al. for standalone web servers with multiple QoS service classes, each of which had soft real-time deadlines [75]. The system builds on a proven schedulability bound for aperiodic tasks, due to which it can sustain less than 2% deadline miss ratio. However, as is common to most previous work, the system is restricted to single-tier servers.

Reconfigurable clusters have already been proposed in the literature as a means of enabling the exploitation of the powered-down system state. Elnozahy et al. present and evaluate by simulation five different power management schemes for single-tier server clusters [19]. 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, which underlines the importance of our approach of optimizing all power states simultaneously. In their paper on power-aware request distribution (PARD), Rajamani and Lefurgy identify key system and workload factors that affect energy management in reconfigurable clusters [69]. Their methodology was similar to ours, but their system models are too simple to capture the complexity of service pipelines (i.e., multi-tier clusters) or a richer set of sleep and performance states. Heath et al. designed an energy management policy in heterogeneous clusters based on extensive system modeling [32], having similar limitations.

Power management of memory and storage devices in servers has also been addressed in the literature. Lebeck et al. [45] evaluate by simulation power-aware page allocation, which results in significant improvement in memory energy efficiency. Tolentino et al. [80] propose feedback control-based memory power management, relying on a modified kernel to achieve substantial system energy savings with minor performance loss. Their work also highlights the importance of memory power as a function of the amount of memory per processor. Joint power management of memory and disks, observing cache size and idle periods, is explored by Cai et al. [13]. Unlike in their work, which relies on disk sleep states to manage disk power, Gurumurthi et al. [29] propose variable-speed disk drives, which have the advantage that they do not incur very costly wakeup latencies. Our work focuses on CPU and system-level power management techniques, and we leave it for future work to integrate memory and storage power management into our solutions.

Recently, virtual machine (VM) architectures have been used as a platform to implement global power management policies [63]. Such architectures make it possible to expose individual DVS states to each guest VM, whereas the virtual machine monitor can choose the actual hardware state, taking the guest VMs' "soft" states as hints. This can provide a framework to the coordination of individual policies across multiple levels of hierarchy (from within a CPU to among racks of machines). Another advantage is the ability to transparently migrate VMs based on power management decisions. However, the framework itself does not address optimal coordination in multi-tier systems with end-to-end performance constraints, which is the focus of our work. On the other hand, our policies could be implemented on such a VM framework if desired. Other works on virtualization address multi-tier application performance [65], but not in conjunction with power management.

Complementary to our research are the efforts directed at dealing with cluster-level power provisioning and thermal issues. The former problem is addressed by several researchers. Lefurgy et al. introduce power capping, a high-level power management technique that allows constraining the peak system power of a single server, supported by actual power measurement in hardware [47]. The underlying DPM technique directed by the power capping logic can be either CPU throttling or DVS. The ability to limit peak power to a *power budget* allows under-provisioning of power supplies and delivery infrastructure for cost savings. Even further savings are possible if the same idea is taken to the cluster (or rack, enclosure etc.) level, where the power peaks of individual servers tend to average out, yielding a smoother total power profile. Therefore, constraining the whole group of servers to an aggregate power budget ideally demands smaller performance sacrifice, assuming the power budget is optimally distributed among the cluster nodes. The dynamic distribution of the total power budget is termed power shifting, and it was investigated independently by Lefurgy et al. and Ranganathan et al. [70]. Fan et al. employed power modeling to attack data center-level power provisioning inefficiencies, and also evaluated the additional gains achievable by data center-level power capping in very large scale clusters running real live workloads [22]. On such a large scale, they find modest improvements from power capping compared to the improvement from power modeling only. Finally, several papers explore cluster-level thermal management [33, 58, 59, 74].

These results may be combined with our energy management policy, provided care is taken to resolve conflicting power state directions from the separate algorithms. For example, states selected by the power capping algorithm should not be unconditionally set in hardware but only if the current state dictated by other policies would exceed the power of those selected states.

In summary, the research described in this dissertation was aimed at covering the gap in energy management literature in the area of resource pipelines with contemporary power management features. Energy management in multi-stage execution systems represents a very important, surprisingly overlooked, topic that significantly impacts the ability of realistic server farms to reduce their operational costs. This topic is especially challenging when the server farm must meet end-to-end latency requirements of different traffic classes. To my knowledge, this work undertook the first comprehensive systems-oriented research effort with an extensive experimental component that addresses energy management policies in multi-tier (i.e. pipeline) latency-sensitive server farms. A significant departure of my work from much of the previous work on latency-sensitive services is also that we are not restricted to periodic task models commonly assumed in real-time systems research. Hence, my results have a much broader impact and applicability in the Internet service application domain. If adopted, my algorithms can lead to additional cost reduction in the operation of large server clusters, enabling deployment of more server capacity in a growing number of data centers.

Chapter 2

DVS for Multi-Tier Servers

2.1 Introduction

Complex web services are commonly realized by *multi-tier* web server systems in order to functionally 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 transferring a trivial amount of data.

In this chapter, 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 [10, 20]. 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 [11]. 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 aggressive power saving policy.

Previous research has studied DVS in a single web server or a single-tier web server cluster with performance guarantees [10, 11, 19, 20, 75]. However, straightforward extensions of these are not sufficient to reasonably optimize power in server pipelines. In a pipeline, the end-to-end delay is composed of highly variable stage delays, therefore independent stage delay control achieved by single-server algorithms cannot be effective in controlling the end-to-end delay. Further, since such independent DVS algorithms have no concept of end-to-end delay, their power optimization cannot be optimal because they lack the proper solution constraint. To our knowledge, no work has been done to address DVS in multi-tier web servers with end-to-end delay constraints.

In this chapter, 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 delay model. The policy is designed for realistic CPUs with discrete DVS frequency settings. Decisions on frequency adjustments are made on each stage locally, governed by a decentralized self-coordination scheme. The self-coordination ensures that each stage can individually compute the globally optimal solution and apply it to itself, without the need for a central entity. We also present the formulation of the problem of determining the globally optimal DVS policy for such systems. We show experimental results from our prototype implementation confirming that our solution is efficient and stable. We experimentally verify that the proposed coordinated scheme outperforms uncoordinated single-machine power management. In particular, we compare it to the default Linux power management as a baseline. Additional energy savings in excess of 30% are observed.

It should be observed that the DVS-based approach explored in this chapter does not exclude the usage of other power saving schemes. Typically, different schemes would be employed at different time-scales. For example, relatively long-term load fluctuation patterns (such as day/night fluctuations) can be accommodated by turning machines on or off to match the anticipated load as proposed in earlier literature [19]. In such an on/off scheme, extra capacity would typically be left on each stage to accommodate shorter-term bursts. Hence, given a particular configuration of machines that are on, the protocol described in this chapter can be used to determine their power-optimal DVS settings. Consequently, energy savings are increased by taking advantage of load fluctuations on shorter time-scales. Moreover, if machines in each tier are roughly load-balanced, their actions would typically be symmetric within the tier. Hence, in order to investigate coordinated DVS schemes across the pipeline, it is enough to consider a pipeline of one machine per stage. In thicker pipelines, assuming homogeneous servers with appropriate load-balancing in each stage, all machines within a stage will likely behave identically. With that in mind, we focus in this chapter on deriving and implementing coordinated power-optimal DVS schemes for thin pipelines (i.e., those with one machine per stage) that respect end-to-end latency constraints.

The main contributions of this chapter are the theoretical optimization of the energy efficiency (i.e., power consumption subject to latency constraints) of soft real-time multi-tier web servers, and the detailed case study and evaluation of our prototype testbed implementation. This work appeared in [35].

The rest of the chapter is organized as follows: section 2.2 presents the general system architecture and DVS solution; section 2.3 details the implementation; and performance evaluation is presented in section 2.4. The chapter concludes with section 2.5.

2.2 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.

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 (i.e., the target value of the controlled variable) being a pre-configured end-to-end delay value. Note that by end-to-end delay we refer to the total latency on the server side, which does not include network latencies between the server and the client. 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.

2.2.1 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. Observe that it could be argued that the reverse is also true. Namely, if the I/O blocking delay, D_i^{block}, is very large and if the disk is the bottleneck, the system will needlessly try to increase CPU speed when it is overloaded. This will decrease power savings without affecting the actual bottleneck delay. Fortunately, this situation is easy to prevent at run-time by disallowing machines with a low CPU utilization from speeding up their CPU. Hence, unless the CPU is the bottleneck at some machine, power savings will not be needlessly impaired. The algorithm described below adopts this restriction.

2.2.2 Simple Dynamic Voltage Scaling

In a first attempt to design an optimal feedback-based DVS scheme in terms of energy savings, we only assume that the CPU delay D_i^{CPU} at stage i is a convex function $g(U_i)$ 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 $d^2g(U_i)/dU_i^2$ is positive. For example, given a Poisson arrival process and exponentially distributed execution times, we know from queueing theory that $D_i^{\text{CPU}} = T_i/(1-U_i)$, where T_i is a constant. Hence, $d^2g(U_i)/dU_i^2 = 2T_i/(1-U_i)^3$, which is positive for $U_i < 1$.

This assumption is generally true of busy servers, and it is intuitively supported by the observation that real-life web servers tend to increasingly saturate when operating at higher CPU utilizations, leading to steeply increasing latencies. When the algorithm described below is applied to workloads for which this assumption does not hold, the resulting system will still maintain the end-to-end delay within constraints, albeit with poorer performance and energy efficiency. For instance, the assumption may be false in completely sequential workloads (i.e., in workloads with no parallel tasks), where CPU utilization has no effect on delay. However, typical web server workloads are highly concurrent in nature because of the large number of independent clients.

The convexity assumption leads to a simple set of rules for adjusting CPU speed to globally 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' 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.

2.2.3 Optimality Conditions

Let us assume that the power consumption P_i of stage i is a general function of CPU utilization:

$$P_i = p_i(U_i) \tag{2.1}$$

Second, assume that the delay D_i^{CPU} of a stage i is approximately related to its utilization U_i by the queueing-theoretic equation:

$$D_i^{\text{CPU}} = \frac{T_i}{1 - U_i} \tag{2.2}$$

where T_i is the mean service time of each stage. In reality, this equation is not exact, since studies suggest that web workloads in general follow a heavy-tailed distribution [87]. Unfortunately, queueing models with heavy-tailed interarrival and service times are very difficult to analyze [23]. However, our model provides a reasonable approximation for deriving a practical optimality condition for typical multi-tier web workloads. The intuition is that with similar distributions in each tier, the relative estimation error diminishes, and the resulting deviation from the optimal system state is insignificant compared to the deviation arising from the inherent discreteness of the system (i.e., small number of available frequencies). Our experimental results also support this intuition by showing improved performance with the proposed model. For other workloads where the model is inappropriate, the same analysis can be carried out with a different delay approximation equation.

Summing over the entire pipeline, the total power consumption P of the N-stage system can be expressed by:

$$P = \sum_{i=1}^{N} p_i(U_i) \tag{2.3}$$

Our objective is to minimize that power consumption subject to the constraint $\sum_{i=1}^{N} D_i^{\text{CPU}} + D_i^{\text{block}} \leq L$, where L is the maximum desired latency. Taking the equality condition as the limiting case, and substituting from Equation (2.2), this constraint can be rewritten as:

$$\sum_{i=1}^{N} \frac{T_i}{1-U_i} = K \tag{2.4}$$

where $K = L - \sum_{i=1}^{N} D_i^{\text{block}}$, 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 add the Lagrange multiplier, λ, which yields:

$$L(U_i, \lambda) = \sum_{i=1}^{N} p_i(U_i) + \lambda \left(\sum_{i=1}^{N} \frac{T_i}{1-U_i} - K \right) \tag{2.5}$$

Using the Kuhn-Tucker Theorem, we can get:

$$\frac{\partial L}{\partial \lambda} = 0 \tag{2.6}$$

and for each i:

$$\frac{\partial L}{\partial U_i} = 0 \tag{2.7}$$

The solution of Equation (2.6) is exactly the constraint given by Equation (2.4), thereby ensuring that it is always satisfied. From Equation (2.7), we get:

$$p_i'(U_i) + \lambda \frac{T_i}{(1-U_i)^2} = 0, \tag{2.8}$$

which implies:

$$\frac{p_i'(U_i)(1-U_i)^2}{T_i} = -\lambda. \tag{2.9}$$

Hence, the optimal solution to the general power minimization problem is the following equalizing optimality condition:

$$\frac{p_1'(U_1)(1-U_1)^2}{T_1} = \cdots = \frac{p_N'(U_N)(1-U_N)^2}{T_N} \tag{2.10}$$

Next, to arrive at a specific solution, let us consider the following equation between system power consumption and CPU frequency:

$$P_i = A_i f_i^n + B_i, \tag{2.11}$$

where A_i and B_i are constants. The general rule of thumb with CMOS technology is that $P \propto V^2 f \propto f^3$, that is, power is proportional to the cube of clock frequency. The rationale is that raising the clock frequency also necessitates increasing the voltage. In reality, however, $f \propto V$ is a simplification, hence our more general expression. This assumption is accurately satisfied in realistic systems, with n ranging between 2.5 and 3. The same power model with $n = 3$ is assumed for analysis by Elnozahy et al. [19]. In general, it is possible to obtain the exponent n and constants A_i and B_i by curve fitting against empirical measurements obtained from profiling the system.

If the workload arrival rate at stage i is λ_i cycles/s, the utilization U_i of that processor is λ_i/f_i, where f_i is the service rate or frequency in cycles/s. Equivalently, $f_i = \lambda_i/U_i$. Substituting in Equation (2.11) yields the specific power-utilization function:

$$P_i = p_i(U_i) = A_i \frac{\lambda_i^n}{U_i^n} + B_i \tag{2.12}$$

Note that the power consumption of a tier of machines has an indirect dependence on the other tiers: the behavior of the other tiers affects the utilization of the mentioned tier, which in turn affects its power consumption. Our model is simple in the sense that it does not contain a prediction component to capture how tiers affect each other's utilization. However, the model is fairly accurate in representing the true power consumption based on the observed utilization, which does reflect the inter-tier dependencies.

Substituting in the general solution given in Equation (2.10), we get:

$$\frac{p_i'(U_i)(1-U_i)^2}{T_i} = \frac{-nA_i\lambda_i^n U_i^{-(n+1)}(1-U_i)^2}{T_i} \tag{2.13}$$

This finally leads us to the following equalizing optimality condition, which provides the optimal solution to our specific power minimization problem:

$$W_1 H(U_1) = W_2 H(U_2) = \cdots = W_N H(U_N) \tag{2.14}$$

where W_i is a weight, which, after simplifying Equation (2.14) by $(-n)$, is given by:

$$W_i = \frac{A_i\lambda_i^n}{T_i}$$

and H is a transformation defined as:

$$H(U_i) = \frac{(1-U_i)^2}{U_i^{n+1}}.$$

To minimize power consumption across the pipeline subject to the end-to-end delay constraint, a feedback loop is added to equalize the weighted transformed utilizations of all stages such that it satisfies Equation (2.14). Utilization is manipulated by changing the CPU frequency settings.

2.2.4 Improved Algorithm with Miss Ratio Considerations

To converge on the condition expressed in Equation (2.14), average local stage CPU utilization 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 distributed DVS algorithm on each machine computes the weighted transformed utilization, $W_i H(U_i)$ 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 $\alpha_{\mathrm{hi}} < 1$ of the actual deadline L. We denote this conditional probability by $\Pr(D[k+1] > L | D[k] < \alpha_{\mathrm{hi}} L)$, 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 $\Pr(D[k+1] > L | D[k] < \alpha_{\mathrm{hi}} L) \leq r$. Given an analytically derived or empirically measured conditional probability function, the equality condition, $\Pr(D[k+1] > L | D[k] < \alpha_{\mathrm{hi}} L) = r$ 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 $D > \alpha_{\mathrm{hi}} L$ (overload), machine i with $\min_i\{W_i H(U_i)\}$ steps up its frequency to the next higher discrete setting. Note that since H is monotonically decreasing in the range of U_i, this will increase the weighted transformed utilization, effectively balancing it as desired.

- If $D < \alpha_{\mathrm{lo}} L$ (underutilization), machine i with $\max_i\{W_i H(U_i)\}$ steps down its frequency to the next lower discrete setting (where $\alpha_{\mathrm{lo}} < \alpha_{\mathrm{hi}}$). Note that by symmetry, this will decrease the weighted transformed utilization, again meaning a balancing action.

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. By applying these rules simultaneously on each machine in the pipeline, we arrive at a distributed algorithm that converges on the globally optimal solution. Note that if W_i are equal for all stages (such as in a homogeneous system with perfectly balanced load over the whole pipeline), the algorithm reduces to the one described in section 2.2.2.

Finally, observe that while we described the algorithm for a single class of clients with the same deadline, 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 $\exists i : D_i > \alpha_{\text{hi}} L_i$ (overload), machine i with $\min_i\{W_i H(U_i)\}$ steps up its frequency.
- Else, if $\forall i : D_i < \alpha_{\text{lo}} L_i$ (underutilization), machine i with $\max_i\{W_i H(U_i)\}$ steps down its frequency (where $\alpha_{\text{lo}} < \alpha_{\text{hi}}$).

The first rule of the aforementioned algorithm can be further improved by excluding machines with a low CPU utilization (i.e., $U_i < U_{\text{lo}}$) from stepping up their speed. As mentioned earlier, this situation might arise if the disk was the true bottleneck making the CPU speed irrelevant. With this improvement, CPU speed is adjusted only if the adjustment is likely to affect delay. The resulting algorithm has better energy savings in systems dominated with disk bottlenecks.

2.2.5 Feedback Control Model

Having defined our DVS algorithm, we proceed to describe and analyze the design of the feedback control system we chose to implement. Note that our goal here is not to design an optimal controller, but rather to demonstrate the practical usefulness of our algorithm. Hence, we develop a simple yet effective controller, focusing on the design limitations of our target systems, such as the small number of available CPU frequencies. Other control-theoretic models of absolute delay control loops with different assumptions have been presented by Sha et al. [73].

The overall system is modeled by a discrete nonlinear feedback control loop with a deadzone. The input of the loop is the pair of threshold parameters α_{lo} and α_{hi}, which define the controller deadzone. The error signal received by the controller is then the difference of the measured end-to-end delay (feedback) and the center of this deadzone. Given the error signal, the controller determines the DVS adjustment as follows. If the error falls in the deadzone, no adjustment is made. If the error is greater, then the CPU frequency of one stage is adjusted by one stepping, as selected by the algorithm given in section 2.2.4. Driven by the current CPU frequencies and the offered load, the multi-tier web server (controlled system) processes requests with a certain end-to-end latency, which is sampled, averaged, and fed back to compose the aforementioned error signal.

Since the available CPU frequencies in the controlled system are limited, the presented saturated controller cannot become unstable. Limit cycles, where the system would only operate at the lowest or highest CPU frequency, are also impossible (assuming that other levels are available) because the frequencies are always adjusted by a single stepping only. However, analyzing stability is still worthwhile for identifying the presence of harmful oscillation in the system. To assure that our controller does not cause oscillatory behavior, it is sufficient to show that under constant offered load no frequency decrease can lead to a frequency increase. This means that whenever the maximum end-to-end delay falls below the lower threshold, the average increase in delay caused by stepping down any CPU frequency should be smaller that the deadzone. Otherwise, the frequency decrease could drive the maximum end-to-end delay beyond the high threshold, which in turn would trigger a frequency increase, resulting in undesired oscillation in the average delay. This deadzone constraint can more formally be expressed from the delay equation (Equation (2.2)) as follows:

$$T_i\left(1-\frac{\lambda_i}{f_i^{<j-1>}}\right)^{-1} - T_i\left(1-\frac{\lambda_i}{f_i^{<j>}}\right)^{-1} \leq Z \tag{2.15}$$

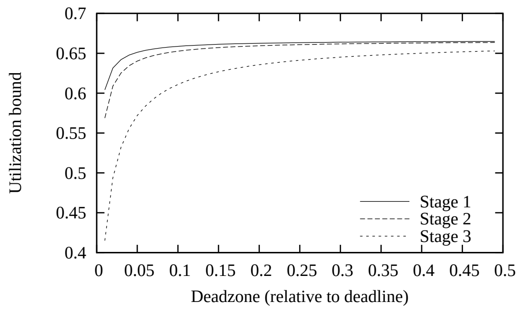
where $f_i^{<j>}$ is the j-th CPU frequency setting in stage i and Z is the deadzone (relative to the deadline). In order to bound the average delay increase, we observe that for any DVS architecture, we can find frequency bounding parameters α and β as follows:

$$\exists \alpha, \beta : \forall j : f_i^{<\Delta j>} \leq \alpha f_i^{<j>} + \beta < f_i^{<j>} \quad \text{s.t.} \quad 0 \leq \alpha < 1 \text{ and } 0 \leq \beta,$$

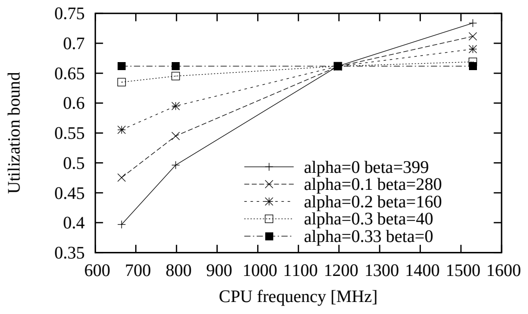
where $f_i^{<\Delta j>} = f_i^{<j>} - f_i^{<j-1>}$. Using these parameters, the following equation satisfies the deadzone constraint given by Equation (2.15):

$$T_i \left(1 - \frac{\lambda_i}{(1-\alpha) f_i^{<j>} - \beta}\right)^{-1} - T_i \left(1 - \frac{\lambda_i}{f_i^{<j>}}\right)^{-1} = Z. \tag{2.16}$$

Solving Equation (2.16) for λ_i, we get the arrival rate λ_i^B, for which the maximal delay increase arising from some frequency adjustment equals Z. Since the delay increase function on the left-hand side of Equation (2.16) monotonically increases in λ_i, Equation (2.15) is satisfied for all $\lambda_i \leq \lambda_i^B$. From this, $\lambda_i^B / f_i^{<j>}$ yields a utilization bound for each frequency setting j, below which the deadzone constraint is satisfied.



(a) Varying the deadzone range.



(b) Different frequency bounding parameters.

Figure 2.1: Effect of parameters on the utilization bound for stable CPU frequency reduction [35, Fig. 1].

In Figure 2.1, we calculate (based on profiled parameter values) the aforementioned utilization bound for various parameters. Figure 2.1(a) shows the results for different deadzone ranges. We can see from the graph that a deadzone range of 0.3, for instance, yields a utilization bound of 64%. Figure 2.1(b) demonstrates how different feasible choices of the frequency bounding parameters affect the utilization bound. As the graph shows, some parameter values give tighter bounds than others. For example, for the CPUs used in our experiments, $\alpha = 0.33$ and $\beta = 0$ give the tightest overall bound, 66%, over the whole frequency range. Together, the graphs show that a deadzone range of 0.3 will prevent frequency oscillation as long as the utilization of the CPU being stepped down is lower than 64%. This condition is likely to be satisfied, as generally the most underutilized stage's capacity is decreased to save power. The conclusion is that the parameter α_{lo} should be selected based on the appropriate deadzone range that prevents oscillatory behavior.

Note that 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 available DVS frequency settings for a CPU is usually small, and multiple-step actuation could easily overreact to load variations. Having said so, the controller gain can be altered by changing the sampling period. Smaller periods result in higher gain since actuation is more frequent (while the actuation step remains the same). It is important to choose a sampling period that does not cause stability problems. Specifically, to avoid unnecessary oscillation, it is sufficient to ensure that the sampling period is long enough that the effect of the last frequency adjustment appears in the newly measured end-to-end delay. Therefore, a suitable sampling period should be inferred from the expected workload arrival rate and target latencies.

2.3 Implementation

2.3.1 Infrastructure Overview

In designing our implementation structure, 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 software that we want to leave intact, and extensible because it needs not be modified to accommodate a new 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 [78], for example, does provide such hooks.

Our prototype three-tier platform is composed of three laptop computers with Mobile AMD Athlon XP DVS-capable processors. The processors have discrete frequency levels ranging from

532 MHz to 1529 MHz, with settling time specified as 100 μs. Each computer runs Linux 2.6. We implemented two separate three-tier web server systems on this platform: a Synthetic system and a TPC-W [82] system. In the Synthetic system, the first two computers run Apache 1.3 as an HTTP front-end and as an application server, respectively, while the third computer runs the MySQL 4.0 database server [62]. The TPC-W system consists of the first computer running Apache 1.3 as HTTP front-end and image server, the second running JBoss 3.2 [71] as an application server, and the third running MySQL 4.0. As for the actual TPC-W software, we adopted a J2EE-based implementation of the TPC-W 1.8 benchmark that uses contemporary technologies such as entity EJBs with container managed persistence for best performance [81]. We have not been able to find a readily available client for this server. Thus, on the TPC-W client side, we used a compliant Remote Browser Emulator from a separate source [64]. Several modifications were necessary to both the server and the client to make them interoperable. Since the client was not capable of accepting browser cookies containing the session identifiers, we modified the server to support session tracking using URL encoding. Further, we modified the client's URI fragments and patterns, as well as resolved interface-level incompatibilities. Our DVS policy is implemented independently as a standalone daemon to be started on all servers. The daemons on each stage establish TCP connections with the previous and next stages. Once they form a pipeline, they start self-coordination and control of the local CPU frequency.

2.3.2 Measurements and Actuation

Measuring end-to-end delay in practice is a challenge. True end-to-end delay could only be measured with kernel support. Alternatively, measuring delay in user space is a flexible yet imprecise solution. Since Linux does not yet provide the necessary timestamping support for TCP packets, we chose the user-space solution, which 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. In many cases, the implementation must also support separate request classes with different deadlines. For instance, the TPC-W specification defines several "web interactions" with different delay constraints. Hence, our extension module also provides a new command, which allows deadline specifications for separate request classes identified by regular expressions against the request URI. Every sample thus consists of a delay measurement and a corresponding deadline. The DVS daemon running on the first stage provides a local (System V) Message Queue IPC interface to gather these samples. The measured end-to-end delay statistics are then periodically sent to all subsequent remote stages via TCP/IP messages.

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 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.

The DVS algorithm is invoked at the end of each 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 statistics, hence they can solve the current global DVS problem instance independently. When a stage's solution indicates that one of the rules need to be activated on itself, then that stage adjusts its CPU frequency (i.e., steps it up or down to the next discrete setting). The actual CPU speed setting is implemented by invoking the standard `userspace` frequency scaling governor of the Linux CPUFreq device driver.

2.3.3 Parameter Selection

When implementing our DVS algorithm, we must make an appropriate choice of the upper and lower delay thresholds, α_{hi} and α_{lo}, described in section 2.2.4. Violations of these thresholds trigger reactions to overload and underutilization respectively. As mentioned in section 2.2.4, the upper threshold is chosen such that $\Pr(D[k+1] > L | D[k] < \alpha_{\text{hi}} L) = r$, 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 2.2 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 2.2, we see that a threshold of $\alpha_{\text{hi}} = 0.7$ guarantees that the maximum miss ratio will remain below 5%. The lower threshold is then selected by using the analysis in section 2.2.5, where it was shown that 0.3 is an appropriate deadzone range for our workload. Hence, we use 0.4 as the lower relative delay threshold, which yields a deadzone range of $\alpha_{\text{hi}} - \alpha_{\text{lo}} = 0.3$.

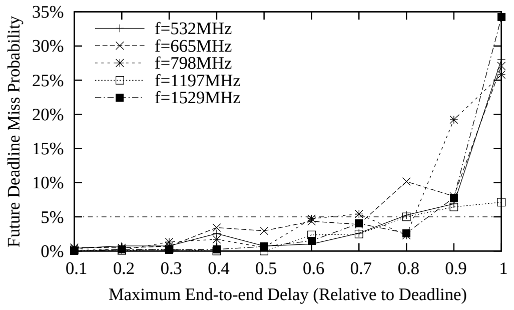


Figure 2.2: Choosing the upper delay threshold [35, Fig. 2].

In settings where the workload is not known at design time, the parameters must be determined online. The analysis presented above simply has to be automated by sampling the delays in the live system. The high threshold can then be computed from the estimated long-term conditional probability as shown above. Finally, an adaptive online algorithm can determine the minimal deadzone by measuring oscillations and adjusting the deadzone to avoid them, leading to an appropriate low threshold for the workload.

2.3.4 Control Performance

As we mentioned in section 2.2.5, when choosing the sampling period, one major concern is to limit controller overshoot as much as possible. A conflicting concern is to get a short response time (rise time) when the system starts violating the performance requirements. We select a short sampling period during overload for the sake of high responsiveness to deadline misses. Our choice is $T = 200\,\text{ms}$ because, with the expected throughput of our workload, 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. Since in modern systems the power consumption of the network interfaces is not significant compared to the main components (CPUs, memories, disks), this communication overhead has very low effect on total system power. Also, since at most one frequency adjustment occurs in every sampling period, our $100\,\mu\text{s}$ per period frequency transition overhead stays negligible even with this short period. 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 ranges from 4 to 10 seconds, depending on the deadlines of the workload. 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. Therefore, stability problems are avoided, since the longer sampling period ensures that the effects of the previous frequency adjustment are seen before further adjustments. This much longer sampling period does not mean, however, that the system becomes unresponsive to deadline misses during underload, because it is implemented in terms of the short periods by aggregating their samples. Hence, if deadline misses occur in any short period, the controller identifies an overload situation, which results in immediate corrective action at the end of that short period. Our results indicate that this yields a good compromise between soft real-time performance and energy savings.

Another design feature that impacts control 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, discussed above), 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 underutilized or overloaded). 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 by substituting the expected value of system reaction for the actuation step size. The implication is that if the impact is unacceptable, then a centralized design might be preferable.

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 one sampling period since the last broadcast of end-to-end delay. Since we choose the sampling period to be small (compared to end-to-end deadlines) for fast system reaction, we argue that this delay is acceptable.

2.4 Evaluation

We evaluate two versions of our algorithm (the Feedback DVS version, implementing the naive policy, and the Weighted Feedback DVS version, implementing the optimization-based policy) by comparing them to a Baseline and an Independent DVS scheme. For the Baseline, we set the CPU frequency to the maximum on all stages. Let us point out that this does not necessarily mean that the CPUs will constantly run at that frequency. Linux (together with 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 and puts it into a low-power state. Thus, our Baseline policy already performs such power management. For the Independent DVS scheme, we control the CPU frequencies independently, running an implementation of the PAST [85] DVS algorithm on each stage. All DVS algorithms are run on top of the Baseline policy. Thus, our reported power savings are those above the aforementioned policy.

The rationalization of our choice of comparison policies is that no other reasonable and applicable algorithm exists in previous literature to compare with. As we discussed in section 1.4, reasonable previous solutions to multi-stage power optimization with real-time constraints are not applicable to aperiodic workloads with unknown worst-case execution times. On the other hand, algorithms devised for standalone servers or server clusters cannot reasonably satisfy end-to-end delay constraints in a multi-stage pipeline setting, unless the end-to-end deadline is partitioned such that each stage works to satisfy a local deadline. Such partitioning must be done dynamically in a manner adaptive to current load, which makes it a non-trivial extension of the single-machine policy. The obvious extension of partitioning the end-to-end deadline *a priori* (e.g., by dividing by the number of stages) works very poorly because the stage load is not balanced, leading to poor performance and stability of such local schemes. Therefore, we deemed that comparisons with such algorithms would be unfair. Instead, we compare to two stable uncoordinated power management policies.

2.4.1 Workloads

To evaluate the expected real performance of our algorithm, we experiment with separate workloads for the two systems we implemented. The workload for the Synthetic system attempts to create a tunable server workload modeled after that of a typical three-tier web server. While it is less representative of a specific application, it is very flexible. On the other hand, while the workload of the TPC-W system does not represent many different types of applications, it is a very realistic model of an online bookstore application. Our goal in implementing two different systems is to conduct a more comprehensive evaluation, and to study the sensitivity of our algorithms with respect to the workload.

2.4.1.1 Synthetic Workload

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 1,000 records, and each record consists of 20 variable character fields. All records are initially filled with a key and 19 random fractional numbers. The physical size of the database (220 MB) prevents it from being entirely cached on our machine (the maximal observed cache memory size was 126 MB), making this stage I/O-intensive.

The second stage implements 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; or 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 spatial 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.

Figure 2.3 shows a histogram of the inherent end-to-end delay distribution of this workload in the Baseline case with no concurrent requests in the pipeline.

Test requests from the client are generated by the `httperf` [60] workload generator tool at various average rates. The request interarrival times are exponentially distributed. An individual TCP connection is created for each request.

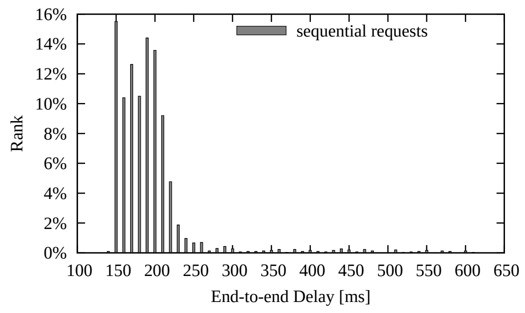


Figure 2.3: Synthetic workload end-to-end delay distribution at sequential load [35, Fig. 3].

2.4.1.2 TPC-W Workload

The database was populated as per the TPC-W requirements, with the scaling factors of 100 emulated browsers (EBs) and 1,000 items. The application server is logically further divided into two sub-tiers: it uses entity EJBs for database access (EJB tier) and servlets to provide access for clients to the specified web interactions (dynamic web tier). In order to achieve better scalability, we increased the size of the database connection pool from the default 20 to 50. We also increased the connection timeout value from 5 to 20 seconds, the largest deadline in the TPC-W benchmark. The HTTP front-end server is loaded with all the static data: 1,000 item images, 1,000 thumbnails, and miscellaneous small images such as buttons and icons. For dynamic requests (i.e. web interactions), the HTTP server is set up as a proxy to the application server. To generate realistic client requests and to collect our statistics, we used the Remote Browser Emulator running in real time (i.e. no slow-down factor was used). The client workload profile used for the evaluation is the TPC-W shopping mix (the basis for the primary TPC-W metrics), which consists of an average mix of browsing and ordering activity. We varied the offered load by adjusting the number of EBs, with other parameters (e.g., think time between user interactions) kept constant at their standard values.

2.4.2 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). The accuracy of the measurements was within 5%, similarly to the solution described in previous work [11]. Since the adapters provide constant voltage (18.5 V), 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 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 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 National 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, are collected from the output of the workload generator tool.

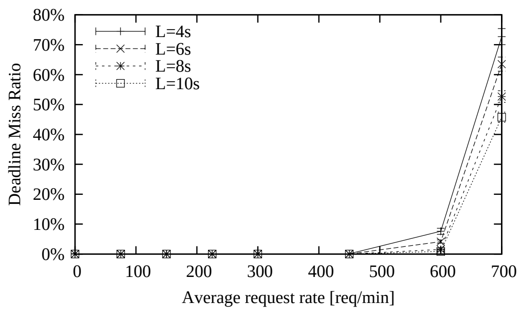
2.4.3 Performance Results

2.4.3.1 Synthetic Workload Results

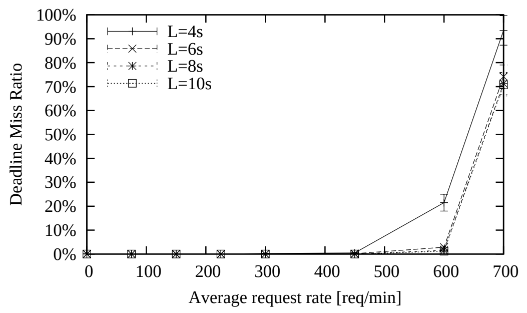
Next, we evaluate the energy savings and deadline miss ratio of the synthetic 3-tier service that runs our DVS algorithm. Each data point in our results is obtained by running several experiments for 3–5 minutes and plotting the average values along with error bars. Since we want to show the stable behavior of the system, we eliminate transient cold-start effects by running a short (18–30 s) lead-in workload prior to starting each experiment.

Figure 2.4(a) and (b) plot the deadline miss ratios of the two comparison policies as a function of the average request rate, which we vary from 0 (no load) to 700 requests/minute (severe overload).

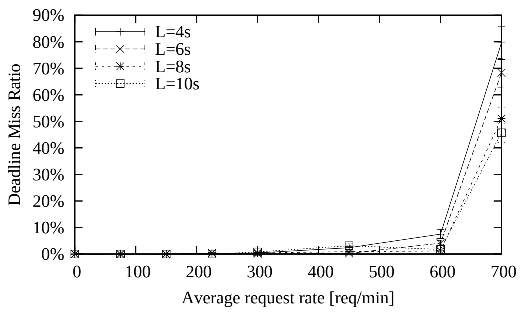
We perform several sets of experiments for different deadlines ranging from 4 to 10 seconds. These deadlines are natural for our setup for a number of reasons. First, delays in multi-tier web servers are the sum of the delays of individual stages. As the database tier typically has much larger delay due to I/O than other stages, a 4-second deadline easily translates to a sub-second delay bound to the first two stages in our three-tier prototype. Second, the typical e-business server workloads that we model usually include computationally complex operations that work on large data sets. These operations can cause delays to be on the order of seconds in these systems. Note that the TPC-W specified delay constraints are also in a similar range (3–20 seconds). Another factor is that our testbed computers are slow compared to real-life web server hardware. Obviously, on faster machines shorter deadlines are possible. Nevertheless, real studies with e-business web site users [9] show that these deadlines are in the tolerable range in most cases. The Baseline graph (Figure 2.4(a)) shows that the system begins to saturate at 450 requests/minute in each case, and that saturation is naturally slower with higher deadlines.



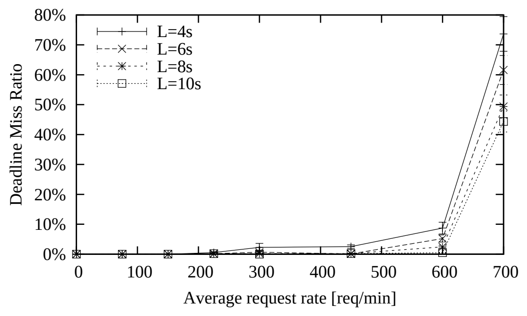
(a) Baseline.



(b) Independent DVS.



(c) Feedback DVS.



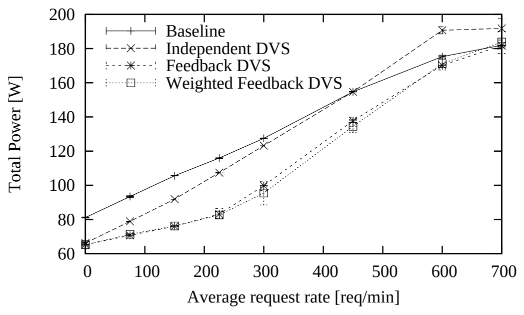
(d) Weighted Feedback DVS.

Figure 2.4: Performance of comparison algorithms and our novel algorithms (Synthetic workload) [35, Figs. 4–5].

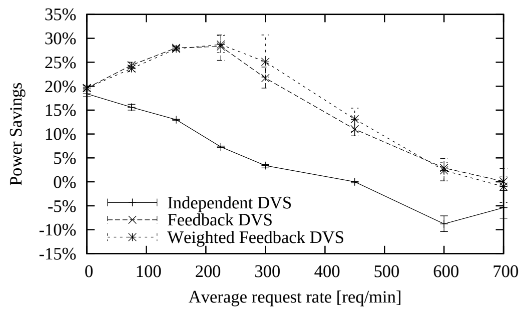
Figure 2.4(c) presents the miss ratio of the (simple) Feedback DVS algorithm. This first version is one where all weights W_i are assumed to be equal (as an approximation). The advantage of this version is that it does not require knowledge of the power characteristics of the CPUs, and characteristics of machine workload. If such information is available, however, it is possible to compute the coefficients W_i derived in section 2.2.4. Figure 2.4(d) shows the resulting improved (optimized) Weighted Feedback DVS algorithm using weights derived from empirical measurements. We can see that when the system is underloaded, our algorithms have a slightly higher deadline miss ratio than the comparison algorithms, but still within our specified tolerable limit, 5%. This means that the end-to-end delays are successfully controlled so that deadlines are statistically (at least 95% of the time) still met. Therefore, the increased miss ratios are acceptable in soft real-time systems such as our multi-tier web service. As we see next, the increased miss ratios are the results of improved power savings with our novel algorithms.

To illustrate what energy savings are achieved, in Figure 2.5(a), the total power consumption of the three servers using the feedback DVS policies is compared to the total power consumption using the two comparison policies. 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 note that at many data points, the power measurements were so consistent across our experiments that the corresponding error bars are not visible graphically.) We can verify that the Baseline power saving policy in fact saves a considerable amount of power in itself when the system is underutilized: it achieves an approximately 80 W base power consumption out of the highest observed power of over 180 W.

Finally, Figure 2.5(b) displays the overall power savings attained by the two feedback DVS policies and the independent DVS policy. 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. As conjectured, both of our algorithms also have a great advantage over the Independent policy. 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



(a) Total system power consumption.



(b) Total DVS power savings.

Figure 2.5: Power consumption using our novel algorithms (Synthetic workload) [35, Figs. 6–7].

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, therefore most of the power saving opportunities are exploited by the Baseline policy, and little can be added by the DVS algorithms. As load increases, there is less chance for the `HLT` instruction to be executed. 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 over-provisioned, a substantial power reduction is possible using our schemes.

2.4.3.2 TPC-W Workload Results

In this section, we describe the experimental results obtained from our 3-tier TPC-W service. Every data point reports the results of multiple repeated experiments, showing the average behaviors and the observed deviations. Each individual test run consists of a 10-minute ramp-up period, a 30-minute measurement interval, and finally a 5-minute ramp-down period. The load placed on the system is identical throughout each test run, but data collection takes place solely in the measurement interval. The ramp-up period is used to warm up the system to the desired operating point, while the purpose of the ramp-down period is to keep the system at that point even as the measurement finishes. Thus, test startup and shutdown effects are eliminated from the results. These test runs are much longer than the ones we performed with the Synthetic workload in order to meet the TPC-W requirements. The long measurement interval is necessary, for instance, to collect sufficient

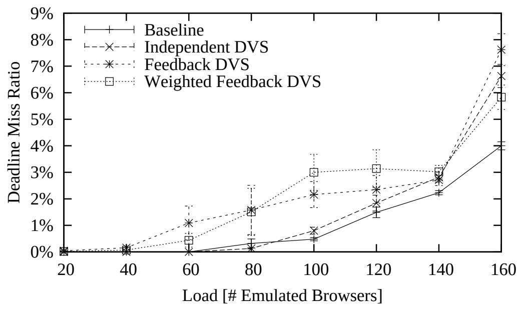
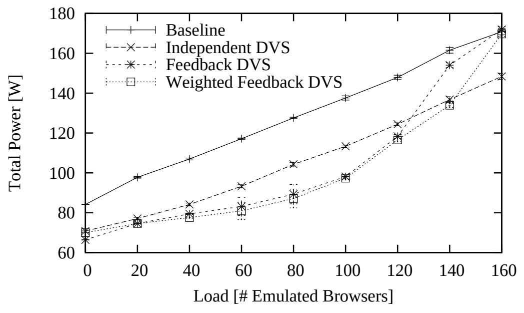
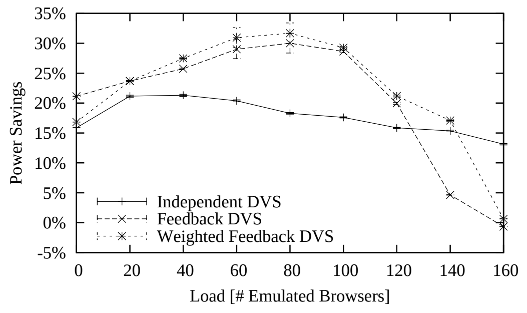


Figure 2.6: Comparative performance (TPC-W workload) [35, Fig. 8].



(a) Total system power consumption.



(b) Total DVS power savings.

Figure 2.7: Power consumption using our novel algorithms (TPC-W workload) [35, Figs. 9–10].

samples from each individual web interaction type.

Figure 2.6 displays the performance (deadline miss ratio) of each algorithm we considered with the TPC-W workload. Since every TPC-W web interaction type (class) has a specified end-to-end deadline, the plotted data points reflect the aggregate miss ratios, i.e. the number of interactions that missed their deadline per the total number of interactions. The graph shows that our performance goal (5% miss ratio) is met by each policy when the system is not overloaded.

Figure 2.7(a) and (b) show the results of the power measurements. The first plots the absolute total system power consumption, while the second visualizes the gains of each considered algorithm relative to the Baseline scheme. We can make similar observations to the ones about the Synthetic system. Both of our new algorithms exhibit improved power savings over both comparison policies. Again, our Weighted algorithm slightly outperforms our simple one as expected. Bigger differences can be expected between the two policies in heterogeneous systems, where the simple algorithm would become less useful. For this workload, the Independent DVS policy saves somewhat more power at the highest loads, but with a slightly higher deadline miss ratio than the Weighted DVS algorithm. In conclusion, we have obtained overall very similar results for the TPC-W workload as for the Synthetic one, which suggests that our schemes are not sensitive to a specific workload type.

2.4.3.3 Discussion

To understand why our algorithms perform better than the comparison policies, the key is to realize that the inputs available to each algorithm are different. Since the local (independent) algorithms have no knowledge of the global performance of the system (i.e., whether end-to-end deadlines are met), they can only assess how much CPU capacity the machine should provide based on local metrics, such as the CPU utilization. As periods of high CPU utilization can occur in some machines without causing global performance problems (deadline misses), in such cases the local algorithms waste power by unnecessarily increasing the overall capacity. By contrast, our global (coordinated) algorithms allow machines to run at lower capacity and hence lower power as long as the actual performance constraints are not violated. As we saw on the graphs, the difference, i.e. the wasted power to provide unnecessary capacity, varies with the offered load. Moving from low to medium load, it increases because the amount of periods with high CPU utilization but no performance constraint violations grows. At medium load the growth stops as violations start occurring, and moving towards high load the difference decreases because the increased capacity gradually becomes more necessary, thus less wasteful.

One might wonder if our algorithms have any advantage over a static scheme that configures the CPUs to run at some fixed clock speed. Since such a scheme is open-loop by nature, our algorithms possess all the advantages of closed-loop systems, most notably: decreased sensitivity to parameter variations in the controlled system, and improved rejection of transient disturbances. In other words, since the static scheme must be calibrated for a specific system and workload, it could not achieve optimal performance because of persistent errors in calibration and transient noise in the system. Further insight can be gained by measuring the average number of CPU speed adjustments during experiments with constant average load. The results shown in Table 2.1 indicate that significant variability exists in the system even if the load is kept constant. (Note that we verified that the adjustments were not controller-induced oscillations.) Therefore, an open-loop algorithm such as the aforementioned static scheme is unlikely to achieve similar performance.

Load (EB)	20	40	60	80	100	120	140	160
Adjustments	0.5	1.3	4.1	9.2	20.8	25.0	18.8	3.5

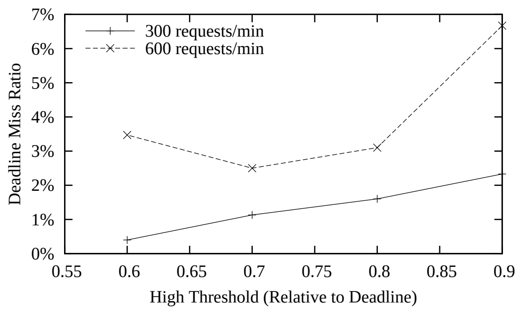
Table 2.1: CPU frequency adjustments per minute with the Weighted DVS algorithm [35, Table 1].

2.4.3.4 Parameter Sensitivity

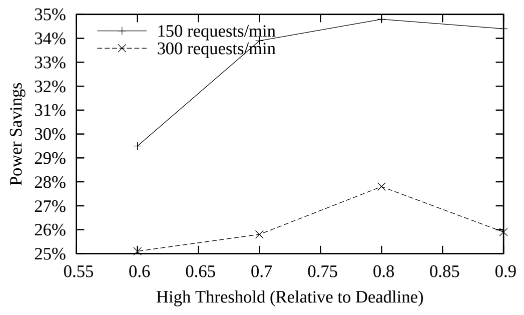
We performed additional experiments to evaluate the effects of varying the thresholds of our algorithm. As we shall see, decreasing these thresholds generally reduces both deadline misses and power savings. In Figure 2.8(a) and (b), we verify that 0.7 is a good choice for high threshold because it yields lower miss ratio than higher values, and it saves more power than lower ones. Next, in Figure 2.8(c) we can see that a low threshold of at most 0.6, which corresponds to a deadzone of at least 0.3 (since the high threshold is fixed at 0.9), yields low miss ratio. At the same time, Figure 2.8(d) shows good power savings when the low threshold is at least 0.4. Reconciling with our selected high threshold of 0.7 and our minimum deadzone requirement of 0.3, we verify that 0.4 is indeed a good choice for the low threshold.

2.4.3.5 Observations

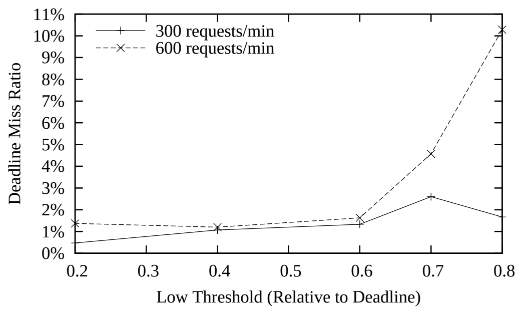
Let us note that for both systems (Synthetic and TPC-W), data points that are compared to calculate the power savings for a specific load level, are obtained from experiments of approximately the same duration and amount of work, with negligible differences. Thus, the measured average power consumption was proportional to the total energy spent during compared experiments. From this, it can be seen that the total energy savings are approximately equal to the total power savings presented. Our intuition is also supported by Figure 2.9, where we plot the energy savings in the TPC-W experiments. The basis of comparison is total system energy spent per web interaction, which is obtained by dividing the average total system power by the average web interactions per second (WIPS, a standard TPC-W metric).



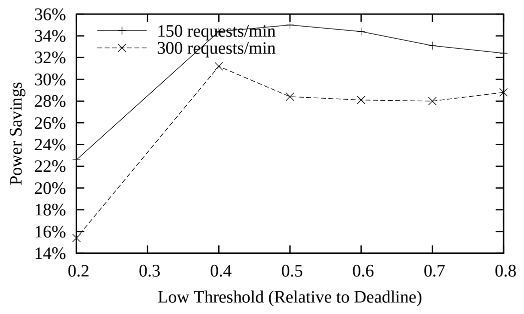
(a) Effects of high threshold on deadline miss ratio.



(b) Effects of high threshold on power savings.



(c) Effects of low threshold on deadline miss ratio.



(d) Effects of low threshold on power savings.

Figure 2.8: Effects of different thresholds on power and performance: (a)–(b) evaluates different high thresholds with fixed low threshold $= 0.5$; (c)–(d) evaluates different low thresholds with fixed high threshold $= 0.9$ [35, Figs. 11–12].

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, more optimal savings occur when the *weighted transformed* utilizations of all machines are equal than when utilizations are perfectly balanced. This interesting observation 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.

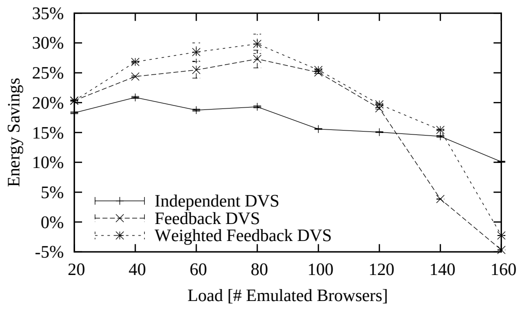


Figure 2.9: Total DVS energy savings (TPC-W workload) [35, Fig. 13].

2.5 Conclusions

In this chapter, we presented a distributed DVS control algorithm that minimizes overall 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 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. Empirical 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 costs of large server farms.

In the following chapters, this work will be extended to multiple classes of clients with different timing constraints, and to larger server clusters with multiple machines per stage.

Chapter 3

Exploiting QoS Classes

3.1 Introduction

In addition to coordinated DVS described in chapter 2, another potential source of energy savings in soft real-time systems exists through the introduction of Quality of Service (QoS) classes. When some part of the workload requires less stringent latency requirements (such as those originating from low-priority or non-interactive clients), it may become possible to further decrease the capacity of the system to conserve more energy. For example, in many practical multi-tier Internet services, classes of clients can be naturally assigned different priorities based on their performance requirements. In addition, there can be different types of requests with different acceptable response delays. Often, the acceptable processing time limit of any particular operation is proportional to its client-perceived complexity. Hence, power savings can be increased by allowing longer latencies for request classes whose client-perceived complexity is greater. The main question this chapter aims to answer is how much additional energy can be saved by such workload classification.

Fundamentally, there are two ways to accommodate different classes. One is to partition resources among classes. This, however, may result in decreased efficiency if extra resources in one partition cannot be borrowed by another. This problem can be addressed using virtualization. An alternative approach for service differentiation is to prioritize access to key resources. Here, the system must be modified to give lower priority to the service classes with more relaxed latency requirements. Then, if the server is prioritized and the DVS algorithm is made aware of the relaxed performance requirements of some classes of requests, it may be able to operate some of the server machines in slower P-states, resulting in overall power savings.

This chapter investigates the design issues and energy savings benefits of service prioritization in multi-tier web server clusters. We make two contributions. First, we demonstrate an inexpensive design to effectively prioritize the whole multi-tier system running commodity software requiring no application or OS modifications. Second, we quantify the improvement in the system's overall energy efficiency due to such prioritization through experiments on our testbed with a realistic multi-tier web server workload while keeping the existing cluster-wide energy management technique. Prioritization saves up to 15% additional energy through exploiting the different performance requirements of separate service classes, with only 3% increase in average deadline miss ratio (DMR) and an up to 4% *decrease* in DMR for high-priority requests. This work appeared in [36].

The rest of this chapter is organized as follows. Section 3.2 motivates and presents our design and discusses alternatives. Section 3.3 describes important implementation issues. In section 3.4, we introduce our testbed and our workload, then present our experimental results. Section 3.5 considers related work specific to QoS, and finally, section 3.6 concludes the topic.

3.2 System Architecture

3.2.1 Motivation

One of the successful existing power and energy management techniques involves power-aware QoS, a combination of dynamic power management and service prioritization [75]. The essence of such a technique is that lower-priority tasks (requests) with longer deadlines can be satisfied with lower system performance. Consequently, the power usage of the system can be reduced.

The power-aware QoS approach was shown to yield good results. However, prioritization in itself has not been evaluated from an energy management standpoint. This makes it difficult to predict how it would affect a system with an existing dynamic power management solution already in place. Therefore, our work is motivated by the question of what the added energy efficiency gain of prioritization is.

3.2.2 Assumptions

We address server farms running a multi-tier service such as an e-commerce website. The general architecture is shown in Figure 3.1. Based on the expected load, each tier is statically preallocated a number of machines (m_i), among which the total offered load arriving at the tier is evenly distributed.

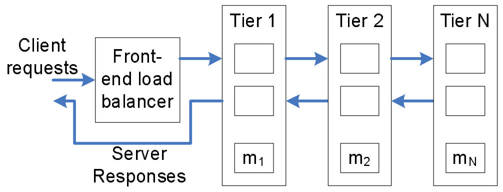


Figure 3.1: General model of clustered multi-tier system [36, Fig. 1].

We assume that, at a given throughput, the most important performance metric for the multi-tier service is end-to-end server latency, measured from the point a client request enters the first tier until a response is sent back to the client. Since response latency is the primary factor behind user satisfaction with web sites [9], maximizing server throughput alone is not sufficient. Hence, we assume that the site owner defines end-to-end deadlines that the server application has to meet. As it is typical in web services, these deadlines are considered soft, where a user-selectable maximum miss rate must statistically be met.

3.2.3 Design

3.2.3.1 Ideal Design

The complete prioritization of a traditional multi-tier server is very costly in terms of implementation effort. Consider the model of a typical server application's architecture shown in Figure 3.2. A request processor accepts requests from the input socket and places them in a task queue. The next available service thread starts executing the task. The task may create subrequests to the next tier,

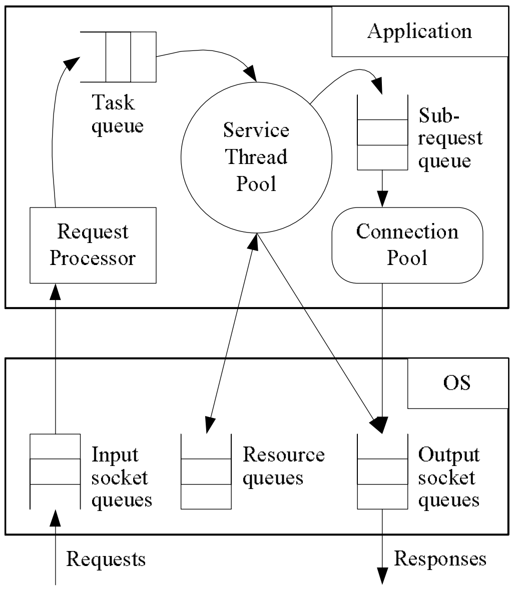


Figure 3.2: Simplified model of typical server in multi-tier system [36, Fig. 2].

which are queued up in the subrequest queue until a free connection is available. A thread may also become blocked and placed in an OS resource queue (e.g., CPU, disk, or semaphores). Finally, the response is buffered in an output socket queue until the OS is able to send it to the network.

Ideally, all of these queues should support priorities and all subcomponents of each task (such as subrequests, disk I/O, resource locks, etc.) should inherit its priority. However, widely used server OSs such as Linux still lack many of these features by default. Moreover, application support is typically largely missing. Hence, to implement the ideal design, one would have to modify critical parts of OS resource management and scheduling, each server application, as well as the communication protocols to propagate priority information between them.

3.2.3.2 Inexpensive Design

Our design goal is to find the least intrusive prioritization solution that is effective for the multi-tier system. We avoid application modifications since most server applications are large and complex, and the source code is often not available. It is nevertheless critical that the task queue and sub-request queue behave as if they were prioritized because in general both can become a bottleneck. The reason is that both of these queues are served by a usually small number of pooled resources (threads or connections) and the service times can be relatively large. The local service time of a task primarily depends on how resource-intensive the given tier is, while the latency of a subrequest is the total task service time of the next tier (including its subrequests).

As a simple solution, in many cases it is possible to run multiple instances (one for each service class) of the same server application and prioritize them on the process level. This in effect creates separate queues for each class, which are served by each instance in FIFO order. Assuming the OS supports preemptive real-time process scheduling (such as the `SCHED_FIFO` scheduling class in Linux), by assigning real-time priorities to the instances, we ensure that higher-priority queues are served first whenever there are idle threads or free connections available. Since now we have separate processes for each class, we also have separate communication channels for each class between the tiers. Thus, fortunately there is no need to modify the communication protocols to add priority information, since it is implicitly preserved by the connection structure: each instance only issues subrequests to next-tier instances of the same priority.

There are many situations where this design must be simplified even further. For instance, database servers are generally not possible to run as multiple instances operating on the same data set. In addition, their disk-intensive workload leaves process-level prioritization less effective, unless OS support for prioritized asynchronous I/O was also added. Following our design goal, we solve the issue by leaving the database server unprioritized, while trying to minimize task queuing in it. By selecting restrictive connection pool sizes in the previous tier, we ensure that it becomes a bottleneck instead of the unprioritized database server. This works because now the bottleneck stage (typically a CPU-bound application server) is prioritized. It also limits the number of concurrent database queries, reducing priority inversion due to transaction and I/O locking. If the original connection pool size was already restrictive, it does not need to be further reduced, only partitioned between instances, since this reserves sufficient connections for high-priority requests to prevent their starvation. Following this principle, in our testbed we assigned 8 connections to 3 instances each, in place of our original pool of 24 connections.

Naturally, the inexpensive design has certain limitations. First, if the server's bottleneck is an OS resource that is not prioritized (e.g., disk or network), then process priorities are not helpful. In most cases, however, the bottleneck can be shifted to the previous tier as discussed above. Second, if an application that does not support multiple instances on the same machine, is not running on the last tier, then this design has no way of propagating request priority to the subsequent tiers. Fortunately, in typical 3-tier setups this is not the case: web and application servers allow multiple instances. Finally, if different-priority instances in the same tier need to perform intensive communication (e.g., to maintain fast-changing shared state coherent), then under heavy overload as the low-priority instances are starved, they cannot respond to any messages. However, such shared state poses scalability limitations in itself, and thus avoiding overload is important even without prioritization.

The inexpensive design represents a great reduction in complexity compared to the ideal design and still results in effective prioritization in our experiments.

3.3 Implementation

3.3.1 Overview

We deployed a three-tier web-serving system on Linux, with Apache on the first tier, JBoss on the second, and MySQL on the third. As the front-end load balancer, we used the Linux Virtual Server solution. Static content (e.g., images) are served directly from the first tier. Forwarding dynamic requests to JBoss and balancing them across the second tier is done using the Apache module `mod_jk`. Finally, database requests are issued by JBoss through the Connector/J MySQL driver. To avoid the complexity of database clustering, the third tier consisted of only one server. Since in our setup database performance is not the bottleneck, improving database scalability via clustering would not significantly affect our findings.

3.3.2 Energy Management

As the existing energy management solution, we employ the following simple feedback control-based DVS policy, discussed previously in more detail in section 2.2.2 (on page 22). The system periodically determines its load using the following user-defined criteria: if less than 5% of response latencies exceed the 50% of the deadline, the system is considered underloaded, while if more than 5% exceed the 90% of the deadline, we consider it overloaded. The algorithm increases the speed (i.e., P-state) of the most utilized server if the system is overloaded, or decreases the speed of the least utilized server if it is underloaded. Since in our workload (described in section 3.4.1), the majority of deadlines were 3 or 5 seconds, a 5-second feedback period was chosen to allow control actions to have an effect by the next period. The latency statistics are smoothed by exponentially weighted moving averaging (EWMA) to reduce measurement noise.

Our implementation has two components: an Apache module to measure the end-to-end latencies, and a daemon on each server that measures its CPU utilization, runs the feedback controller, and sets its P-state.

3.3.3 Server Replication

Simply replicating a server such that multiple identical instances are executed on the same machine can result in OS resource conflicts. Since many of these are exclusive (bound sockets, output files, server state, etc.), separate resources must be configured for each instance.

Another issue is that server instances that are assigned real-time process priorities can starve regular (non-realtime) processes such as our user-space DVS daemon. Hence, it may not get a chance to increase the machine's speed when needed. Therefore, the daemon is assigned an even higher real-time priority than the server instances.

3.4 Experimental Evaluation

3.4.1 Workload

We used the TPC-W benchmark, a very realistic model of an online bookstore application. On the first tier there are 2 web servers serving image files, on the second tier we have 4 application servers maintaining session state and executing business logic, and on the third tier we have a database server handling all persistent data. Client machines are running the Remote Browser Emulator, which emulates specified numbers of web clients. The database was populated with the scaling factors of 100 emulated browsers (EBs) and 1,000 items. The workload profile was the TPC-W shopping mix (the basis for the primary TPC-W metrics).

Experiments were repeated 5 times, and consisted of a 5-minute ramp-up period (to warm up the system), a 15-minute measurement interval, and finally a 30-second ramp-down period (to maintain load as measurement finishes). The error bars in our graphs show the standard deviation.

3.4.2 Experimental Setup

Our testbed consists of 8 AMD Athlon 64 PCs with 512 MB RAM. The processor has two P-states with frequencies at 1.8 GHz and 1.0 GHz. The machines are connected with 100 Mbps Ethernet. For measuring power, we use a Watts Up Pro power analyzer (with 3% accuracy).

We performed our experiments in three different test configurations. The Baseline configuration has no prioritization or DVS. In the NP-DVS configuration, we add our DVS solution, and all clients are treated equal. In the P-DVS configuration, however, we differentiate three clients by relaxing some of their deadlines, assigning them to corresponding priority classes, and prioritizing the multi-tier server cluster as described in section 3.2.3.2. For example, if a request type has a 3-second standard deadline, the three clients are assigned deadlines of 3, 6, and 9 seconds and priority classes 1 (most important), 2, and 3, respectively.

3.4.3 Evaluation Metrics

Our primary metrics to evaluate performance are the system's throughput (web interactions per second, WIPS) and its deadline miss ratio (DMR).

Our main results evaluate the energy efficiency of the system, showing total system power, and total system energy per web interaction (Joules/WI). The first metric is important since many clusters are thermally constrained and lowering the average power typically lowers related costs. Further, since it is measured for equal intervals, it is directly proportional to total system energy, which translates to electricity costs. The second metric is important because it is a measure of energy efficiency (more precisely its reciprocal), which represents the amount of "useful" energy

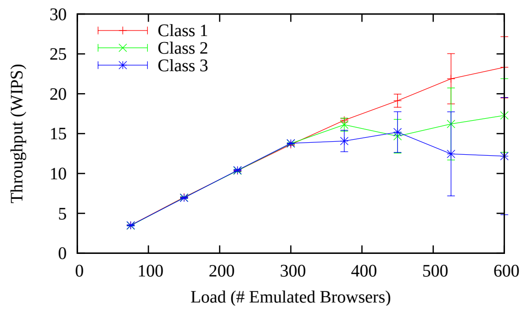


Figure 3.3: Throughput of individual classes with the P-DVS setup [36, Fig. 3].

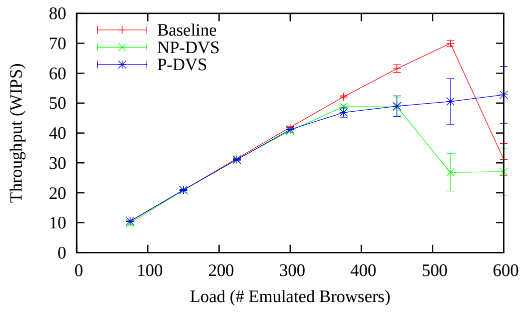


Figure 3.4: Total throughput comparison, summed across all classes [36, Fig. 4].

spent executing the successful web interactions. It also equals to average power divided by average throughput, which demonstrates that it also accounts for performance.

3.4.4 Performance Results

As expected of the prioritized system, Figure 3.3 shows that the throughput of higher-priority classes is better as the load increases toward peak capacity. When the load is low, the system has enough capacity to satisfy requests from all classes equally, hence the throughput is also identical.

Figure 3.4 compares the combined throughput of all classes achieved by each setup. Note that in the higher load region the Baseline setup sustains a higher throughput than both NP-DVS and P-DVS. This behavior is expected because, with many concurrent requests in the system, response times are stretched out more as the DVS policy slows down some servers, hence the web interactions become more spread out, resulting in lower throughput. In other words, since the think times in the emulated browsers do not depend on the response times, if the latter are longer, the next requests will arrive later, effectively reducing throughput. This is, however, not a problem as long as deadlines are honored; in fact, the goal of the DVS policy is to use up all slack time to save energy. It is a major advantage in that the DVS-capable server automatically paces client behavior without missing deadlines, as opposed to operating faster and bringing more requests upon itself that ultimately results in lower energy efficiency.

We make two key observations from Figure 3.4. First, the Baseline setup's performance quickly falls off under very high loads due to inefficient scheduling. In contrast, P-DVS maintains a more consistent throughput by still allowing most high-priority requests to complete. Second, P-DVS clearly wins over NP-DVS, which degrades even earlier than the Baseline because the stretched execution times increase concurrency, exacerbating the scheduling inefficiencies. The fact that performing DVS in a throughput-oriented soft real-time system places more stress on the scheduler points to a fundamental performance advantage of prioritization in conjunction with energy management.

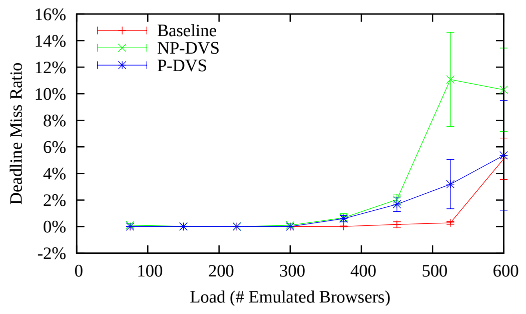


Figure 3.5: Overall deadline miss ratio comparison. Includes all classes. [36, Fig. 5]

Comparing the overall deadline miss ratios between setups in Figure 3.5, we can see that when heavily loaded, NP-DVS has a higher miss ratio since treating all clients equally causes it to saturate earlier. P-DVS, on the other hand, successfully maintains an acceptable miss ratio by reducing scheduler contention via prioritization and by relaxing deadlines of low-priority requests.

3.4.5 Energy Efficiency Results

The average power of each experiment, measured on the whole cluster, is shown in Figure 3.6. As expected, NP-DVS follows the curve seen with conventional DVS algorithms, converging with the Baseline as load increases. In contrast, P-DVS keeps power usage lower even as the system becomes

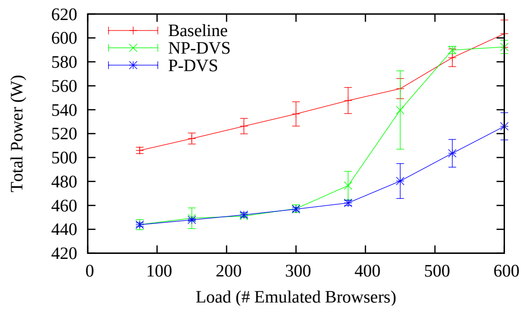


Figure 3.6: Total system power comparison [36, Fig. 6].

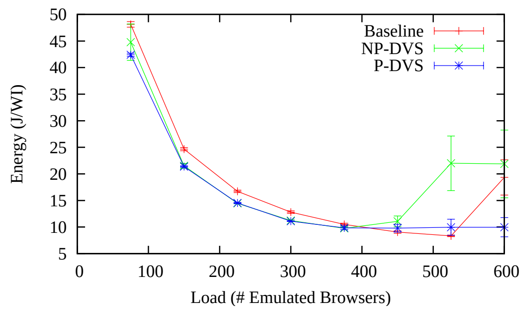


Figure 3.7: Comparison of total system energy per web interaction [36, Fig. 7].

overloaded. As we saw earlier, the differentiated client classes result in a better overall deadline miss ratio, which allows as much as 15% greater power savings. Note that since all experiments had identical measurement intervals, this also equals to the total energy savings.

Our final goal is to quantify and compare the energy efficiency benefits of the DVS setups. As discussed in section 3.4.3, energy efficiency is important since it factors in throughput in addition to energy usage. Figure 3.7 plots the average energy spent per web interaction, or the reciprocal of energy efficiency for each setup. While the graph shows no significant difference between the DVS setups at lighter loads up to 375 EBs (7–14% savings over the Baseline for NP-DVS vs. 6–13% for P-DVS), around 525 EBs we see a large increase with NP-DVS. This means a large drop in energy efficiency, which is due to the earlier saturation we observed in section 3.4.4 that causes throughput reduction. P-DVS, on the other hand, avoids wasting energy on less important requests and even achieves 48% savings over the Baseline at 600 EBs because of its ability to use less power and still have higher throughput.

3.5 Related Work in QoS

Several papers address priorities in individual servers [3, 18, 54, 75]. These solutions, however, require modifying the server application, which is costly and often not feasible. Other efforts [1, 77] have been directed at middleware QoS solutions that do not require application or OS changes. These do not, however, address multi-tier servers or energy consumption. In contrast, our work is concerned with the interaction of service differentiation with power management and the energy consumption of clusters. Closely related to our research is the work by Sharma et al. [75]. While we build on some of their results, we have a multi-tier system model that requires coordinated energy management.

There is significant recent research on energy management in server clusters [10, 20, 25, 31, 53, 68, 72]. However, they either do not deal with service differentiation or they are restricted to a single-tier cluster. An economically-driven energy and resource management framework was presented for clusters by Chase et al. [14]. This research is closely related to our work since it allows service differentiation in conjunction with energy management. However, extending it to the case of multi-tier servers with *end-to-end* latency constraints is not straightforward. Our work is distinguished from the above literature in that we address *multi-tier* clusters with different service classes, and we focus on how prioritization affects the energy efficiency of such clusters.

3.6 Conclusion

This chapter investigated how much additional benefit prioritization has on a multi-tier server system's energy efficiency with an existing DVS-based power management policy. We prioritized the multi-tier system without application or kernel modifications, and performed experiments using three client classes, two with relaxed deadlines. Our results clearly illustrate that the main benefit of prioritization is not only more stable performance as the system nears overload, but also greatly improved energy efficiency, resulting in energy savings of up to 15%.

Chapter 4

Reconfigurable Clustered Tiers

4.1 Introduction

The analyses in the previous chapters addressed multi-tier servers where each tier is, or is treated as, a single computer. However, my research seeks solutions to the energy management problems of large-scale server farms. To this end, the analysis is extended to the case where every tier of the system consists of a multiple-machine cluster. Observe that if the number of machines in each tier does not change, the extension is simple. To a first level of approximation, each tier can be viewed as a single super-processor of the combined capacity of all constituent machines and with a larger number of possible DVS performance states (P-states). Nevertheless, we investigate the more interesting and more general problem posed by variable-sized tiers, i.e. where the number of machines in each tier can change. This can be implemented such that each tier is a reconfigurable cluster, which has been previously investigated in single-tier servers [14, 67, 69]. In particular, we investigate the important problem of optimal assignment of machines to tiers: given a fixed total number of machines in the farm, the question is how to partition and assign those machines to tiers in a way that maximizes energy-efficiency. We analyze the problem, and propose an energy management policy for achieving theoretically optimal dynamic tier assignment.

Obviously, certain physical assignment constraints must be observed. For example, some machines may need to be outside a firewall while others might need to be inside. Such constraints are easy to incorporate by reducing the space of feasible assignments. Among those assignments that are feasible, certain ones are more energy-efficient than others. This is because the assignment essentially determines the overall load of tiers, affecting in turn the power-saving opportunities. For instance, in a traditional three-tier Web server farm, assigning an extra machine to the application server tier (which has CPU-intensive workload) instead of the database server tier may be energy-efficient if it significantly reduces the CPU utilization of that tier and therefore allows previously saturated servers to lower their frequencies. On the other hand, the same assignment may not be energy-efficient if the database tier experiences the greatest fraction of the end-to-end delay, and the addition of an extra machine would substantially reduce that fraction.

In the previous chapters, we only considered the DVS power management technique. Yet, many applications are able to deal with cluster reconfiguration, which has the following implications. With reconfigurable clustered tiers allowed, it also becomes feasible to automatically turn machines on or off based on load conditions, as long as at least one machine remains turned on in each tier. Moreover, due to the high idle power exhibited by modern hardware, it is *preferable* to turn off (i.e. transition to the `Soft-Off` sleep state) those machines that are not needed, as opposed to having them enter their lowest P-state. Intuitively, as the leakage power of CPUs is approaching 50% or more of their total power, leaving CPUs active with low utilization becomes extremely costly. At the same time, as entering the `Soft-Off` state involves shutting down all but a few circuits and subsystems, it has an even greater power saving potential than DVS, which should be leveraged. Therefore, our energy optimization policy is designed to account for this and show a bias toward putting machines to sleep whenever possible.

The rest of this chapter is structured as follows: section 4.2 discusses our system model along with power and performance characterization, section 4.3 presents an analysis of our power optimization problem, section 4.4 details our algorithms and design issues pertaining to them, section 4.5 contains implementation information, section 4.6 presents our evaluation methodology and experimental results, and finally we discuss related work and conclude.

4.2 System Modeling

In order to analyze the machine assignment problem, we must first create a formal system model. We begin by stating our assumptions about the system and introducing some notation. Then we proceed to develop models for estimating server tier latencies and power consumption built on the introduced system variables. Importantly, our power model does not simply follow from the commonly assumed CMOS processor power model [61] but from our system-level empirical observations and theoretical analysis.

4.2.1 Definitions and Assumptions

We consider multi-tier server clusters, consisting of a constant number of tiers, each of which are composed of a variable number of machines. All machines in one tier run the same application, and requests go through all tiers. The end-to-end server performance must meet a predefined service level agreement (SLA). We assume a simple SLA specifying a target average end-to-end server delay. Finally, each machine optionally supports multiple P-states (DVS).

In order to simplify the problem, we make a few important assumptions. First, we assume that machines within a single tier have identical power and performance characteristics; in other words tiers are homogeneous in terms of hardware. This is a reasonable assumption because, even though whole data centers are typically heterogeneous, it is normally preferred that groups of servers running the same application and being load balanced (i.e. the equivalent of a tier) are identical—this simplifies load balancing, among others. The second, related assumption is that there is perfect load balancing within a tier such that the CPU utilization of the constituent machines is equal. While this is idealistic, it is only required for analytical tractability, and based on our empirical observations the impact of moderate imbalances on actual performance is insignificant.

One concern when designing algorithms for large-scale clusters is invariably that of scalability. The assumptions above are not only helpful in reducing the complexity of the problem, but also in ensuring scalability because they allow the optimizations to be performed on groups as opposed to individual machines—aggregate measures of a whole tier will represent individual machines well.

Variable	Description
s	number of tiers (pipeline stages)
λ_i	total average arrival rate at tier i
m_i	number of machines in tier i
f_i	CPU frequency of each machine in tier i
U_i	CPU utilization of each machine in tier i
P_i	power consumption of each machine in tier i
P_{tot}	total power consumption
W_i	average service latency of tier i
W_{tot}	end-to-end (total) average service latency
D	target average end-to-end latency (soft deadline)

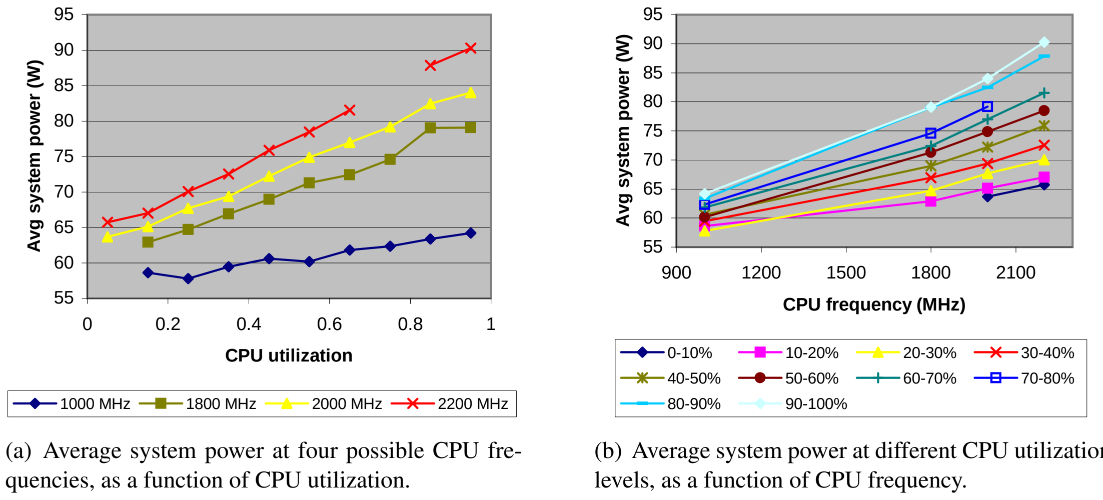
Table 4.1: Model variables of reconfigurable multi-tier cluster.

Also note that typical large-scale cluster architectures are composed of hierarchical units, where communication within the same unit (e.g., a rack of servers, or servers within the same enclosure) is less expensive than across units. This means that data can be collected (and commands distributed) efficiently through highly scalable hierarchical aggregation (and dissemination).

Table 4.1 summarizes the variables of our system model and their notation. As an implication of the assumptions above, we do not need to model individual machine CPU frequencies—it is sufficient to use a single value per tier. This is because, since power is a convex function of frequency [37], setting the same frequency on all nodes within a tier is equally or more power-efficient than setting different values. Further, the performance effects of frequency scaling are not greater than its effects on power [46], therefore there is no performance or energy-efficiency benefit from different settings within a (homogeneous) tier either. Therefore, we limit our study to the model where each tier is operated at a single CPU frequency.

4.2.2 Power Model

In our system model, two variables have a significant effect on the power draw of an individual active machine: its CPU utilization and frequency (which also affects core voltage). Other variables may have an indirect effect by influencing these. To obtain a predictive model from these two variables for machine power consumption, P_i, we analyzed the actual power measurements from a large pool of characterization experiments, in which a machine operates at varying U_i and f_i (varied together with voltage using DVS). Our experimental setup is described in section 4.6. The characterization data points were obtained by varying the incoming load (number of emulated clients) in each possible DVS state, and recording the CPU utilization, frequency, and the measured power. Figure 4.1 shows the resulting curves.



(a) Average system power at four possible CPU frequencies, as a function of CPU utilization.

(b) Average system power at different CPU utilization levels, as a function of CPU frequency.

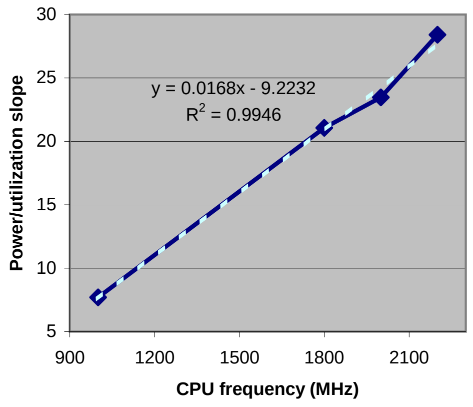
Figure 4.1: Average system power measurements with varying CPU utilization and frequency.

We found that P_i is approximately linear in both U_i and f_i for any fixed f_i and U_i, respectively. The near-linear relationship seen between power and utilization was expected, and was documented in previous work [22]. The power-frequency relation in Figure 4.1(b) is in reality slightly super-linear (as expected from the CMOS power model), however its linear approximation is still sufficiently accurate. This simply results from the supported core frequency-voltage pairs in our processors (listed in Table 4.2). Calculating fV^2 from the table (as a simple approximation for CMOS power) yields a qualitatively similar graph to our measured power. An at least quadratic function of f (suggested throughout the literature [e.g., 11, 19, 57] from the assumed CMOS model) can fit this almost perfectly. However, a linear function of f also provides a good fit with $R^2 = 0.9929$, and therefore it is preferable for our purposes because of its simplicity.

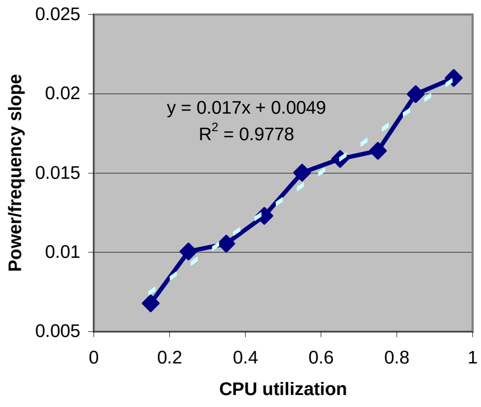
In addition to the nice linearity we see with each variable, even the slopes of those lines have good properties that can be easily established. As deducible from Figure 4.2, the slope over one variable is well approximated by a linear function over the other variable. That is, the slope $\partial P_i/\partial U_i$

Frequency (MHz)	Voltage (V)
1000	1.10
1800	1.30
2000	1.35
2200	1.40

Table 4.2: Performance states (core frequency-voltage pairs) available in our processors.



(a) Slope of system power over CPU utilization, as a near-linear function of CPU frequency.



(b) Slope of system power over CPU frequency, as a near-linear function of CPU utilization.

Figure 4.2: Characterization of the slopes of average system power over both independent variables. Data points show slope values calculated from the measured data. The dotted line shows a linear regression fit, labeled with its equation and R^2.

is approximately linear in f_i (Figure 4.2(a)) and $\partial P_i/\partial f_i$ is approximately linear in U_i (Figure 4.2(b)). With this information, we can analytically derive the general form of the two-variable predictive power model. First, let

$$\frac{\partial P_i(f_i,U_i)}{\partial U_i} = b_{i1} f_i + b_{i0}, \tag{4.1}$$

$$\frac{\partial P_i(f_i,U_i)}{\partial f_i} = c_{i1} U_i + c_{i0}. \tag{4.2}$$

Next, the differential dP_i can be written in the form

$$dP_i = \frac{\partial P_i}{\partial U_i} dU_i + \frac{\partial P_i}{\partial f_i} df_i. \tag{4.3}$$

Since $P_i(f_i, U_i)$ is "well-behaved" (single valued and both itself and its derivatives are continuous), we can proceed as

$$\frac{\partial^2 P_i}{\partial U_i \partial f_i} = \frac{\partial^2 P_i}{\partial f_i \partial U_i}$$

$$b_{i1} = c_{i1}. \tag{4.4}$$

Note that this is confirmed by our empirical data in Figure 4.2, where $b_{i1} = 0.0168$ and $c_{i1} = 0.017$. From this, it follows that Equation (4.3) is an exact differential equation. The solution can be obtained as

$$P_i = b_{i1} f_i U_i + c_{i0} f_i + b_{i0} U_i + constant, \tag{4.5}$$

which after renaming yields the final model:

$$P_i(f_i, U_i) = a_{i3} f_i U_i + a_{i2} f_i + a_{i1} U_i + a_{i0}. \tag{4.6}$$

In order to instantiate the model for a specific machine, its parameters need to be determined. We did so by simply running an optimization algorithm to find parameter values that minimized the sum of squared errors over all data points between the model prediction and our actual measured data. This process resulted in optimal values lying close to the theoretically obtained solution in Equation (4.5), further validating our methodology. Table 4.3 contains the values for reference. Power estimation for our test system using this model was fairly accurate, showing a good fit ($R^2 = 0.9879$) with an average error of 1% and a worst case error less than 4%. A more general, third-order polynomial model was also attempted for comparison, but it did not yield actual improvement (it merely resulted in over-estimation, i.e. the estimation following measurement disturbances more closely).

Parameter	Theoretical value	Optimal value
a_{i3}	0.0168	0.0155
a_{i2}	0.0049	0.0059
a_{i1}	-9.223	-7.212
a_{i0}	—	49.967

Table 4.3: Power model parameter values (theoretical and optimal).

4.2.3 Delay Model

4.2.3.1 CPU Utilization

The service latency of short requests in concurrent servers is mostly a function of CPU utilization. Hence, in order to predict latency for various cluster configurations and performance states, we first need to model CPU utilization in those configurations. This is achieved through a two-step process: we first estimate the current offered load from measurements, and then based on this we can predict utilization for the selected configurations.

Given our assumptions on perfect load balancing and equal CPU frequencies in a tier, the relation between offered load λ_i, CPU utilization U_i, tier size m_i and frequency f_i can be estimated using

$$\lambda_i = m_i f_i U_i. \tag{4.7}$$

Once this is done, the utilization of any tier configuration can be predicted using the same relation:

$$U_i = \frac{\lambda_i}{m_i f_i}. \tag{4.8}$$

In Figure 4.3, we experimentally evaluate the accuracy of this approach. We ran individual experiments in all possible tier configurations, at various constant offered loads. Ideally, for a given load we want to see the same estimation in all configurations (i.e., the same line repeated on the graph), since the true offered load (number of clients) stays the same regardless of server configuration. The graph shows that while varying tier size does not, varying CPU frequency does cause inaccuracy in the estimates. The estimation is imperfect due to the fact that CPU frequency scaling does not exactly linearly affect server processing capacity. However, this inaccuracy is acceptable,

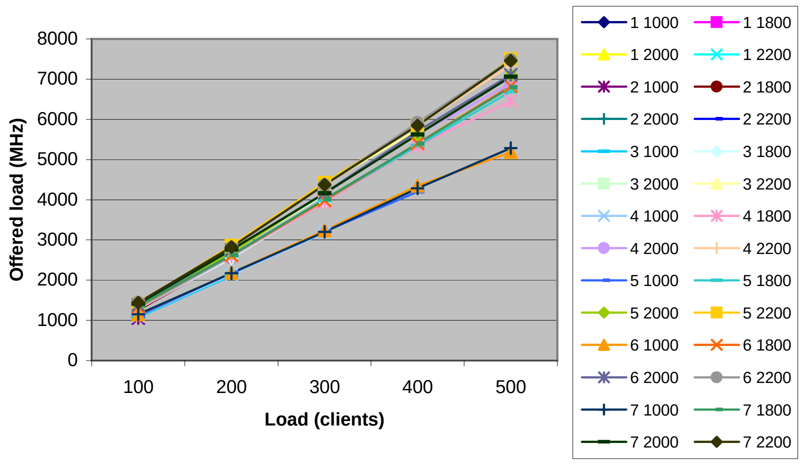


Figure 4.3: Estimation of the offered load at different operating points. Each line represents a different tier configuration: the legend shows the tier size and the CPU frequency in MHz. The true offered load (number of clients) was varied in each configuration, and the estimated load was calculated from the measured CPU utilization.

because it simply results in the slight overestimation of the performance effect of a frequency adjustment. If the estimate is used to predict utilization at some smaller or greater frequency than the current one, the prediction will be too high or low, respectively. This means that an algorithm based on this model will likely adjust the frequencies slightly more conservatively (with smaller modifications) than using a perfect model. Further note that model errors will be compensated for using feedback control.

Finally, the prediction model was validated: load estimates from one configuration ($m_i = 7$, $f_i = 2200$) were used to predict the CPU utilization for all other configurations. This produced fairly accurate values with a good correlation to the actual measurements ($R^2 = 0.9690$).

4.2.3.2 Service Latency

Using the model discussed above to predict CPU utilization, service latency can also be predicted for any tier configuration. To obtain our latency model, we performed nonlinear regression analysis using a heuristically decided format. After computing the fitting coefficients (via curve fitting to our measurements) we obtained the following model ($R^2 = 0.9968$).

$$W_i(U_i) = \frac{w_{i1}}{(1-U_i)^2} + w_{i0} \tag{4.9}$$

Note, however, that in reality, requests can take longer and so latency also slightly depends on CPU frequency. According to measurements in our test system, this had a very limited impact and could be safely ignored.

4.3 Theoretical Analysis

4.3.1 Optimization Problem Formulation

For our purposes, a power management strategy is optimal if it assigns machines to tiers and determines their operating frequencies such that the total power consumption of the system is minimal while deadlines are still met. This is more formally expressed as the following minimization problem:

$$\begin{aligned} \min_{m_i, f_i} \quad P_{\text{tot}} &= \sum_{i=1}^{s} m_i P_i \left(f_i, \frac{\lambda_i}{m_i f_i} \right) \\ &= \sum_{i=1}^{s} \left(a_{i3}\lambda_i + a_{i2} m_i f_i + a_{i1} \frac{\lambda_i}{f_i} + a_{i0} m_i \right) && (4.10a) \\ \text{subj. to} \quad W_{\text{tot}} &= \sum_{i=1}^{s} W_i \left(\frac{\lambda_i}{m_i f_i} \right) \le D && (4.10b) \\ \text{and} \quad \sum_{i=1}^{s} m_i &\le M. && (4.10c) \end{aligned}$$

4.3.2 Tier Balancing Condition

We solve the problem using the method of Lagrange multipliers. The Lagrangian function is:

$$\begin{aligned} L(m_i, f_i, l_1, l_2) = &\sum_{i=1}^{s} \left(a_{i3}\lambda_i + a_{i2} m_i f_i + a_{i1} \frac{\lambda_i}{f_i} + a_{i0} m_i \right) \\ &+ l_1 \left(\sum_{i=1}^{s} W_i \left(\frac{\lambda_i}{m_i f_i} \right) - D \right) + l_2 \left(\sum_{i=1}^{s} m_i - M \right). \end{aligned} \tag{4.11}$$

Setting $\forall i : \nabla_{m_i,f_i} L = 0$, we get for each i:

$$\frac{\partial L}{\partial m_i} = a_{i2} f_i + a_{i0} + l_1 \frac{\partial W_i}{\partial m_i} + l_2 = 0, \tag{4.12a}$$

$$\frac{\partial L}{\partial f_i} = a_{i2} m_i - a_{i1} \frac{\lambda_i}{f_i^2} + l_1 \frac{\partial W_i}{\partial f_i} = 0. \tag{4.12b}$$

Equations (4.12) are independent of the latency function. To arrive at a specific solution, the actual latency function has to be substituted. For instance, using Equation (4.9), which our characterization has led to, the general latency constraint (4.10b) becomes:

$$W_{\text{tot}} = \sum_{i=1}^{s} \left(w_{i1} \left(1 - \frac{\lambda_i}{m_i f_i} \right)^{-2} + w_{i0} \right) \leq D, \tag{4.13}$$

and Equations (4.12) expand to:

$$\frac{\partial L}{\partial m_i} = a_{i2} f_i + a_{i0} - 2 l_1 \frac{w_{i1} \lambda_i}{\left(1 - \frac{\lambda_i}{m_i f_i} \right)^3 f_i m_i^2} + l_2 = 0, \tag{4.14a}$$

$$\frac{\partial L}{\partial f_i} = a_{i2} m_i - a_{i1} \frac{\lambda_i}{f_i^2} - 2 l_1 \frac{w_{i1} \lambda_i}{\left(1 - \frac{\lambda_i}{m_i f_i} \right)^3 m_i f_i^2} = 0. \tag{4.14b}$$

It is easily shown that the Equations (4.14) are not independent, therefore a concrete solution in general cannot be obtained. However, solving (4.14) for l_1 results in our optimality criterion:

$$G(m_i, f_i) = (1 - U_i)^3 \frac{m_i}{w_{i1}} \left(\frac{a_{i2} f_i}{U_i} - a_{i1} \right) = 2 l_1. \tag{4.15}$$

This means that, assuming the constraints are active (i.e., the total latency can exceed the deadline), total cluster power use can only be optimal if G are equal across all tiers, more formally if

$$\forall i, j : G(m_i, f_i) = G(m_j, f_j). \tag{4.16}$$

4.4 Policy Design

We designed an energy management policy that takes advantage of our predictive models and theoretical results. The policy is periodically invoked, when it obtains basic measurements from the system, applies the models, computes the optimal power states, and initiates the necessary power state transitions.

4.4.1 Energy Management Policy

A simple approach to computing the optimal power states would be to perform an exhaustive search over the possible cluster configurations, selecting the one that best matches the optimality criterion derived above. However, there are two issues with this. Firstly, the criterion is only a *necessary* condition of optimality, and may not be a sufficient condition. Secondly, an exhaustive search does not scale up to large cluster sizes, or to large numbers of possible frequencies. Therefore, to address these issues, we designed a greedy heuristic search algorithm to perform the optimization. The algorithm relies on an additional assumption regarding the power characteristics of the hardware: it assumes that the static system power dissipation is large enough that turning on an additional machine and lowering the common CPU frequency such that CPU utilization is kept constant, never saves power. More formally,

$$(m_i+1)P_i(\frac{\lambda_i}{(m_i+1)U_i},U_i) \geq m_iP_i(\frac{\lambda_i}{m_iU_i},U_i), \tag{4.17}$$

which reduces to

$$a_{i1}U_i + a_{i0} \geq 0. \tag{4.18}$$

This is a very realistic assumption with current server hardware. For instance, the machines in our testbed clearly satisfy it according to the parameter values in Table 4.3, and prior work has relied on similar findings [68].

As per the aforementioned assumption, the optimization can be performed in two rounds. The first round finds the minimum number of machines in each tier so that the end-to-end performance constraint $W_{\text{tot}} \leq D$ is satisfied using the highest frequency for the machines. We start from computing a minimum allocation that only ensures that any one tier will not exceed the end-to-end deadline in itself. Then, we continue the search by incrementally assigning machines to tiers one by one, until the predicted total latency meets the constraint. At each step, we need to decide which tier to add a machine to. We base this decision on our optimality criterion: we choose the tier that, with the added machine, would result in the "most equal" values $G(m_i, f_i)$. More precisely, we choose the assignment for which the standard deviation of the values G is minimal. In the second round, tier frequencies are decreased one at a time in a similar fashion, as long as the response time constraint is still met. The decision which tier to affect at each step, is guided by the same objective: select the change that minimizes the standard deviation of $G(m_i, f_i)$. A detailed pseudo-code is listed in Algorithm 4.1. (The two-round structure of our heuristic is similar to the queuing theory based heuristic proposed by Chen et al. [15].)

As with any greedy algorithm, the risk of finding local minima exists. However, its impact is limited to reducing energy savings and not performance, since it only affects *how* the performance goal is met. Furthermore, the impact is small in larger clusters because adjustments to G become relatively smaller, making it easier to equalize.

4.4.2 Allocation Constraints

A few additional constraints must be given attention. For example, if the minimum feasible m_i, calculated in line 4, was greater than M (e.g., due to a severe system overload), the second round might erroneously reduce some frequencies. Therefore, it must be ensured that initial values m_i are feasible (i.e., by cropping tier sizes until their sum equals M).

Another important practical consideration is that some tiers may not actually be reconfigurable. A good example is a database server tier, which does not support hot addition or removal of its cluster nodes. This constrains the feasible allocations, which must be reflected by reducing the search space. The solution might seem trivial: for non-reconfigurable (static) tiers, instead of calculating a minimum feasible m_i, simply assign the static tier size to m_i, and then do not attempt to add more machines to them (modify the `for` loop at line 7 to only loop over reconfigurable tiers). However,

Algorithm 4.1 MultitierClusterEnergyOpt pseudo-code.

```
Ensure: $W_{\text{tot}} \leq D$ and $P_{\text{tot}}$ is near minimal
 1: for $i = 0$ to $s$ do
 2:     $f_i \Leftarrow$ Maximum supported frequency for tier $i$
 3:     $\hat{U}_i \Leftarrow 1 - \sqrt{w_{i1} / \left(D - \sum_j^s w_{j0}\right)}$ {Maximum feasible $U_i$ s.t. $W_i = D$}
 4:     $m_i \Leftarrow \lceil \lambda_i / (\hat{U}_i f_i) \rceil$ {Minimum feasible $m_i$ s.t. $U_i \leq \hat{U}_i$}
 5: end for
 6: while $W_{\text{tot}}((m)_i, (f)_i) > D$ and $\sum_i^s m_i < M$ do
 7:     for $i = 0$ to $s$ do
 8:         $\sigma'_i \Leftarrow \text{StdDev}((G(m_j, f_j) | 0 \leq j < s, j \neq i), G(m_i + 1, f_i))$
 9:     end for
10:     $i \Leftarrow \text{argmin}_i(\sigma'_i)$
11:     $m_i \Leftarrow m_i + 1$
12: end while
13: while $W_{\text{tot}}((m)_i, (f)_i) < D$ and not all $f_i$ are minimal do
14:     for $i = 0$ to $s$ do
15:         $f'_i \Leftarrow$ Next frequency lower than $f_i$
16:         $\sigma'_i \Leftarrow \text{StdDev}((G(m_j, f_j) | 0 \leq j < s, j \neq i), G(m_i, f'_i))$
17:     end for
18:     $S_i \Leftarrow \sigma'_i \quad$ if $W_{\text{tot}}((m)_i, (f)'_i) \leq D)$
19:     if $S = \emptyset$ then
20:         break
21:     end if
22:     $i \Leftarrow \text{argmin}_i S_i$
23:     $f_i \Leftarrow f'_i$
24: end while
25: return $((m)_i, (f)_i)$
```

an important detail must not be overlooked: static tiers may be much less energy efficient (due to overprovisioning), and accordingly may have a hugely different G than the rest of the tiers that can only be slightly affected by adjusting f_i. Including that value during optimization would cause some other (reconfigurable) tier to be assigned many unneeded machines in an attempt to bring its G closer to the inefficient tier, reducing the overall $\text{StdDev}(G_i)$. This would result in seriously *degraded* system energy efficiency. Therefore, in the first round of the algorithm, static tiers must be completely excluded from the set of tiers used in the standard deviation calculation (line 8).

4.4.3 Load Estimation with Feedback Control

As discussed in section 4.2.3.1, estimation of the offered load is crucial for predicting the latency in a target cluster configuration. While we demonstrated that this is easy to calculate from the current CPU utilization with relatively good accuracy, note that this approach has a limitation: demand that is higher than total CPU capacity cannot be detected from utilization measurement because it saturates at 100% (and in practice even lower). Therefore, relying on this method only would result in unacceptable underestimation of offered load as the system got into saturation. Hence, other inputs are needed to detect this condition. We chose to add two types of performance monitoring: one to detect when response times exceed D, and another to detect service errors resulting from overload (e.g., timeouts), both meaning our estimate of λ_i is low. Once detected, we rely on feedback control to increase our load estimates, in order to quickly and reliably drive performance within the SLA specifications. It is important to note that the addition of our feedback controller is beneficial in several ways:

- it solves the aforementioned limitation of CPU utilization-based load estimation;
- it allows performance specifications to be met despite errors in modeling, characterization inaccuracies, parameter variations, and practical issues such as server errors due to overload;
- it results in improved fault tolerance properties of the energy management policy, and in reduced sensitivity to aberrant behavior (this point will be explored in chapter 6).

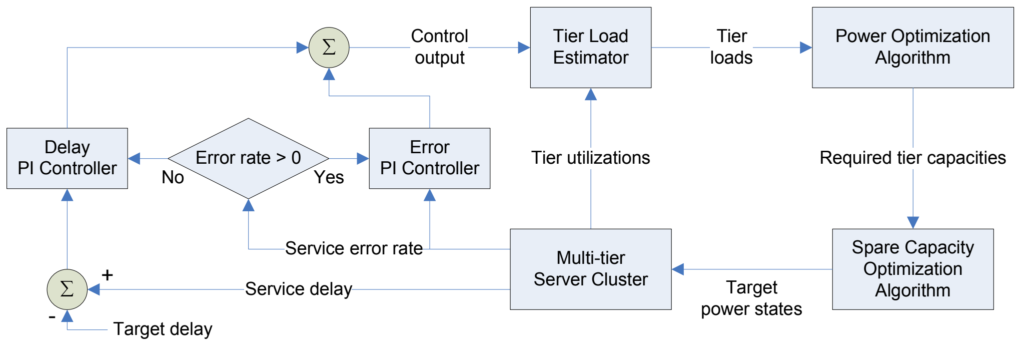


Figure 4.4: Design of the energy management policy with closed-loop control.

Our feedback control loop, sketched in Figure 4.4, is designed as follows. The controlled system (or “plant”) is a multi-tier server cluster. Each period, three kinds of measurements are taken from it: tier CPU utilization, end-to-end service delay, and service error rate. These are all easy to obtain in practical systems: end-to-end response time samples are collected at the front-end tier and averaged across servers (weighted by the number of samples from each server), while error rates are summed. We employ two Proportional-Integral (PI) feedback controllers: one responds to latency violations, and one to service errors. (The reason to separate them is that they require different gains.) If service errors were observed, their rate is fed to the error controller, otherwise delay error is calculated by subtracting the target delay D from the measured average delay, which is fed to the delay controller. The final control output is produced by adding the selected controller’s output and the other controller’s integral component. Since the service error rate can never go below its target (i.e. zero), the integral component of the error controller is exponentially decayed (halved) when the rate is zero.

Next, the tier load estimator takes the utilization measurements and calculates an estimated offered load for each tier using the method described in section 4.2.3.1, adding the control output (if positive). The increment is distributed between reconfigurable tiers proportionally to their measured utilization, which ensures convergence to the true load plus any compensation added by the controller to correct steady state error. Then, the estimated tier loads are input to the optimization algorithm (section 4.4.1), which determines the new energy-optimized allocation, reflecting minimal tier capacities required to maintain the required performance. Any extra machines not allocated are subject to being allocated as spare servers to absorb load bursts, or to being transitioned to a sleep state, which decision is handled by spare capacity optimization, the subject of chapter 5. Finally, the determined target power states are applied to the cluster, affecting the service and thereby closing the loop.

4.5 Prototype

We implemented our energy management policy within our ClusterControlWare framework, described below.

4.5.1 Overview of ClusterControlWare

We developed a middleware framework that greatly simplifies the implementation of control policies for server clusters. It allows the programmer to create sensors to measure the system, actuators to perform elementary adjustments on it, and controllers that may invoke these across cluster nodes. For instance, our energy management policy is implemented as a controller, which invokes sensors to take measurements and actuators to effect power state transitions. The software allows flexible definition of cluster nodes, including what objects (sensors, actuators, and controllers) need to be created on each and their parameters. It directly supports multi-tier clusters via node group definitions, where groups may have additional objects that are only created on member nodes. For example, a group can be defined for a Web server front-end tier with a sensor to measure end-to-end latency. Groups can further have power and performance parameters (plugging into our models), as well as a specified set of allowed CPU frequencies to use for DVS.

We also implemented support for multi-tier cluster reconfiguration, by allowing groups to be marked static or dynamic. Dynamic groups may have new members added or existing members removed during operation, with the framework taking care of automatically setting up all necessary services. To achieve this, groups must define a service actuator that can start or stop the service, and a corresponding service sensor that can report whether the service is running. The service actuator must also provide for configuring the load balancer of the service, if any, to add or remove backend nodes started or stopped in the next tier. For example, after starting a new application server, the load balancers of all servers in the previous tier (e.g., Web servers) must be updated to start using it. Reducing the need for overprovision, this solution also enables the migration of nodes among tiers (if allowed by the user), provided that multiple tier applications are installed on the nodes, and, in addition to the services, all group-specific sensors and actuators are started and stopped as well. Idle nodes belonging to dynamic groups may be shut down and at later time waken up as they become needed. All of this is supported through complex node and membership state handling—in fact, much of the code is devoted to this. ClusterControlWare currently consists of slightly less than 5,000 lines of Python code.

Our framework also acts as a research tool, by collecting traces showing cluster state over time, such as: demand, capacity, estimated load, service errors, service delay, machine states, group membership states, etc. We have used these cluster traces extensively during our experimental studies to evaluate the behavior of our algorithms.

4.5.2 Spare Capacity

Since realistic Web workloads are bursty, some of the idle nodes must be kept active to accommodate short-term load spikes. However, keeping idle machines alive is costly due to their static power consumption. Hence, a balance must be found in the trade-off between energy savings and responsiveness during bursts. Here we simply note that our prototype contains an optimization algorithm that automatically determines the minimum number of spare idle machines necessary to maintain a level of responsiveness, and refer the reader to chapter 5 for the details.

4.6 Experimental Results

We evaluate our policy on a realistic testbed with commonly used server software in a typical multi-tier setup. We use a highly dynamic Web workload that is an appropriate application of this kind of system, and we simulate the load fluctuations observed in real-life Web server traces. Performance and energy efficiency of our policy is measured, and the benefits of dynamic tier allocation are quantified by comparing three schemes: (*i*) traditional (energy-oblivious) provisioning, (*ii*) energy-conscious static capacity planning, and (*iii*) our dynamic energy-optimizing allocation policy.

4.6.1 Experimental Setup

Our testbed uses a 12-node cluster composed of Linux-based PCs. Each PC has a DVS-capable AMD Athlon 64 processor, which supports the frequencies 1.0, 1.8, 2.0, and 2.2 GHz, as well as an Ethernet interface that supports the Wake-On-LAN protocol for remote wakeup capability. The machines also have 512 MB RAM, a hard disk, and a simple display adapter, but no monitor in order to keep the setup more realistic. A Watts Up Pro power meter is connected to the main power strip, measuring the AC power draw of the entire cluster.

The cluster is set up as a 4-tier Web server consisting of the following tiers:

1. A front load balancer tier with 1 machine statically allocated. It runs the Linux Virtual Server (LVS) [90] high-performance layer-4 (TCP-level) load balancer. Packet forwarding to the backend servers is set up in direct routing mode, the most scalable of the available mechanisms. The power draw of this tier was not measured because in a real-life cluster it would represent a much smaller fraction of total power than in our setup, thus our results would be skewed if we included it.
2. A Web (HTTP) server tier with dynamic machine allocation. This runs Apache [78] with the `mod_jk` connector module, which forwards dynamic requests to the load balanced backend servers.
3. An application server tier, also with dynamic machine allocation, which runs the JBoss enterprise Java application server [71]. It uses the MySQL Java connector to forward queries to the backend database cluster.
4. A database cluster tier with 3 machines statically allocated. The MySQL Cluster scalable database package [62] is used with 3 data nodes, over which the data set is partitioned. An SQL node (through which the cluster can be accessed) is co-located with each application server.

For each tier, standard versions of server software are used with no modifications necessary. We developed a loadable module for Apache, which reports end-to-end service latency samples to ClusterControlWare through a System V message queue. We also implemented scripts to start, stop, and check each service and to add or remove backends to LVS and `mod_jk`.

4.6.2 Parameter Selection

Selecting appropriate control parameters is very important to get good performance response. Our goal was to design a highly responsive controller, therefore we opt for an 8-second control period that can be considered very short in server applications. Hence, care must be taken when choosing the controller gains to avoid unstable behavior (i.e., oscillations between extremes). Since the controlled process is complex and dynamic, it is preferable not to rely on a detailed model when determining the gains.

We use the classic Ziegler-Nichols tuning method [93], which although imperfect [30], has the advantage that it requires very little information about the system. The two separate PI controllers must be independently tuned, with the other disabled in both cases. Further, the delay controller is tuned with a workload that does not cause server errors. In the final controller, the total integral component is clipped to between 0 and 8.8 GHz, which is approximately 2/3 of the total reconfigurable capacity in our cluster. Such clipping is typically performed to prevent excessive response in Integral controllers that may exhibit steady state error due to system constraints (e.g., not enough machines are available to reach the target).

After experimenting with our system, we chose 250 ms for the average latency target, which is reasonable both in terms of user expectations of an interactive website, and compared to the 120–140 ms baseline with no energy management.

4.6.3 Trace-driven Dynamic Workload

A 3-tier implementation of the TPC-W benchmark [82] was used as the test workload. It models a realistic bookstore website where all types of pages are dynamically generated using fresh data from the database, and also contain typical web images (buttons, logos, item thumbnails, etc.). The server establishes sessions with shopping carts, and ensures all transactions within the same session are routed to the same application server (session stickyness). A separate client machine is used to generate load for the servers. It can emulate hundreds of independent Web users, each navigating through pages as guided by a stochastic transition matrix. Retrieving a page and all images referenced by it is defined as one *web interaction*. Server throughput is measured in web interactions per second (WIPS), and latency in web interaction response time (WIRT).

The load imposed on the system by the TPC-W application is determined by the number of emulated browsers (EBs) the client executes. In performance-oriented studies (the original intent of the benchmark), this is sized to the maximum target server capacity. However, in an energy management study, we are interested in performance and energy efficiency during realistic operation, which includes significant periods of off-peak load conditions. A simple way to create such conditions is to run a smaller number of emulated browsers than what the system can handle. However, the results have limited practical interpretation because this creates a constant load as opposed to one that realistically fluctuates over time. A more desirable approach is to simulate the load variations found in real-life Web server traces, but using our dynamic benchmark—to create a trace-driven dynamic workload. Note that it is not possible to simply replay arbitrary server logs because the content they serve differs from our benchmark's content.

Our solution is to calculate the hourly average loads (in requests/s) for the trace in question, scale each value by a constant factor to obtain a corresponding number of EBs such that the peak equals to a given value (e.g., the maximum EBs the system can handle), and then run experiments in which the number of EBs is varied over time to match the resulting sequence. Between data points, we linearly interpolate to achieve smooth fluctuations. This technique also allows easily speeding up the traces in order to significantly reduce the time required for experiments. Figure 4.5 demonstrates that the resulting throughput profile closely matches that of the source trace shown in Figure 1.1(a) (on page 3). We use a speedup factor of 5$\times$, by which we can run this day-long trace in 4.8 hours. The same trace-driven approach was used by Rajamani and Lefurgy [69] (but with a much higher speedup factor of 48$\times$).

4.6.4 Performance and Energy Efficiency

We use our trace-driven TPC-W workload to evaluate our policy. For a cluster energy study to be comparable, it is important to scale the workload such that the peak requires close to full cluster capacity. After experimenting with our system, we determined that a peak of 400 EBs is appropriate.

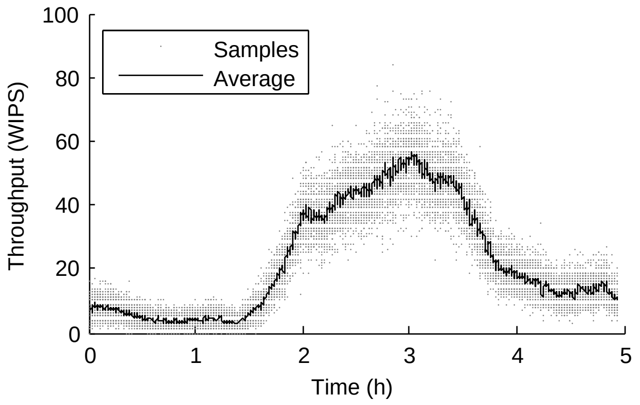
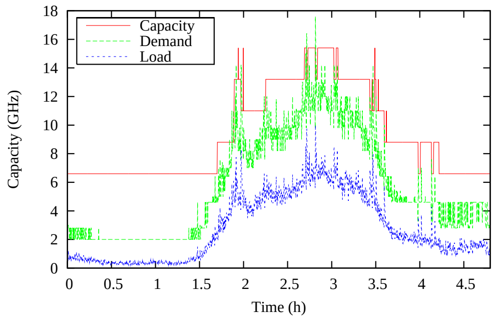


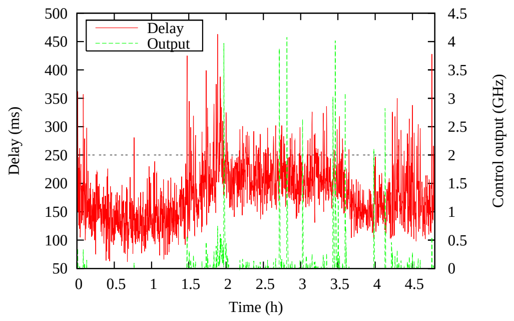
Figure 4.5: Throughput using our dynamic workload simulating fluctuations in the EPA Web server trace. Compare with Figure 1.1(a).

Our performance results are shown in Figure 4.6. In Figure 4.6(a), we see the actual cluster capacity dynamically varied over time, together with the required capacity (demand), which is computed by our algorithm based on the estimated load. As expected, the algorithm leaves sufficient margin over the estimated load when determining demand, so that tier utilizations stay low enough to meet the target latency. Figure 4.6(b) shows that this is successful even with little controller involvement in the steady state. Average delay (WIRT) was 227 ms, below the 250 ms target. Some of the controller reactions are due to short bursts of server errors caused by a limitation in our implementation: sessions belonging to an application server that is brought down are lost, resulting in failed next requests in those sessions. This could be alleviated, for example by existing session management solutions. Even with this limitation, we saw less than 2 errors per minute on average.

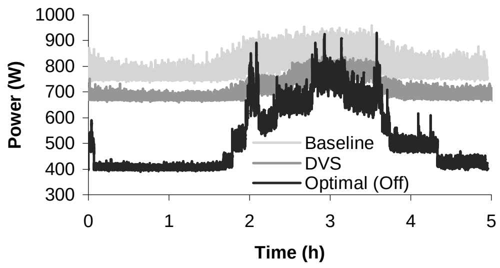
To evaluate the energy benefits, we ran this workload on the same cluster in two other setups. The Baseline setup is a cluster that is statically provisioned for our peak load, i.e. all 12 machines are operating at the maximum frequency. Note that even this basic setup is energy-aware in the sense that no extra machines are provisioned beyond this particular workload's peak demands. The DVS setup uses our policy but without dynamic cluster reconfiguration, i.e. it also operates all 12 machines, but allows DVS to save energy. Power traces for our optimization-based policy ("Optimal") and the two other setups (Baseline and DVS) are compared in Figure 4.6(c). Looking at pure cost reduction, our policy achieved 34% total cluster energy savings by dynamically turning



(a) Evolution of total cluster capacity (available machines), demand (required capacity as computed by our algorithm), and estimated load.



(b) Delay control performance: evolution of request latency and control output.



(c) Comparison of cluster power traces between a static provisioning baseline, using DVS only, and using our optimized energy management policy.

Figure 4.6: Power and performance results running the trace-driven workload.

off some machines when fewer could handle the load. To compare energy efficiency, we must factor in performance degradation as well. We use the energy-delay product, a widely used metric, defined in terms of experiment duration T, total energy consumed E, average power P, number of web interactions I, and throughput $WIPS$ as follows:

$$\frac{E}{I} \times \frac{T}{I} = P \times \frac{T^2}{I^2} = \frac{P}{WIPS^2} \tag{4.19}$$

in other words we use the energy per request multiplied by the inverse of throughput, which we calculate from average power divided by the square of throughput. From this, we obtained 0.996 Js for our policy, a substantial improvement over the energy efficiency of the baseline with 1.475 Js.

The key observation from our results is that our Optimal policy is successful in managing reconfigurable cluster capacity. When load increases, capacity is timely increased such that it is always sufficient to meet the performance target. End-to-end latency is kept close to the target even as the load fluctuates. At the same time, substantial energy is saved compared to both a statically provisioned case and also to using DVS alone.

We ran another experiment with a much lighter peak load (100 EBs), in order to measure how much energy could be saved by our policy during extended low-load periods (e.g., weekend days). We used the same trace accelerated to $60\times$ to shorten the duration of the experiments. We measured 40% total energy savings, which can be seen as the maximal benefit of the policy at this cluster size (note, in a larger cluster there are greater opportunities under such conditions). Our policy sustained 199 ms average latency, well under target (the baseline was 121 ms). Energy efficiency was improved from 22.77 Js to 14.58 Js.

4.6.5 Dynamic Tier Allocation

We are also interested in measuring whether dynamic tier allocation can save more energy than some arbitrary static scheme. Note that the static scheme is not overprovisioned, it is an energy-conscious policy merely restricted in capacity allocation between tiers. For the comparison to be general, we choose not to implement one particular policy but to emulate an "oracle" policy that chooses the best static allocation for each test workload. We implement this by running multiple experiments with different static allocations, searching for one that uses the least energy and still meets the SLA. We then compare our dynamic policy against that result.

For these experiments, instead of a trace-driven workload, we use a slightly modified TPC-W workload at constant levels. Our modification creates a gradually changing imbalance between the Web and the application server tiers. This simpler (but still highly dynamic) workload allows more straightforward comparisons than if we had used a fluctuating trace. The results are shown in Figure 4.7. At lighter loads, there is not much opportunity to improve on the optimal static allocation because only very low capacity is needed (most machines are sleeping in both cases). However, at greater loads, we obtain additional savings up to 6%. The performance penalty is lower than 35 ms above the SLA. The amount of extra savings also depends on how efficient the static allocation is at the given load level—for example, at load 200, we gain only about 2% because the best static allocation happens to be very efficient. It should be noted however that in practice such ideal policy cannot be implemented. The main point is that whenever the balance of tier loads can shift over time (e.g. the request mix is time-dependent), significant benefit can be realized by our policy over *any* static allocation policy.

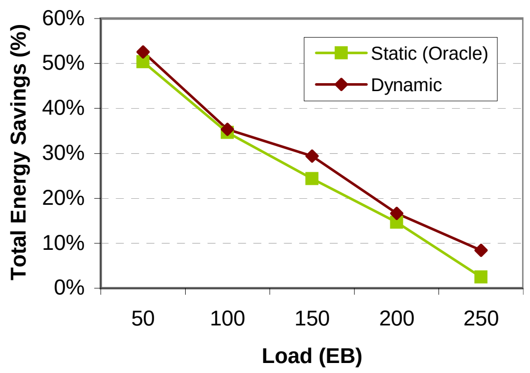


Figure 4.7: Energy comparison of the dynamic allocation policy against ideal static allocation. The baseline is static provisioning for the peak load (i.e. all 12 machines operate at full frequency).

4.7 Related Work

DPM techniques for dynamic reconfiguration of components in individual machines were surveyed by Benini et al. [8]. These techniques in general are not appropriate for throughput-oriented workloads because they rely on exploiting completely idle periods of operation. They do not address energy optimization with respect to a variable number of power-manageable, load balanced components, a key concept in the server domain.

Pinheiro et al. [68] design a power management scheme for reconfigurable clusters based on measuring and distributing demand for resources to a subset of the cluster. Transient demand variations are compensated for with PID feedback control. They evaluated the scheme on a Web server serving static content with no sessions involved. They report energy savings of 45%, however their workload was network interface-bound with peak CPU utilizations of only about 25%. Elnozahy et al. [19] propose cluster reconfiguration combined with DVS, assuming a cubic relation between CPU frequency and power, evaluated by simulation. Based on this assumption, they conclude that the cluster uses least energy when all machines operate in an optimal frequency range, not necessarily using the least number of active machines. This is in contrast with our observations based on our power model derived from actual measurements. The work by Rajamani and Lefurgy [69] is highly relevant to ours. They point out critical workload factors that must be properly selected for valid evaluations of energy management schemes. Hence, we use a similar methodology in our evaluations.

Heterogeneous clusters were also addressed in the literature [32, 72]. Heath et al. [32] developed a solution based on complex system modeling. Implementing an adaptive Web server that takes advantage of the model, they achieve 42% energy savings on a small cluster with two types of machines, serving mostly static requests. One shortcoming is that their model does not incorporate DVS. Rusu et al. [72] report 45% energy savings in a small heterogeneous cluster combining reconfiguration and local (independent) DVS. The rely on power, execution time, and server capacity characterization to provide energy management with soft real-time performance.

Direct comparison of our power and energy savings to the state of the art above is problematic for two reasons. First, hardware used in other studies may have lower idle power, which leads to higher savings *regardless* of the policy. Second, our multi-tier setup requires at least half of our small cluster to be active (1 machine per tier, plus the static DB tier), while other works may scale down to 1 active machine, increasing their savings. In essence, our setup has a more limited dynamic power range. This may be alleviated by consolidation of several services to a single machine, however this was not explored because in large-scale clusters the power consumption of one machine per tier is insignificant. For smaller-scale clusters, it could be addressed in future work. It is worth noting that our policy still achieves a better ratio between energy savings and maximum power savings (34% energy to 50% power) than previously reported results [68] (45% energy to 71% power).

Finally, our work is conceptually novel and different from the above in that (*i*) we address multi-tier clusters with highly dynamic workloads, (*ii*) we perform joint energy optimization using coordinated reconfiguration and DVS based on theoretical analysis, (*iii*) we use feedback control to correct for estimation errors, and (*iv*) our policy is evaluated with a realistic trace-driven benchmark with the peak load correctly sized to our cluster's capacity.

4.8 Conclusion

In this chapter we presented an energy management policy for reconfigurable clustered multi-tier servers. We have shown that the policy affords greatly increased energy efficiency compared with the conventional cluster setup provisioned for peak load. This can lead to energy savings up to 40% without significantly reducing service performance. Furthermore, our policy even outperforms *energy-aware* static allocation by up to 6%.

The key factor behind this result is the algorithm's theoretical foundation, which enables optimized automatic allocation of processing capacity between the tiers, from a pool of available machines supporting several (DVS) performance states. An implication of this is that offline (e.g., table-based) techniques that statically predetermine the optimal cluster configuration based on the overall load [e.g., 32, 72] are energy-inefficient under highly dynamic workloads, where a dynamic allocation policy such as ours is needed.

Chapter 5

Spare Capacity Optimization with Multiple Sleep States

5.1 Introduction

In a reconfigurable clustered tier, if serving the offered load with optimal energy efficiency is feasible using fewer machines than what is provisioned for the tier, then the extra machines may be put to sleep. The great power reduction potential of sleep states, however, obviously comes at the expense of completely stopped processing in nodes while asleep, but also during the time for shutting down and waking up (shutdown and wakeup latency). Additionally, these state transitions (to and from a sleep state) also consume significant energy. Hence, it must be determined whether the overhead of putting a machine to sleep is warranted, which problem was addressed in prior work [15] through quantifying machine reboot costs.

Dynamic, bursty workloads (such as typical Web workloads) pose further challenges by displaying heavy load fluctuations and unpredictable load spikes. The transitioning decision, therefore, must not disregard performance constraints: the machines awake must still be able to process load spikes within the specified response time limits. To that end, it may be necessary to keep some of the extra machines awake to provide spare capacity to absorb such load spikes. In this chapter, we examine how to optimize this spare capacity such that given performance requirements can be met with the least amount of energy. Research we are aware of has not addressed this problem beyond simple heuristics [e.g., 68]. Intuitively, the more spare capacity is left by the energy management policy, the more extra energy is spent, but also the better the system is at absorbing bursts. The optimal trade-off therefore also depends on how bursty the load is expected to be and how tolerant the user is to transient violations of acceptable performance. We provide a simple way to reconcile this energy-performance trade-off by allowing the user to specify the maximum burst rate to be still accommodated by the policy.

Extending the optimization space even further (and presenting additional complexity), modern machines offer *multiple*, increasingly deep sleep states characterized by decreasing power demands and increasing transition latencies. Finding the optimal amount of spare capacity in the presence of multiple sleep states in the server farm (such that spikes in the load can be accommodated) becomes significantly more complicated. We show that the optimal spare capacity that should be reserved to assure performance guarantees for future load with minimal power is a *combination* of spare capacity in several power states. The optimal combination can be determined as a function of the various wakeup latencies (including power-on latency for turned-off machines) in effect throughout the farm at the time, together with the predicted workload characteristics. Our results indicate that over 10% extra energy savings are possible with highly dynamic workloads when multiple sleep states are exploited.

5.2 System Model and Assumptions

We assume a reconfigurable multi-tier cluster composed of machines that support n system sleep states, each characterized by a distinct power level and wakeup latency. The power dissipation and wakeup latency of state i is denoted p_i and ω_i, respectively. We assume that $\forall i, 0 < i \leq n$: $p_i \leq p_{i-1} \wedge \omega_i \geq \omega_{i-1}$, where state 0 is the `On` state with $\omega_0 = 0$. As an example, recent ACPI-compliant [21] hardware may support up to 5 system sleep states (S-states). For instance, these include the Hibernate-to-RAM state, in which CPU state is saved to RAM and many components (such as CPUs and disks) can be shut down while RAM is still powered to maintain its contents, and the Hibernate-to-Disk state, where volatile program state (in the RAM and CPU registers) is saved to disk and most components can be shut down. Note that our model is not restricted to these states.

The most widely used approach to utilizing multiple power states in general is to start out in the shallowest state and progressively move to deeper states over time while the hardware is idle. We refer to transitioning to a deeper system sleep state as demotion[1]. We assume that direct demotion is not possible—the machine has to be first waken up and then transitioned to the desired new state. This is realistic since the OS must perform additional work during the transition. For example, when moving from Hibernate-to-RAM to Hibernate-to-Disk state, the OS must save memory content to disk. This means that moving to the best state at first is much more efficient than via progressive demotion.

With respect to the workload, we assume that load spikes are not predictable with sufficient accuracy that reconfiguration decisions could be based on this. However, we do assume that arbitrarily steep spikes are either not present, or need not be supported by the cluster. If this assumption does not hold, sleep states cannot be safely used to save energy because accommodating a step increase in load would require *immediate* wakeup of some sleeping machines. Intuitively, a trade-off exists between system responsiveness and the ability to exploit sleep states, which should be up to the user (administrator) to control. Hence, we define a system parameter called maximum accommodated load increase rate (MALIR) to be the maximal rate of increase of the demand for cluster capacity that must be met despite any power management activity. This essentially prescribes the lower bound of responsiveness and the upper bound of energy savings. For simplicity, we assume a CPU-bound workload with the demand and MALIR measured in CPU cycles per second (Hz). If another resource is the bottleneck, it can be defined for that resource analogously.

A reconfigurable cluster may consist of multiple independent pools, from which machines may be allocated to tiers on demand. Separate pools may be necessary to obey allocation constraints imposed by the software setup (i.e., presence or absence of software and data on certain nodes) or the network environment (e.g., location of firewalls), for instance. Assuming it is possible to migrate nodes between tiers within a pool with negligible latency cost, it is sufficient to limit demand increase for pools instead of for individual tiers. This is beneficial since it may smooth out load fluctuations at the pool level. We denote the MALIR of a pool v by σ^v.

[1]Note, in ACPI terminology, demotion for C-states means the opposite: transitioning to a shallower state.

5.3 Related Work

To provide some spare capacity to compensate for wakeup latency in their system, Pinheiro et al. [68] assume that machines have a lower than actual capacity. To control the amount of spare capacity, a heuristic parameter can be used. They experiment with values 30 and 15%, the latter being determined from the maximum rate of traffic increase for the trace they use (but no details are provided on how this was obtained). Rajamani and Lefurgy [69] study how the availability of spare servers affect energy savings in different workloads. They find that in workloads having steep slopes, spare capacity is necessary, and dedicated spare servers may even reduce cluster energy by allowing other machines to be better utilized. They also argue that the rate of change of the load profile is a key workload characteristic for energy management studies. They do not, however, attempt to derive the optimal number of spare servers. In their paper, Rusu et al. [72] define the `max_load_increase` variable as the maximum slope of the cluster's load, which can be obtained through characterization. Based on its value and accounting for the wakeup latency, they precompute a table of load values at which additional machines must be turned on.

The MALIR parameter we introduced in the previous section is conceptually similar to what was independently proposed in the above papers. A key difference is that they do not address multiple sleep states, which we exploit for further energy savings. In general, it is not straightforward to extend the techniques presented in previous work to multiple states, because the new trade-off between wakeup latency and sleep power must be examined.

5.4 Optimal Spare Capacity

5.4.1 Energy Minimization

An optimal energy management policy should be able to decide under what conditions it is worthwhile to enter some sleep state, such that the real-time performance constraints are satisfied with minimal energy consumption. This is determined in general by two factors: the spare capacity needed and the transition overheads (time and energy) involved. The spare capacity needed determines the upper bound of capacity to enter into each sleep state such that steady-state power is minimized while observing the MALIR constraint, whereas the overheads determine the timing of transitions that results in best overall energy efficiency. In terms of these components, total cluster energy is given by:

$$total\_energy = active\_energy + sleep\_energy + transition\_energy. \tag{5.1}$$

Assuming a reasonable energy management policy for the active cluster capacity, which involves determining the most energy-efficient cluster configuration, i.e. the set of machines to be kept active and possibly their DVS states, *active_energy* is dictated by load and it is not affected by the other terms in Equation (5.1). As long as sleeping machines can be waken up in time to meet demand, energy optimization of the rest of the cluster (i.e., the non-active nodes) can be decoupled and treated as a separate concern.

Minimization of *transition_energy* is considerably less important in typical Internet server workloads than in desktop systems. This is because load fluctuations occur on a larger time scale (i.e., daily or longer), which means transitions are infrequent and their energies remain an insignificant fraction of total energy. Prior works have implicitly ensured that this assumption holds by smoothing out short bursts using exponentially weighted moving averaging and imposing a minimum time between cluster reconfigurations [32, 68].

Therefore, we focus on minimizing the *sleep_energy* component through spare capacity optimization. Since it is decoupled from the other components, optimal steady-state energy is achieved by optimizing for power. This means, because of the property $p_i \leq p_{i-1}$, that each spare (unallocated) server should be put to the deepest possible state (greatest i), subject to the MALIR constraint.

5.4.2 Feasible Wakeup Schedule

Our problem becomes that of determining the minimum number of spare servers for each sleep state such that if the offered load increases with rate σ^v, a feasible wakeup schedule still exists. Before we can formulate this more precisely, we must begin with definitions. Let $c^v(t)$ and $d^v(t)$ denote cluster capacity and demand, respectively, for pool v at time t. Further let t_0 be the time at which load begins to increase. Assume that both $c^v(t_0)$ and $d^v(t_0)$ are known (e.g., from measurements and estimation). Then, from the definition of σ^v:

$$d^v(t) = d^v(t_0) + \sigma^v(t - t_0). \tag{5.2}$$

Next, in order to determine whether a feasible wakeup schedule exists, it is sufficient to consider the case when *all* spare servers are waken up at time t_0—if this does not result in a feasible schedule, then one does not exist. Let S_i^v stand for the number of spare servers in sleep state i, and $f_{\max}^v$ the maximum CPU frequency for pool v. For the case described, capacity can be obtained from the recursive formula:

$$c^v(t_0 + \omega_i) = c^v(t_0 + \omega_{i-1}) + S_i^v f_{\max}^v. \tag{5.3}$$

(The formula terminates at $\omega_0 = 0$.) Phrased in words, cluster capacity at the expiration of the wakeup latency of some state increases by the maximum capacity of all machines in that particular state. From this, and observing that $d^v(t)$ continuously increases while $c^v(t)$ is stepwise, we can formulate that a feasible wakeup schedule exists iff:

$$c^v(t_0 + \omega_i) \geq d^v(t_0 + \omega_{i+1}), \quad 0 \leq i < n. \tag{5.4}$$

5.4.3 Spare Servers

Since we want to put spare servers in the deepest possible sleep states, we examine the limit case of Equation (5.4):

$$c^v(t_0 + \omega_i) = d^v(t_0 + \omega_{i+1}), \tag{5.5}$$

so that no slack time is allowed before waking up spare servers in case of a load spike. Because this requires the set of greatest still feasible wakeup latencies, it also results in the smallest feasible overall power due to the monotonicity of ω_i and p_i in i. Substituting Equation (5.5) into Equation (5.3), we get:

$$d^v(t_0 + \omega_{i+1}) = d^v(t_0 + \omega_i) + S_i^v f_{\max}^v, \tag{5.6}$$

which can be simplified by applying Equation (5.2) to get:

$$\sigma^v \omega_{i+1} = \sigma^v \omega_i + S_i^v f_{\max}^v. \tag{5.7}$$

After rearrangement, we obtain the optimal number of spare servers for each sleep state as follows:

$$S_i^v = \sigma^v \frac{\omega_{i+1} - \omega_i}{f_{\max}^v}. \tag{5.8}$$

The above formula is in the continuous domain, whereas *a*) cluster reconfiguration is typically performed periodically; and *b*) only whole machines can be transitioned to sleep states. Hence, the formula is discretized to yield the final solution:

$$S_i^{v*} = \left\lceil \sigma^v \frac{\mathcal{T}(\omega_{i+1}) - \mathcal{T}(\omega_i)}{f_{\max}^v} \right\rceil, \quad 0 \leq i < n, \tag{5.9}$$

where $\mathcal{T}(\omega) = \lceil \omega/T \rceil \cdot T$ and T is the control period. This accounts for both that wakeups will only be initiated, and that the cluster will only be reconfigured to include the awakened machines, at controller activation.

Finally, it is worth noting that, depending on the system, some sleep states may not be beneficial and should be excluded from the optimization. Specifically, if $p_i = p_{i-1}$, or if $\omega_i = \omega_{i+1}$, then state i does not provide any benefit. In addition, all states should be made optional since their individual usefulness may be application dependent.

5.5 Algorithm Design and Implementation

We developed two policies for comparison: Optimal, which is based on our optimization result above, and Demotion, which gradually demotes spare servers after some fixed timeouts. The reason for this choice is as follows. The state of the art in DPM techniques addressing multiple sleep states consists of predictive and stochastic approaches. The fixed timeout-based predictive technique is the most common power management policy [8]. It basically assumes that after an idle period of at least the timeout, the system will likely stay idle for enough time to compensate for the extra energy spent on the sleep and wakeup transitions. More complex predictive techniques have been proposed [e.g., 16, 17], but they target interactive workloads.

To ensure a fair comparison, a wakeup scheduling algorithm is used together with both policies that always wakes up machines when needed to meet the expected future load based on the MALIR parameter. The difference is that while the Optimal policy attempts to ensure that machines can be awakened by the time they are needed, the Demotion policy does not. Below, we describe the design of our policies.

5.5.1 Optimal Capacity-based Policy

The design of the optimal capacity-based ("Optimal") policy is relatively simple. An algorithm precomputes S_i^{v*} for each server pool and each system power state ($0 \leq i < n$) allowed by the user at initialization. Then, the policy is invoked in each control period with the list of nodes determined by the main energy management algorithm to be idle. Our policy is responsible for assigning these to sleep states as necessary. We start by checking if there are more idle machines in the `On` state than the optimum S_0^{v*}. If so, each surplus machine must be assigned to one of the sleep states. Going from shallow to deep states, we assign as many surplus machines to each as necessary for it to reach its optimum. Finally, any remaining surplus machines are assigned to state n (which is the implicit optimum for this state).

5.5.2 Fixed Timeout-based Policy

The main idea of the Demotion policy is that whenever a machine becomes idle a timer is started, and as it reaches each timeout, the machine is demoted into the corresponding sleep state. There are two problems with directly applying this approach in a cluster. First, it is not scalable to maintain individual timers for each machine in a large cluster; and second, unless the activation of idle nodes is always done in a predetermined order (e.g., by rank), it is suboptimal. To see this, consider an example where demand fluctuates between 1 and 2 machines, but whenever it increases to 2, a different node is activated in a round robin fashion (e.g., for thermal load distribution). If this happens frequently enough, timers may never reach any timeouts, even though most of the cluster is always idle.

Therefore, instead of per-node timers, our solution is to just maintain the *count* of idle machines together with the time each smaller count was first seen. More precisely, we define a (per-pool) list of timestamps named `idle_since`, which is initialized empty and then maintained as follows. If the number of idle machines at the current time t is greater than the size of `idle_since`, then t is appended to its end as many times as the difference. If it is smaller, then as many elements as the difference are removed from the list's end. Then, for each list element e, its timeout for state i has been reached if $t > e + \omega_i$. Summing up the number of elements by the deepest state they reached, we obtain the timeout count for each state. Taking the difference between the number of nodes currently in each state and that state's timeout count, we get the number of surplus nodes (or deficit) in each state. Then, working our way from the `On` state to deeper states, surplus nodes are demoted to states with a deficit, filling up deepest states first.

5.5.3 Service Preloading

In practice, sleep state wakeup latency is not the sole factor determining the time to activate a system (resume latency). Service start latency must not be ignored since it can be significant compared to the wakeup latency. For example, the start latency of the application server in our testbed was 25–72 s (95th percentile: 40 s), from over 2800 samples. This would have to be added to all wakeup latencies when calculating the optimal spare capacities.

However, with hibernation states, we can take advantage of the preserved program state they offer. As a first step, services should not be stopped before hibernation, so that after wakeup only the load balancers need to be updated. However, to also be able to reduce spare capacity requirements, one has to *ensure* that any spare servers contain the suspended service and thus can be activated faster. To this end, we *preload* the service prior to entering any hibernation state if not already loaded. This means that the service start latency only has to be added to the wakeup latency of nonhibernation sleep states (i.e. `Soft-Off`), resulting in smaller resume latencies and lower optimal spare capacities. Naturally, our wakeup scheduling algorithm is also aware of what services are suspended on each node and calculates time to activation accordingly.

5.6 Experimental Evaluation

5.6.1 Experimental Setup and Policies

Our experimental setup is identical to the one described in section 4.6.1 (on page 79). In addition, our PCs support four ACPI system power states:

- On (S0): Normal operating mode, with DVS available to manage power.
- Hibernate-to-RAM (S3): System state is saved in RAM, which must remain powered, while most other components can be shut down. On wakeup, no boot is required.
- Hibernate-to-Disk (S4): System state is saved on hard disk, and all components can be shut down (except those necessary for initiating wakeup) as in Soft-Off mode. A partial boot is required before the system image can be restored on wakeup.
- Soft-Off (S5): Same as the conventional (mechanical) `Off` mode, except that components necessary for initiating wakeup are powered. A full boot is required after wakeup, and all processes must be reloaded.

We have one pool of machines shared by the Web and application server tiers. For realistic evaluations, we use the same trace-driven workload approach based on the TPC-W benchmark as previously (see section 4.6.3). For each experiment, we calculate the correct MALIR for the pool, based on the trace and an estimate of the server capacity required per EB (this only needs to be determined once).

As our baseline, we consider a traditional cluster statically provisioned for peak load (i.e., not overprovisioned overall), which serves as our reference in energy efficiency and performance. We are interested in finding out what additional gains can be realized by using multiple states, over previous approaches that only exploit one state (S5). Hence, we experiment with two different cases for both the Optimal and the Demotion policy for comparison: a case when multiple states

Policy	Description
Baseline	Statically provisioned for peak load.
Demotion (Off)	Demotion policy using the `Off` state only.
Demotion (Multiple)	Demotion policy using Multiple states.
Optimal (Off)	Optimal policy using the `Off` state only.
Optimal (Multiple)	Optimal policy using Multiple states.

Table 5.1: List of policies compared.

are used (Multiple), and another with only the `Soft-Off` state allowed (Off). As we will see, in the Multiple case, the S5 state is not beneficial and thus it is not used. The policies are listed in Table 5.1.

5.6.2 Determining Optimal Timeout Values

For a fixed-timeout policy, careful selection of timeout values is crucial. We follow the methodology proposed by Benini et al. [8], who define the break-even time T_{BE} for each sleep state, which is the minimum inactivity time required to compensate for the transition energy. Based on the theoretical result on competitive algorithms for the analogous *spin-block* problem by Karlin et al. [42], they suggest the optimal choice for timeout (yielding 2-competitive energy savings) is equal to the break-even time. To obtain this for each state, characterization of the system is necessary to obtain the following parameters: time to enter and exit the state (T_{Enter} and T_{Exit}), average power while entering and exiting (P_{Enter} and P_{Exit}), and average power while in the state (P_{Sleep}).

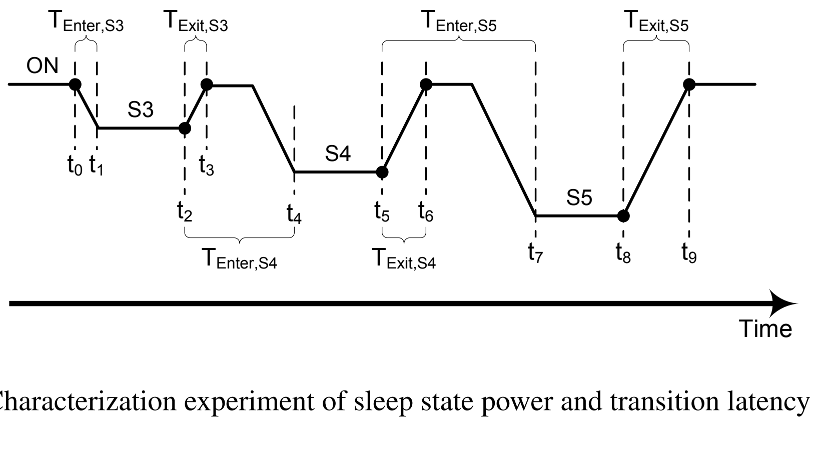


Figure 5.1: Characterization experiment of sleep state power and transition latency parameters.

We perform the experiment depicted in Figure 5.1 to measure the parameters. During the ex-

State	P_{Sleep} (W)	T_{Enter} (s)	P_{Enter} (W)	T_{Exit} (s)	P_{Exit} (W)	T_{BE} (s)
Multiple States						
S0	64.88					
S3	2.55	4.79	46.72	14.78	64.97	20
S4	1.43	26.35	64.73	66.17	77.43	5980
S5	1.43	84.24	73.70	83.40	75.42	∞
Soft-Off Only						
S0	64.88					
S5	1.43	18.06	60.76	83.40	75.42	114

Table 5.2: Sleep state parameters and break-even times.

periment, the time at points t_0–t_9 must be recorded, and power consumption must be continuously measured. Timestamping points marked with a dot is trivial, however the points demarcating when entering a state has finished (t_1, t_4, and t_7) can only be inferred from the power samples, which we performed during offline analysis. From the data collected, calculating the above-mentioned parameters is trivial. Finally the break-even times (i.e. the timeouts) are calculated using the formula given in [8]. A similar experiment is performed for the case when only the `Soft-Off` (S5) state is used. The parameters and results are shown in Table 5.2. Importantly, with multiple states allowed, the S5 state is not beneficial because its power level is identical to that of S4—correspondingly, its break-even time is infinity, i.e. it should never be entered. Hence, it is excluded from both policies.

5.6.3 Results

We perform comparison studies along three dimensions, to understand how certain aspects affect the performance of each policy. We consider different workload profiles, then varying the peak load intensity, and the rate of load fluctuations. The performance of each policy, including energy savings, energy-delay product (i.e. energy efficiency), average end-to-end latency, and error rate (i.e. timeouts), is compared. As explained in section 4.6.4, errors are partly caused by losing sessions of servers being put to sleep—a limitation of our testbed setup. However, in other cases they are signs of an important weakness of a policy: the inability to meet demand in time, which we will note.

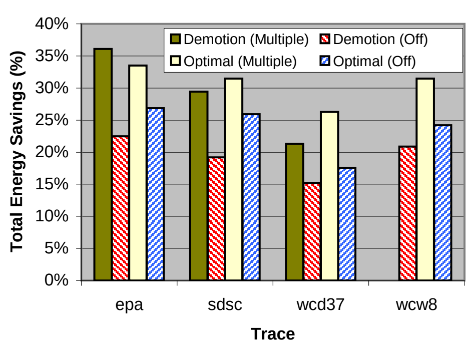
5.6.3.1 Shape of Load Fluctuations

To study the effect of different load profiles, we experiment with several different traces listed in Table 5.3. Each trace has a unique shape of daily load fluctuations, giving rise to different energy saving opportunities. The peak load for each trace is identical (400 EBs), sized to total cluster capacity. All traces are accelerated 20× to shorten experiments. Results are shown in Figure 5.2. Note, despite several attempts, the Demotion (Multiple) policy running the week-long `wcw8` trace invoked so many transitions over its 8 h 24 m duration that the (not fully stable) ACPI support in our test machines could not handle, causing a node to eventually crash, and so this data point is not available.

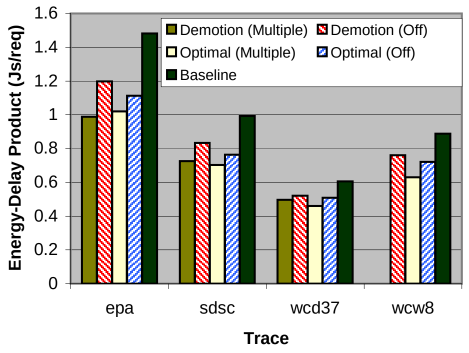
A number of key observations can be made. First, exploiting multiple states yields significant extra energy savings in the range 6–14% (Figure 5.2(a)). Average gain is 10% for Demotion and 7% for Optimal. Second, the workload sensitivity of Optimal (i.e., range of extra savings across workloads) is substantially smaller than that of Demotion (7–9% vs. 6–14%), which means a more predictable performance can be expected from it. Third, Optimal overall outperforms Demotion in all aspects (with one data point exception), by up to 7% in energy savings. An extended period of `epa` has such a light load that no spare capacity is needed at all, giving an advantage to Demotion. The other traces have a slightly smaller dynamic range (in load level), where that advantage is lost, and Optimal wins by putting most nodes into deep sleep immediately instead of waiting for a long timeout. Finally, due to the common wakeup scheduling algorithm, all policies manage to maintain the expected performance with the fluctuations present in these workloads. The error rates only reflect lost sessions, as explained above. (It is worth noting that the Off-only policies' failure to

Name	Length	Service
epa	1 day	The EPA WWW server at Research Triangle Park, NC; Aug. 30, 1995 (Wed).
sdsc	1 day	The SDSC WWW server at the San Diego Supercomputer Center in San Diego, CA; Aug. 22, 1995 (Tue).
wcd37	1 day	1998 World Cup Web site [4]; Jun. 1 (Mon).
wcw8	1 week	1998 World Cup Web site; May 3–9 (Sun-Sat).

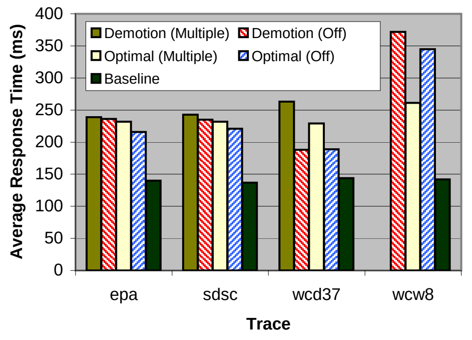
Table 5.3: Description of Web traces used in our experiments.



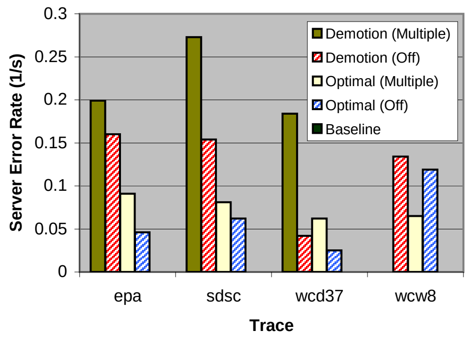
(a) Cluster-wide energy savings compared to the baseline.



(b) Per-request energy efficiency. (Lower values mean better efficiency.)



(c) Average client-perceived latency (WIRT). Target is 250 ms.



(d) Rate of errors (caused by timeouts or lost client sessions due to reconfiguration).

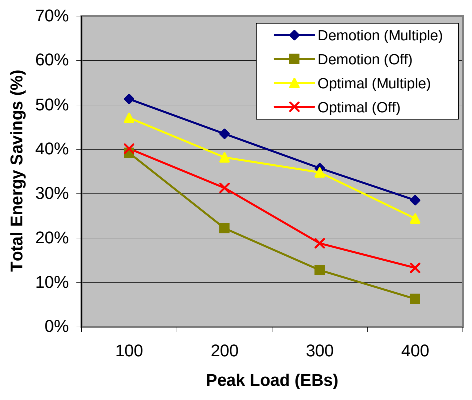
Figure 5.2: Comparison of the energy efficiency and performance of the experimental policies run against several traces. Peak load is 400 EBs (full capacity), traces are accelerated to 20×. Data for `wcw8` running under Demotion (Multiple) could not be obtained.

meet the latency target with `wcw8` is due to the fact that during the long experiment some of the reboots take unexpectedly long; for the most part, latency is well controlled.)

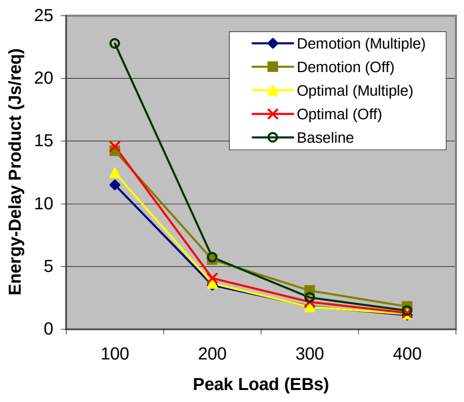
The most important points are that Optimal is better than Demotion for the majority of traces, and that using multiple states is always beneficial in terms of energy efficiency.

5.6.3.2 Peak Load Intensity

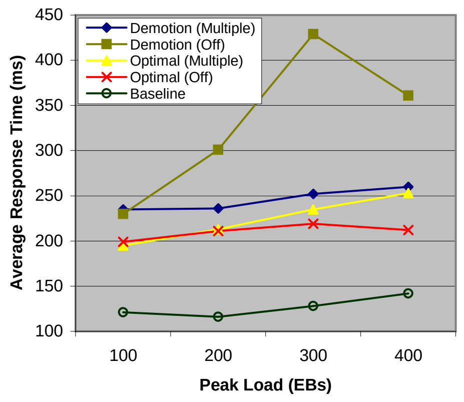
In many instances, the daily peak load does not approach cluster capacity, thereby increasing energy management opportunities. For example, there are weekly and seasonal load variations, or the cluster may be heavily overprovisioned. Although we cannot simulate realistic seasonal changes in reasonable time frames, we can run the same daily trace scaled to various peak levels to gain insight into policy performance at lighter loads. We use the `epa` trace accelerated to 60× and vary peak load from 100 (very light) to 400 EBs (full capacity). Results are shown in Figure 5.3.



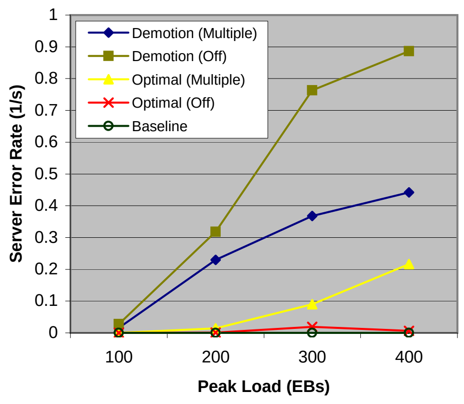
(a) Cluster-wide energy savings compared to the baseline.



(b) Per-request energy efficiency. (Lower values mean better efficiency.



(c) Average client-perceived latency (WIRT). Target is 250 ms.



(d) Rate of errors (caused by timeouts or lost client sessions due to reconfiguration).

Figure 5.3: Effects of varying the peak load intensity (from very light to full capacity) on energy efficiency and performance of different policies. The `epa` trace is used at 60× speed.

Again, we observe significant extra energy savings from exploiting multiple states, in the range 7–23% (Figure 5.3(a)); average gain is 20% for Demotion and 10% for Optimal. As seen in the previous results, for this trace Demotion (Multiple) outperforms Optimal (Multiple) by 1–5%. However, Demotion (Off) is highly inefficient—Optimal (Off) outperforms it by up to 9%. Moreover, with anything higher than very light load, it has *worse* energy efficiency than the baseline (more clearly seen in Figure 5.4(b)), and it exhibits poor performance: it fails to meet the target latency requirement and has a very high error rate. This is caused by the policy's inability to meet demand in time, because it does not leave the necessary spare capacity to absorb load while machines are waken up. In contrast, Optimal (Off) works with the same wakeup latencies, but has excellent performance because of its correct provisioning of spare servers. Note that there was no performance problem with Demotion (Off) in the previous experiments using a 20× speedup factor with the same trace. This raises the question of how this factor affects results, which we will explore in the next section. Also note that Demotion (Multiple) still has good performance (its errors are due to lost sessions only). The reason is that, even though it does not ensure sufficient spare capacity, wakeup from S3 in our system is sufficiently fast (< 15 s) in practice to accommodate fluctuations even at 60× speed.

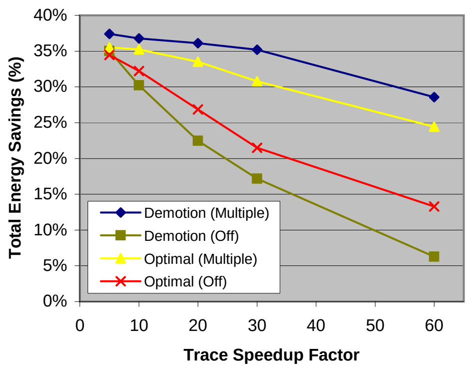
In summary, using multiple states improves energy efficiency, and the improvement is greater with higher peak loads. The reason is that higher peaks mean steeper load increases, requiring greater MALIR and more spare capacity, and multiple states allow more effective spare server energy optimization. Multiple states also help alleviate some of the performance problems of Demotion, because the impact of late wakeups is significantly reduced by the much shorter wakeup latency. Finally, it is worth noting that with peak load 1/4 of the full capacity, over 50% total cluster energy savings are achieved.

5.6.3.3 Time Scale of Fluctuations

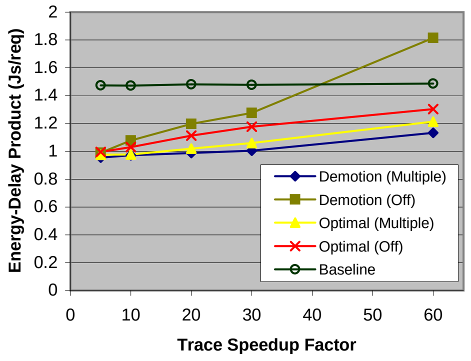
Previous trace-based experimental studies on cluster energy efficiency have picked arbitrary acceleration factors in order to shorten experiments, without evaluating its impact on the results. However, different factors yield workloads where the time scale of fluctuations is different relative to system invariants such as wakeup latencies, service startup times etc., which cannot be accelerated. As we have seen in the previous section, this can lead to marked differences in policy performance. Therefore, it is imperative to study sensitivity with respect to the speedup factor.

The `epa` trace is used again, peak load scaled to full capacity (400 EBs). Each policy is tested with several acceleration factors: 5×, 10×, 20×, 30×, and 60×. The results are presented in Figure 5.4.

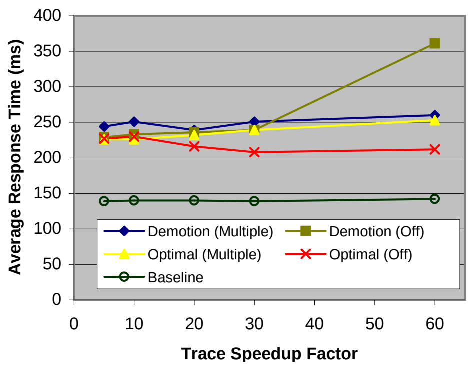
The most important observation is that the speedup factor has a major influence on energy



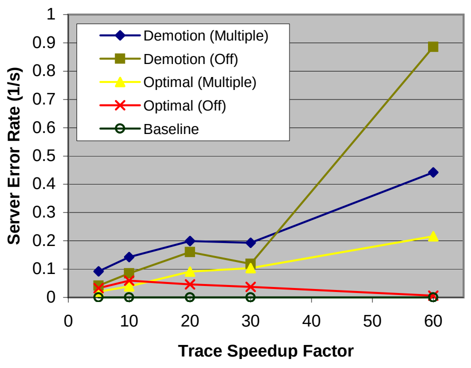
(a) Cluster-wide energy savings compared to the baseline.



(b) Per-request energy efficiency. (Lower values mean better efficiency.



(c) Average client-perceived latency (WIRT). Target is 250 ms.



(d) Rate of errors (caused by timeouts or lost client sessions due to reconfiguration).

Figure 5.4: Effects of different trace acceleration factors on energy efficiency and performance of different policies. The `epa` trace is used with peak load of 400 EBs (full capacity).

savings and efficiency. The reason is that the higher this is, the greater the MALIR must be to maintain performance, which in turn requires more spare capacity, resulting in diminished energy savings. How optimally the policy handles that MALIR requirement, is what makes a difference in energy use. Specifically, policies using multiple sleep states handle it very efficiently because lower transition latencies of shallower states allow them to enter *some* sleep state for a greater portion of time, and because even the shallowest state affords significant power reduction. Conversely, with greater time scale (slower) fluctuations, little spare capacity is necessary, and the difference between policies becomes minor (assuming a MALIR-based wakeup scheduling algorithm is used). The results also show that spare capacity optimization is especially important in Off-only systems with high rates of workload fluctuations, where a Demotion-like naive policy should clearly not be used because of significant performance degradation. However, it should also be noted that if fluctuations (and load increase) are very slow, good results can be achieved even by simple Off-only policies.

5.7 Conclusion

Energy management for reconfigurable clusters can be separated into two independent concerns: the management of active and spare capacity. Spare capacity optimization minimizes energy of idle nodes subject to responsiveness constraints. In contrast with existing ad-hoc approaches only addressing a single `Off` state, we theoretically analyzed the problem in the context of multiple sleep states, and designed an optimization policy based on the results.

Our policy was validated by measurements on a realistic testbed with appropriate workloads and using actual hardware sleep states. Extra energy savings of up to 16% over the Off-only version of the same policy were observed in highly dynamic workloads with negligible performance impact. We also carefully designed and implemented a fixed-timeout policy (the most widely used predictive policy) optimized for clusters. Comparisons with different traces show that our policy outperforms it in energy efficiency with the majority of workloads. In addition, our policy is superior because it: (*i*) avoids unavailable periods inevitable with timeout-based policies due to late wakeups; (*ii*) guarantees responsiveness to the user-specified level (MALIR) with the least amount of energy; and (*iii*) does not rely on workload prediction, and therefore handles unexpected bursts and typical expected load fluctuations equally well.

An important implication of our results is that energy savings reported from policy studies using different trace acceleration factors may not be directly comparable (unless equal spare capacity was provisioned). A sensitivity analysis with respect to this factor is desirable to help interpret such results. Further, it is not clear how results from accelerated traces in small testbeds apply to real-time workloads in large-scale clusters because, even if spare capacity is the same, it represents a very different fraction of total cluster energy.

In this work, we focused on minimizing sleep energy, decoupled from transition energy. However, transitions typically incur machine wear costs as well, and therefore minimization of both the number and energy of transitions is a desirable goal. In future work, we would like to integrate the analysis presented here with a transition cost model, such as:

$$transition\_cost = transition\_energy + machine\_wear\_cost,$$

which will allow cost savings beyond energy. Machine wear cost has been quantified and controlled in existing work using heuristics that minimize the number of reboots over time [15]. We expect such a model to further emphasize the benefits of our optimizing policy, which inherently avoids superfluous transitions typical of the (purely opportunistic) timeout-based policies.

Chapter 6

Fault Tolerance of Power Management Algorithms

6.1 Introduction

A significant concern in connection with machine performance control (e.g. as part of an energy management policy) is how faults affect the control strategy. It is desired that whenever external disturbances are present, the controller automatically attempt to minimize their effects on the system. Disturbances can come from different sources: for example, simple machine or component failures, CPU throttling due to thermal emergency, or even from malicious load injection following a security breach.

Pipelined parallelism refers to the coarse-grained parallelism of a computation that has been partitioned into processing stages (e.g., producer and consumer processes). A major source of performance gains of pipelining come from the ability to replicate processes in a given stage, allowing parallel processing of the incoming workload (i.e. scalability). For example, in multicore systems, compile-time techniques have recently been proposed to statically allocate cores to processes [27, 79]. Another form of pipelined parallelism is multi-tier processing in server clusters, where requests flow through different types of servers before a response is sent to the client. Here, allocation of machines to tiers (stages) is typically done as part of capacity planning or through dynamic provisioning [89].

The growing importance of power management, not only in data centers but also in individual CMP machines, is widely known. At the chip level, area is scaling down faster than power due to non-ideal voltage-supply scaling and non-ideal capacitance scaling. This means greater power density and, for same chip area, greater total power [40]. Thermal constraints will likely require aggressive DVS scaling in future CMP machines with a large number of cores [49]. Hence the role of the power management policy will be emphasized. However, unexpected conditions (e.g., thermal overload) may interfere with it by influencing the same power management controls (actuators) that the policy uses, such as DVS, without its knowledge. For example, a thermal overload condition may override any application-requested DVS state, inhibit higher power states (or even enforce a factory defined low-frequency state in an attempt to cool the processor core). Such transparent reductions in machine processing capacity may produce severely suboptimal power management behavior if the policy is not tolerant with respect to them. While it is highly desirable for a policy not to degrade the performance of the original system significantly, past works have not evaluated robustness under such conditions. In this chapter, we consider the fault tolerance of power management policies in multi-tier server clusters because they represent an application category of huge commercial importance.

Feedback control is a proven method grounded in theory, which has been shown effective for server performance control in general [15, 32, 65]. Its key strength comes when combining it with models constructed from *a priori* analysis, where it can improve a complex system by compensating for inevitable model or parameter errors. While prior work has applied feedback control in pipeline-parallel systems [e.g., 65], the effectiveness of this approach for overload or anomaly response has not been studied. In the following sections, we shall demonstrate that by utilizing feedback control, our algorithm remains effective even despite such adverse events. This result is not immediately apparent since the behavior of a feedback loop when the system's actuators are partially disabled can be affected in non-obvious ways.

The importance of this result, with respect to the energy management solutions described in previous chapters, comes from the fact that they all incorporate DVS, a technique that is directly impacted by dynamic thermal management, and load estimation, which is impacted by aberrant load. Since both assume an integral role in performance control, it must be investigated whether performance goals can be met despite these impacts. The main contribution of this chapter is the empirical investigation of these issues. We experimentally evaluate our policy's fault tolerance properties under capacity-limiting overload conditions. Our work shows that besides realizing excellent power savings, it also has robust response against aberrant behavior. We further demonstrate via comparison with an open-loop version of the system that closed-loop control is essential to this property.

6.2 Related Work in DTM

Dynamic thermal management (DTM) has become an active research area due to increasing power densities that may result in thermal emergencies. Temperature-aware workload distribution has been proposed to prevent and deal with such thermal overload events [59]. This approach was shown to be useful for reducing cooling costs over a long period by optimizing the placement of workload units among machines in a data center. The algorithms typically require long calibration experiments. Our work is different in that our algorithm does not directly affect workload placement, only the active capacity (number of active machines and their frequencies), driven by performance-constrained power optimization. A combination of the two solutions could further improve the thermal properties of our algorithm by the controlled selection of nodes in the data center to provide the necessary capacity.

Weissel and Bellosa [86] studied application-level DTM techniques to prevent emergencies and to optimize system-wide energy efficiency. However, their work does not address sudden changes in ambient temperature (e.g., due to equipment failure) that trigger immediate action to reduce core temperature, and its performance ramifications. A related approach [44] explored the performance benefits of application-level DTM that proactively reduces core temperature so that the penalties of hardware DTM can be reduced. Our work instead focuses on performance evaluation of our algorithm after hardware DTM has been activated.

6.3 Baseline

For our baseline evaluations, we use the same energy management policy and experimental setup as in chapter 4. We slightly modify the test parameters as follows. The target delay (i.e., setpoint) for the PI controller is increased to 5 s. Correspondingly, various server timeouts are increased to prevent server errors resulting from overload as much as possible. The clipping of the integral component of the controller was also increased from 8.8 GHz to 15 GHz, which is approximately the total reconfigurable capacity in the cluster—this will allow larger controller reactions necessary to deal with larger errors effectively. We do all of this because, for these evaluations, we are interested in observing aberrant load effects rather than ensuring optimal performance.

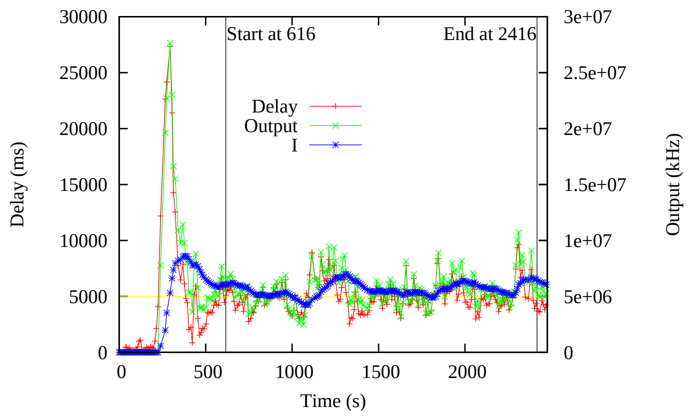


Figure 6.1: Baseline delay with proposed feedback control and no aberrant load. "Output": total output of PI controller; "I": integral component of PI controller. The delay target is 5000 ms. Start and End marks delineate the measured period (after warm-up).

With no unexpected overload events, Figure 6.1 demonstrates that the algorithm causes the latency to quickly converge around the setpoint (5000 ms) and maintains it until the end of the experiment. This is achieved by keeping the number of active machines relatively steady, with only minor variations reflecting those in the client workload. We also observed that the application server machines spent most of their time at the two highest available CPU frequencies (2 and 2.2 GHz). In the next section, we will examine the consequences of a thermal overload, which forces these machines to run at their lowest possible frequency.

It is worth noting that the algorithm achieves the expected level of energy savings of up to 45%

in a lightly loaded cluster, with unchanged throughput and only a negligible increase in average server delay, still well within tolerable limits at little over 300 ms.

6.4 Thermal Overload

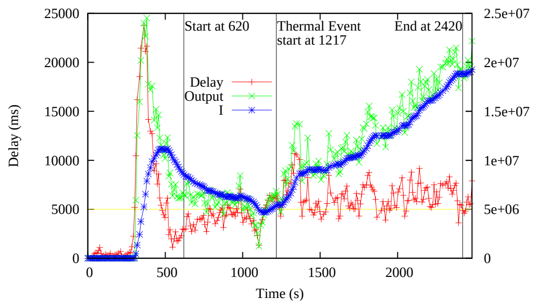
6.4.1 Hardware DTM Scenario

Thermal overload occurs when a thermal sensor on a processor core detects that one of the factory preset temperature limits has been exceeded. In the simpler case, there is only one limit, which protects the processor from damage and causes it to immediately shut down once exceeded (this is called a thermal emergency). However, in some (as of this writing, Intel) designs, there is an additional limit, which serves as an early detection of a possible future thermal emergency, and instead of immediate shutdown, it forces the processor to enter a power-saving state. In the case of Intel's TM2 technology, the processor will first attempt to cool itself down by entering its lowest-frequency DVS state. If this is not sufficient, clock cycle modulation (throttling) will be engaged subsequently.

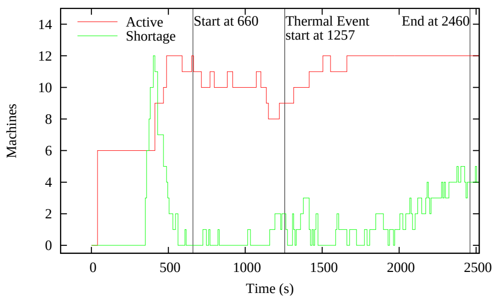
To create a reproducible scenario, we emulate the first measure taken in TM2. To do so, we utilize a feature of the Linux frequency scaling framework that allows one to restrict the range of allowable frequencies. At some point during the experiment (after 20 minutes), we simply set the top of this range to the minimum supported frequency of the processor in each cluster node. This simulates a global cooling system degradation, which affects an entire set of machines. We verify that all machines suddenly drop to the minimum frequency (1.0 GHz) as a result, drastically reducing server capacity. Although this is not an entirely realistic scenario in a well-designed data center, it serves as a worst-case example, where algorithm performance is examined with respect to the greatest possible step decrease in capacity.

6.4.2 Thermal Overload Results

The algorithm's response, specifically the PI controller output and cluster capacity evolution can be examined in Figure 6.2. Immediately after the beginning of the thermal overload condition, server



(a) Delay control performance.



(b) Trace of cluster capacity.

Figure 6.2: Algorithm behavior before and during thermal overload.

delay sharply increases because the frequency reduction causes a proportional increase in CPU utilization. Hence, as the total estimated load remains approximately constant, the optimization algorithm computes that it is only necessary to increase the frequencies back. However, as the machines reject any frequency increases and delay continues to exceed the target, the feedback controller keeps increasing the estimated load for the optimization algorithm until it computes that additional machines need to be activated to meet that load.

As the figures show, this relatively quick controller reaction activates the available (sleeping) machines, thus successfully managing to reverse the surge, and continues to maintain a stable latency level about the target. Although the latency slightly exceeds that of the baseline, notice that no further spare capacity exists, all 12 machines are already activated and therefore additional delay reduction is not possible. Further note that the algorithm is completely unaware that the frequencies temporarily cannot be increased (and in fact assumes they can), but nevertheless performs exactly as expected. The reason is that as long as the optimization algorithm is unable to bring down latency through attempts at increasing the frequencies, the integral part of the PI controller accumulates and ensures progressively larger capacity additions (resulting in additional machine allocations) by producing continuously increasing adjustments to the estimated demand.

For additional insight, we also compared the algorithm with the same system running under open-loop control. We performed an exhaustive search via experiments to find the best machine and frequency allocation for a particular offered load. Then, we emulated the same hardware DTM event as before, but now with the cluster running at that allocation. As shown in Figure 6.3, even though the best allocation has much greater margin for the delay, it suffers a huge performance hit after the event. This demonstrates the importance of closed-loop control in dealing with thermal overload and DTM.

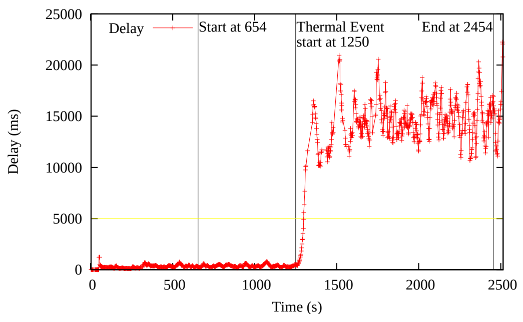


Figure 6.3: Open-loop delay during thermal overload.

6.5 Aberrant Workload

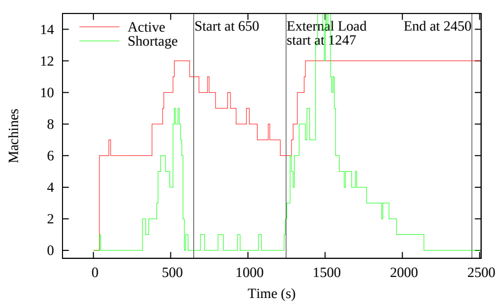
6.5.1 External Load Scenario

A different scenario, in which a cluster's available capacity can be reduced, is when an aberrant workload occupies that capacity. This can occur inadvertently: for example, a software bug can manifest itself in an infinite loop, or a user can make a mistake in a command. The external workload can also come from malicious activity, deliberately injected in order to cause denial-of-service.

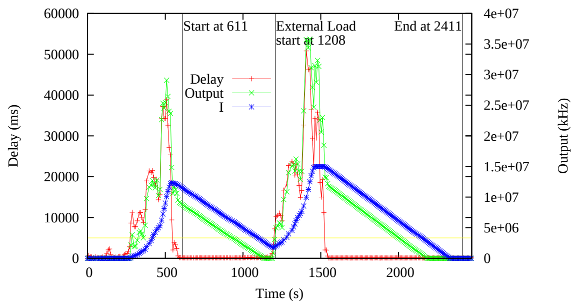
In our experiment, we create an aberrant load by starting a shell process running an infinite loop on the machines. Unlike in the thermal overload experiment, not all machines are overloaded, since that would leave no room for *any* algorithm to recover—the cluster could not meet the performance goal, no matter how many machines were activated. We can evaluate a more interesting scenario by leaving a few (three) machines unaffected. We verified in our logs that all except three machines become 100% utilized after the external load event.

6.5.2 Aberrant Workload Results

Figure 6.4(a) shows that shortly after the external load is introduced, the algorithm activates all 12 machines. The reason, as we can see in Figure 6.4(b), is the sharp increase in delay, recognized



(a) Trace of cluster capacity.



(b) Delay control performance.

Figure 6.4: Algorithm behavior before and during aberrant external load.

by the feedback controller. After activation (and optimal allocation) of the 3 unaffected machines, latency is quickly brought down. Importantly, due to very high measured utilization, all machines are kept active through the entire event despite the delay becomes low. This is reasonable since the algorithm does not distinguish whether the load is aberrant, and therefore does not know which machine's capacity can be safely reduced. Recognizing the nature of arbitrary loads would increase application dependencies (and complexity), which we strived to avoid. Additionally, this simplicity still implicitly results in the expected behavior, assuming we are conservative with respect to performance (i.e. if we prefer acceptable performance to acceptable energy savings, which is the typical case).

Finally, Figure 6.5 presents the request throughput of the system over time. Overall throughput remains remarkably close to the baseline. Further, it is clearly seen that the injected load only temporarily disrupted service. Thanks to the algorithm's behavior, original service rates are quickly restored (in only about 4 minutes) and original throughput is maintained thereafter, despite the continuous overload. It should be noted that service disruptions (response time spikes and throughput drops) seen in our experiments are easily abridged with spare idle servers. We deliberately did not leave spare servers in order to illustrate performance impacts.

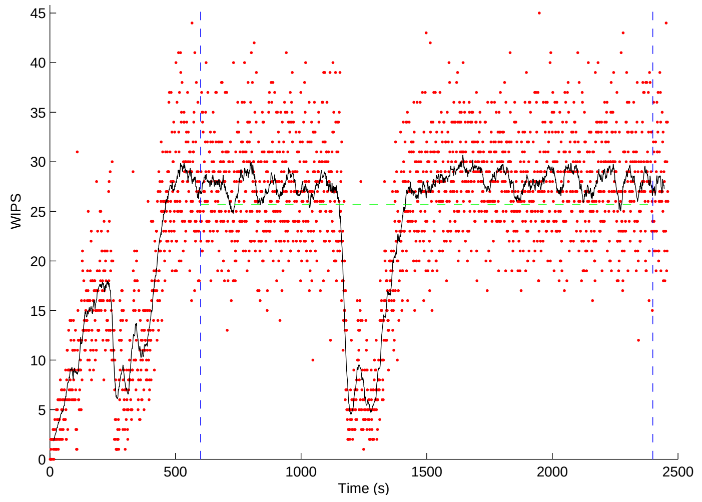


Figure 6.5: Throughput (web interactions per second) before and during aberrant external workload experiment. Dots: samples; Solid line: moving average.

6.6 Conclusions

With an increasing emphasis on DTM to overcome deficiencies of average-case cooling, the *interaction* between power management algorithms and DTM will become more pronounced in the future. The interplay is especially interesting in pipeline-parallel applications (whether in multi-tier server clusters or many-core PCs) since more complex reconfigurations are possible. The same is true for other overload conditions that limit system capacity similarly to DTM. Our work focused on exploring both.

We evaluated the resilience of our algorithm to certain unexpected events (i.e., DTM and load injection) that result in system overload. Fault tolerant behavior is desired such that the algorithm remains effective and does not further compromise the system during such events. We found that feedback control is a key element in enabling this fault tolerance. Even though the algorithm has no *direct* knowledge of these circumstances, its feedback controller allows it to perform successfully, without the need to explicitly monitor that the actual state of the system is in correspondence with the algorithm's view. These properties make these kinds of algorithms robust to aberrant capacity reduction in general, and therefore they are much more attractive in practical settings. Conversely, we also found that simple open-loop algorithms may not be able to preserve system performance in a DTM-intensive (or otherwise limited) system.

These basic insights are likely applicable to other workloads: for instance, task parallelism in multicore chips that support per-core DVS and perhaps a few possible sleep states, is a promising research direction for future work. With a multi-tier set of tasks running on a single chip, tasks can be assigned to cores and cores can individually be put in different power states or be shut down.

Chapter 7

Conclusions

7.1 Summary

We have explored several open problems within the relatively new field of server energy management and offered novel solutions to them. A unique approach was taken by addressing multi-tier servers, a more general class of systems than conventional servers, which is widely used for its greatly improved scalability. Because in practice most large-scale services are multi-tier, these workloads are very important to consider. We relied on formal optimization to build a theoretical foundation under our policies and algorithms, which was reinforced by feedback control to correct for modeling inaccuracies and to help tolerate aberrant node behavior. We started with a rather limited system model, characterized using simple theoretical power and delay functions. Gradually, this was generalized into a fully realistic reconfigurable cluster model with refined machine and workload characterization based on actual system measurements. Our final model is capable of describing large-scale multi-tier server farms with modern power management features, for which we provide practical, effective energy management solutions. Our results naturally also apply to single-tier servers as a special case.

The importance of this work is also affected by a number of factors. Since we control only server-side latency, if network latency from the clients to the data center, or client-side processing is the overwhelming portion of client-perceived latency, then our solutions have limited impact on it. Note that this would allow even more aggressive server energy management, since the relative client-perceived performance impact is smaller. Another factor is memory and disk power usage. This is typically not dominant in current systems, but if a very large number of these components is being used per server, relative energy savings from controlling CPU power would be reduced. Still, cluster reconfiguration would remain effective because system sleep states affect all of these components. Finally, as future CPUs will have many cores, different strategies may be necessary to optimize energy consumption. While the effectiveness of DVS may diminish, new techniques such as per-core sleep modes might be exploited.

7.2 Limitations and Lessons Learned

Early on in the project, we identified that since end-to-end delay is the most important client-perceived performance metric, our goal must be to maintain it below a given setpoint or deadline. We then proceeded to model it using simple queuing theory, and later using non-linear regression. Additionally, in our earliest work, we achieved soft real-time performance by empirically calculating the conditional probability of future deadline misses, solely from the current latency samples. While these approaches were somewhat successful (i.e., we obtained acceptable latency results), by now it has become clear to me that they are not the best ways to achieve latency control. The inherent problem lies in the fact that a multi-tier server cluster is much more complex than what these simple techniques can capture (e.g., resource saturation leads to errors instead of long latencies, we have limited control over many timeout mechanisms, there are sophisticated request proxying algorithms, query caching, etc.). In other words, these techniques are easily broken just by adjusting a few system parameters, such as connection pool sizes, timeouts, etc. Systems require careful tuning to get good results. The reason is that using only service delay as the performance constraint proves to be too restrictive in practice. The well-studied inherent burstiness of Web workloads results in short-term overloads even when average delay stays constant.

What is needed, instead, is a scheme that can cope with the non-ideal realities of complex server software behavior. Even the per-tier latency distributions are bursty and self-similar [56]. Condensing end-to-end performance into a tractable yet accurate function seems unrealistic. Instead of a utilization-delay characteristic function, simple load thresholding techniques that limit the number of connections or sessions on each server to a safe value, were apparently used successfully [69]. In multi-tier servers, the limit should be allocated dynamically to each tier, based on a new tier equalizing condition derived form this constraint instead of our current delay function. Although harder to argue about its energy efficiency, one should keep in mind that meeting SLAs is normally always more important than optimal energy goals. Other possibilities include using additional inputs to help better estimate demand. We took a step in this direction by adding a server error sensor into our feedback control loop, and it markedly improved performance. The drawback is that, being a feedback controller, it cannot avoid errors altogether. Other indicators for server performance control could supplement our design in order to avoid overload, such as server queue length, which has been successfully applied before. Incorporating this would only require a simple extension to our feedback controller.

An additional limitation of our system model is that it does not directly incorporate I/O performance control. This might be a problem with reconfigurable tiers running completely I/O-bound applications, in which CPU utilization is a bad indicator of delay. This limitation can be overcome by extending the performance and power models with I/O utilization measurements.

An implementation-related limitation and corresponding lesson is regarding session management. In the reconfigurable cluster setup, the assumption was made that servers selected for transitioning to sleep state do not have to drain their sessions (i.e., wait until all sessions are finished). I believe this is a reasonable assumption, since doing so could take an arbitrary and unpredictable (client-dependent) amount of time, during which that node falls out of energy management—a clearly undesirable situation. However, I did not implement a mechanism to hand off the involved session data, resulting in a server error for the next request in each of those sessions. This turned out to be unfortunate, because in the spare capacity optimization study, the error rate would be an interesting metric if these session-related errors could be separated. Even worse, after shutting down a server, the increased error rate is noticed in our own feedback controller, which will try to compensate for it. Luckily, this effect was not frequent enough to make a major difference in the results. However, it makes it hard to argue about the error figures.

A further limitation related to sessions is that with cluster reconfiguration, session distribution incurs additional overhead when adding machines, which overhead decreases energy efficiency. Therefore, this will need to be accounted for in the analysis, which is an important area for future work. Sessions or other large distributed data sets stored in a tier also place limitations on migration between tiers, since data synchronization on migration might be overly costly to be beneficial altogether. This may typically apply to database server tiers. Note that for this reason, in our prototype, we disallowed migration from the database tier.

One issue with interpreting our results is that it is not straightforward to accurately extrapolate them to real-life data centers at realistic scales. The difficulty arises because of differences in workload fluctuations (e.g., slower relative demand changes, significant long-term fluctuations) and a much smaller ratio of necessary spare machines to total machines. That said, I do not expect large-scale results to be fundamentally different, since savings from DVS are per-machine, and assuming the same workload profile, a similar portion of the machines would be unnecessary, resulting in similar relative savings from sleep states as well.

As far as our workloads and results are concerned, I would now focus more on how to emphasize the benefit of the multi-tier optimization over simple heuristic algorithms. Even some simple tweak in an otherwise static workload that causes a dynamic imbalance, similar to what we used in section 4.6.5, should give better results than what we obtained with the static load profiles, which do not highlight the benefits. By the time I was working on the reconfigurable cluster setup, I realized that we needed much more realistic workloads for the results to be convincing, which is reflected in the trace-driven TPC-W approach we first used there. However, to convincingly show why multi-tier energy management must be treated separately, an ideal workload would have, for instance, different request types that place different loads on the tiers, so that varying the request mix would produce a dynamic shifting of tier loads, highlighting the benefits of dynamic allocation.

Finally, it is worth noting that experimentation on a real testbed was extremely time-consuming. There are significant setup costs, tuning both the system under test and the experiment timings, including diagnosing performance problems, isolating and fixing their causes. Instabilities, crashes frequently occur, especially with the still immature power management technology. Even when everything works smoothly, producing an interesting number of data points usually requires weeks of pure run time. Simulation should be considered as an alternative. Nevertheless, the insight and experience I have gained in the process is invaluable, and ultimately this methodology gave the results much more credibility than simulation alone would have.

7.3 Future Work

Research we are aware of has not yet addressed joint optimization of peak power and energy. However, in real data centers, both may be desired. Thus, it would be interesting to formally analyze the joint problem to determine whether their trivial combination is optimal.

A significant portion of total system power is attributed to disks. In fact, in most typical systems disks are the second largest power consumers after CPUs. Still, traditional disk power management capabilities are very limited, only allowing the disk to be spun up and down. When the disk is spun down, no activity can take place on it, which severely limits the usefulness of this technique due to the high spin-up latency involved. Fortunately, recent research has led to the DRPM [29] technique, which allows the disk rotation speed to be dynamically changed, analogously to the DVS technique in CPUs. Given the importance of disk power, it would be worthwhile to extend our analysis with the model of such variable-speed drives, and derive the optimal combination of power states, including disk speed, that minimizes the total energy usage of the cluster.

For our reconfigurable cluster policy, we did not incorporate support for multiple client classes and request classes. Adding this would be beneficial for handling workloads where the request mix dynamically changes over time.

Additional issues arise from virtualization. When virtual machines (VMs) are present, several individual OS power management policies may attempt to control the same hardware feature (e.g. DVS) due to VM consolidation. To handle this, power management could be performed at the VM monitor (VMM) level. This requires hooks that allow the individual VMs to express their objectives, enabling global power optimization by the VMM. One such alternative is for the VMM to export virtual DVS states for each VM, and then take the individual VM's settings as hints for the global policy [63]. Also, cluster reconfiguration may be simplified by the virtualization layer if it can automatically move data and network addresses when consolidating services, thus retaining sessions associated with the server being moved. The consolidation of tiers may provide for additional optimization opportunities, because the VMM has knowledge that the separate VMs are actually cooperating to meet a common global objective.

Multicore architectures also have new issues and provide new opportunities. For instance, placement of threads on the cores becomes a significant factor in both performance and energy consumption. This is especially true because of different communication costs between cores within the same chip and those on different chips, which further affects load balancing and power management algorithms. Specifically, the two algorithms might cooperate to increase overall energy efficiency by consolidating threads such that better communication performance is achieved, and at the same time entire chips can be put into a sleep state, as opposed to only individual cores but with all chips active. In summary, for good cluster-level energy management, more sophisticated machine-level algorithms are needed in a multicore environment.

Appendix A

Glossary

ACPI Advanced Configuration and Power Interface. Establishes industry-standard interfaces enabling OS-directed power and thermal management. Defines device-, CPU-, and system-level power states.

CMP Chip Multiprocessor. A CPU with multiple processing cores.

DPM Dynamic power management. A design methodology for automatically setting the appropriate power state for components based on the workload. Decisions about power state transitions are made by a power management policy.

DTM Dynamic thermal management. A design methodology for dynamically reducing power dissipation of a component (CPU) if its temperature exceeds a certain value, in an attempt to prevent further heat buildup or damage.

DVS Dynamic voltage scaling. A power management technique involving setting discrete CPU voltage and frequency combinations during operation.

DVFS Dynamic voltage and frequency scaling. An alternate name for *DVS*.

Hibernate-To-Disk The ACPI S4 power state, whereby volatile program state (in the RAM and CPU registers) is saved to disk and most components can be shut down. Its power demand is similar or identical to the Soft-Off state but the wakeup time may be shorter because programs are already loaded and initialized in the saved RAM image.

Hibernate-To-RAM The ACPI S1 or S3 power state, in which CPU state is saved to RAM and many components (such as CPUs and disks) can be shut down. The RAM is still powered to maintain contents. Power demand can vary based on implementation and on the amount and type of RAM modules. Wakeup time is fast as it mostly involves the initialization of CPUs (and other devices).

P-state ACPI CPU performance state, which is the ACPI terminology for DVS state.

PI control Proportional-Integral control. A widely used feedback control mechanism, which determines its output based on the current error and the sum of recent errors.

Pipeline-parallel system A system where computation has been partitioned into processing stages. The stages can work in parallel, yielding substantial speedup.

S-state System-level power state in ACPI, such as Hibernate-To-RAM, Hibernate-To-Disk, and Soft-Off.

SLA Service Level Agreement. A specification of the performance requirements for a hosted Internet service.

Soft-Off The ACPI S5 power state, in which all except a few circuits in a computer are shut down. The active circuits allow non-mechanical wakeup, for example by pressing a keyboard button or by the network adapter through Wake-On-LAN. In contrast to the mechanical off state where all circuits are completely disconnected from power, the Soft-Off state requires a small amount of power.

Wake-On-LAN A protocol for waking up a machine remotely, over the network. The network interface listens for an appropriate wakeup packet even in `Soft-Off` state.

Ziegler-Nichols tuning method A manual P, PI, or PID controller tuning procedure that determines stable controller gains based on measurements on the system.

Bibliography

[1] Tarek F. Abdelzaher, Kang G. Shin, and Nina Bhatti. Performance guarantees for Web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 13(1):80–96, 2002.

[2] Nevine AbouGhazaleh, Robert N. Mayo, and Parthasarathy Ranganathan. Idle time power management for personal wireless devices. Technical Report HPL-2003-102, HP Laboratories, Palo Alto, CA, USA, September 2003.

[3] Jussara Almeida, Mihaela Dabu, Anand Manikutty, and Pei Cao. Providing differentiated levels of service in web content hosting. Technical Report CS-TR-1998-1364, 1998.

[4] Martin Arlitt and Tai Jin. 1998 world cup web site access logs, August 1998. URL `http://www.acm.org/sigcomm/ITA/`.

[5] Hakan Aydi, Pedro Mejia-Alvarez, Daniel Mosse, and Rami Melhem. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS'01)*, page 95, 2001. ISBN 0-7695-1420-0.

[6] Luiz André Barroso. The price of performance. *Queue*, 3(7):48–53, 2005. ISSN 1542-7730. doi: 10.1145/1095408.1095420.

[7] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40(12):33–37, 2007. ISSN 0018-9162. doi: 10.1109/MC.2007.443.

[8] Luca Benini, Alessandro Bogliolo, and Giovanni De Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(3):299–316, 2000. ISSN 1063-8210. doi: 10.1109/92.845896.

[9] Nina Bhatti, Anna Bouch, and Allan Kuchinsky. Integrating user-perceived quality into web server design. In *Proceedings of the 9th International World Wide Web Conference on Computer Networks*, pages 1–16, Amsterdam, The Netherlands, May 2000. North-Holland Publishing Co. doi: 10.1016/S1389-1286(00)00087-6.

[10] Ricardo Bianchini and Ram Rajamony. Power and energy management for server systems. *IEEE Computer*, 37(11):68–74, November 2004. ISSN 0018-9162. doi: 10.1109/MC.2004.217. Special issue on Internet data centers.

[11] Pat Bohrer, Elmootazbellah N. Elnozahy, Tom Keller, Michael Kistler, Charles Lefurgy, and Ram Rajamony. The case for power management in web servers. In Robert Graybill and Rami Melhem, editors, *Power-Aware Computing*, Kluwer/Plenum series in Computer Science. Kluwer Academic Publishers, January 2002.

[12] David Brooks, Margaret Martonosi, John-David Wellman, and Pradip Bose. Power-performance modeling and tradeoff analysis for a high end microprocessor. In *PACS '00: Proceedings of the First International Workshop on Power-Aware Computer Systems-Revised Papers*, pages 126–136, London, UK, 2001. Springer-Verlag. ISBN 3-540-42329-X.

[13] Le Cai, Nathaniel Pettis, and Yung-Hsiang Lu. Joint power management of memory and disk under performance constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(12):2697–2711, December 2006. ISSN 0278-0070. doi: 10.1109/TCAD.2006.882587.

[14] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. Managing energy and server resources in hosting centers. In *Symposium on Operating Systems Principles*, pages 103–116, 2001.

[15] Yiyu Chen, Amitayu Das, Wubi Qin, Anand Sivasubramaniam, Qian Wang, and Natarajan Gautam. Managing server energy and operational costs in hosting centers. *SIGMETRICS Performance Evaluation Review*, 33(1):303–314, 2005. ISSN 0163-5999. doi: 10.1145/1071690.1064253.

[16] Eui-Young Chung, Luca Benini, and Giovanni De Micheli. Dynamic power management using adaptive learning tree. In *ICCAD '99: Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*, pages 274–279, Piscataway, NJ, USA, 1999. IEEE Press. ISBN 0-7803-5832-5.

[17] Eui-Young Chung, Luca Benini, Alessandro Bogliolo, Yung-Hsiang Lu, and Giovanni De Micheli. Dynamic power management for nonstationary service requests. *IEEE Transactions on Computers*, 51(11):1345–1361, 2002. ISSN 0018-9340. doi: 10.1109/TC.2002.1047758.

[18] Lars Eggert and John S. Heidemann. Application-level differentiated services for web servers. *World Wide Web*, 2(3):133–142, 1999.

[19] Elmootazbellah Elnozahy, Michael Kistler, and Ram Rajamony. Energy-efficient server clusters. In *Proceedings of the Workshop on Power-Aware Computing Systems*, February 2002.

[20] Elmootazbellah Elnozahy, Michael Kistler, and Ram Rajamony. Energy conservation policies for web servers. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, March 2003.

[21] Hewlett-Packard et al. Advanced configuration and power interface specification, 2006. URL `http://www.acpi.info/spec.htm`.

[22] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz André Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Computer Architecture*, pages 13–23, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-706-3. doi: 10.1145/1250662.1250665.

[23] Martin J. Fischer, Denise M. Masi, Donald Gross, and John F. Shortle. One-parameter Pareto, two-parameter Pareto, three-parameter Pareto: is there a modeling difference? In *The Telecommunications Review*, pages 79–92. Mitretek Systems, 2005.

[24] Krisztián Flautner, Steve Reinhardt, and Trevor Mudge. Automatic performance setting for dynamic voltage scaling. *Wireless Networks*, 8(5):507–520, 2002. ISSN 1022-0038. doi: 10.1023/A:1016546330128.

[25] Vincent W. Freeh and David K. Lowenthal. Using multiple energy gears in mpi programs on a power-scalable cluster. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 164–173, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-080-9. doi: 10.1145/1065944.1065967.

[26] Ricardo Gonzalez and Mark Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–1284, September 1996. ISSN 0018-9200. doi: 10.1109/4.535411.

[27] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 151–162, New York, NY, USA, 2006. ACM. ISBN 1-59593-451-0. doi: 10.1145/1168857.1168877.

[28] Kinshuk Govil, Edwin Chan, and Hal 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, Berkeley, California, United States, 1995. ACM Press. ISBN 0-89791-814-2. doi: 10.1145/215530.215546.

[29] Sudhanva Gurumurthi, Anand Sivasubramaniam, Mahmut Kandemir, and Hubertus Franke. Reducing disk power consumption in servers with DRPM. *Computer*, 36(12):59–66, 2003. ISSN 0018-9162. doi: 10.1109/MC.2003.1250884.

[30] C. C. Hang, Karl J. Astrom, and Weng Khuen Ho. Refinements of the Ziegler-Nichols tuning formula. *IEE Proceedings D, Control Theory and Applications*, 138(2):111–118, March 1991. ISSN 0143-7054.

[31] Taliver Heath, Bruno Diniz, Enrique V. Carrera, Wagner Meira Jr., and Ricardo Bianchini. Self-configuring heterogeneous server clusters. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power*, September 2003.

[32] Taliver Heath, Bruno Diniz, Enrique V. Carrera, Wagner Meira Jr., and Ricardo Bianchini. Energy conservation in heterogeneous server clusters. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 186–195, New York, NY, USA, 2005. ACM. ISBN 1-59593-080-9. doi: 10.1145/1065944.1065969.

[33] Taliver Heath, Ana Paula Centeno, Pradeep George, Luiz Ramos, Yogesh Jaluria, and Ricardo Bianchini. Mercury and freon: temperature emulation and management for server systems. *SIGPLAN Not.*, 41(11):106–116, 2006. ISSN 0362-1340. doi: 10.1145/1168918.1168872.

[34] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 2003. ISBN 1558607242.

[35] Tibor Horvath, Tarek Abdelzaher, Kevin Skadron, and Xue Liu. Dynamic voltage scaling in multi-tier web servers with end-to-end delay control. *IEEE Transactions on Computers*, 56 (4):444–458, 2007. ISSN 0018-9340. doi: 10.1109/TC.2007.1003.

[36] Tibor Horvath, Kevin Skadron, and Tarek Abdelzaher. Enhancing energy efficiency in multi-tier web server clusters via prioritization. *Proceedings of the 2007 NSF Next Generation Software Workshop, in conjunction with the IEEE International Parallel and Distributed Processing Symposium*, pages 1–6, March 2007. doi: 10.1109/IPDPS.2007.370509.

[37] Chung hsing Hsu and Wu chun Feng. When discreteness meets continuity: Energy-optimal dvs scheduling revisited. Technical Report LA-UR 05-3104, Los Alamos National Laboratory, Los Alamos, NM, USA, February 2005.

[38] Sandy Irani and Kirk R. Pruhs. Algorithmic problems in power management. *SIGACT News*, 36(2):63–76, 2005. ISSN 0163-5700. doi: 10.1145/1067309.1067324.

[39] ITA. The internet traffic archive, 2000. URL `http://ita.ee.lbl.gov/`.

[40] ITRS. International technology roadmap for semiconductors, 2006. URL `http://www.itrs.net/`.

[41] Dong-In Kang, Stephen Crago, and Jinwoo Suh. Power-aware design synthesis techniques for distributed real-time systems. In *Proceedings of the ACM Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'01)*, pages 20–28, New York, June 2001.

[42] Anna R. Karlin, Mark S. Manasse, Lyle A. McGeoch, and Susan Owicki. Competitive randomized algorithms for non-uniform problems. *Algorithmica*, 11(6):542–571, June 1994. ISSN 0178-4617. doi: 10.1007/BF01189993.

[43] Jonathan G. Koomey. *Estimating total power consumption by servers in the U.S. and the world*. Analytics Press, Oakland, CA, February 2007.

[44] Amit Kumar, Li Shang, Li-Shiuan Peh, and Niraj K. Jha. Hybdtm: a coordinated hardware-software approach for dynamic thermal management. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 548–553, New York, NY, USA, 2006. ACM. ISBN 1-59593-381-6. doi: 10.1145/1146909.1147052.

[45] Alvin R. Lebeck, Xiaobo Fan, Heng Zeng, and Carla Ellis. Power aware page allocation. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 105–116, New York, NY, USA, 2000. ACM. ISBN 1-58113-317-0. doi: 10.1145/378993.379007.

[46] Charles Lefurgy, Karthick Rajamani, Freeman Rawson, Wes Felter, Michael Kistler, and Tom W. Keller. Energy management for commercial servers. *IEEE Computer*, 36(12):39–48, 2003. ISSN 0018-9162. doi: 10.1109/MC.2003.1250880.

[47] Charles Lefurgy, Xiaorui Wang, and Malcolm Ware. Power capping: a prelude to power shifting. *Cluster Computing*, November 2007. ISSN 1386-7857.

[48] Charles Lefurgy, Xiaorui Wang, and Malcolm Ware. Server-level power control. In *ICAC '07: Proceedings of the Fourth International Conference on Autonomic Computing*, page 4, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2779-5. doi: 10.1109/ICAC.2007.35.

[49] Yingmin Li, Benjamin Lee, David Brooks, Zhigang Hu, and Kevin Skadron. Cmp design space exploration subject to physical constraints. *The Twelfth International Symposium on High-Performance Computer Architecture*, pages 17–28, February 2006. ISSN 1530-0897. doi: 10.1109/HPCA.2006.1598109.

[50] Tristan Louis. How many Google machines, 2004. URL `http://www.tnl.net/blog/entry/How_many_Google_machines`.

[51] Zhijian Lu, Jason Hein, Marty Humphrey, Mircea Stan, John Lach, and Kevin Skadron. Control-theoretic dynamic frequency and voltage scaling for multimedia workloads. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 156–163, Grenoble, France, October 2002. ACM Press. ISBN 1-58113-575-0. doi: 10.1145/581630.581654.

[52] Zhijian Lu, John Lach, Mircea R. Stan, and Kevin Skadron. Reducing multimedia decode power using feedback control. In *Proceedings of the 21st International Conference on Computer Design*, pages 489–496, October 2003.

[53] Lykomidis Mastroleon, Nicholas Bambos, Christos Kozyrakis, and Dimitris Economou. Autonomic power management schemes for internet servers and data centers. *Global Telecommunications Conference*, 2, 2005.

[54] David T. McWherter, Bianca Schroeder, Anastassia Ailamaki, and Mor Harchol-Balter. Priority mechanisms for OLTP and transactional web applications. In *Proceedings of the 20th*

International Conference on Data Engineering, page 535. IEEE Computer Society, 2004. ISBN 0-7695-2065-0.

[55] Pedro Mejia-Alvarez, Eugene Levner, and Daniel Mosse. Power-optimized scheduling server for real-time tasks. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)*, page 239, 2002. ISBN 0-7695-1739-0.

[56] Ningfang Mi, Qi Zhang, Alma Riska, Evgenia Smirni, and Erik Riedel. Performance impacts of autocorrelated flows in multi-tiered systems. *SIGMETRICS Performance Evaluation Review*, 64(9-12):1082–1101, 2007. ISSN 0166-5316. doi: 10.1016/j.peva.2007.06.016.

[57] Akihiko Miyoshi, Charles Lefurgy, Eric Van Hensbergen, Ram Rajamony, and Raj Rajkumar. Critical power slope: understanding the runtime effects of frequency scaling. In *ICS '02: Proceedings of the 16th international conference on Supercomputing*, pages 35–44, New York, NY, USA, 2002. ACM. ISBN 1-58113-483-5. doi: 10.1145/514191.514200.

[58] Justin Moore, Jeff Chase, and Parthasarathy Ranganathan. Weatherman: Automated, online and predictive thermal mapping and management for data centers. *Autonomic Computing, 2006. ICAC '06. IEEE International Conference on*, pages 155–164, 13-16 June 2006.

[59] Justin Moore, Jeff Chase, Parthasarathy Ranganathan, and Ratnesh Sharma. Making scheduling "cool": temperature-aware workload placement in data centers. In *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference*, pages 61–74, Berkeley, CA, USA, 2005. USENIX Association.

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

[61] Trevor Mudge. Power: A first-class architectural design constraint. *Computer*, 34(4):52–58, 2001. ISSN 0018-9162. doi: 10.1109/2.917539.

[62] MySQL AB. MySQL database server. URL `http://www.mysql.com/`.

[63] Ripal Nathuji and Karsten Schwan. Virtualpower: coordinated power management in virtualized enterprise systems. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 265–278, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5. doi: 10.1145/1294261.1294287.

[64] ObjectWeb Consortium. TPC-W benchmark. URL `http://jmob.objectweb.org/tpcw.html`. Based on the implementation from University of Wisconsin - Madison.

[65] Pradeep Padala, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, and Kenneth Salem. Adaptive control of virtualized resources in utility computing environments. *SIGOPS Operating Systems Review*, 41(3):289–302, 2007. ISSN 0163-5980. doi: 10.1145/1272998.1273026.

[66] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 89–102, Banff, Alberta, Canada, 2001. ACM Press. ISBN 1-58113-389-8. doi: 10.1145/502034.502044.

[67] Eduardo Pinheiro, Ricardo Bianchini, Enrique V. Carrera, and Taliver Heath. Load balancing and unbalancing for power and performance in cluster-based systems. Technical Report DCS-TR-440, Department of Computer Science, Rutgers University, May 2001.

[68] Eduardo Pinheiro, Ricardo Bianchini, Enrique V. Carrera, and Taliver Heath. Dynamic cluster reconfiguration for power and performance. In L. Benini, M. Kandemir, and J. Ramanujam, editors, *Compilers and Operating Systems for Low Power*. Kluwer Academic Publishers, 2002.

[69] Karthick Rajamani and Charles Lefurgy. On evaluating request-distribution schemes for saving energy in server clusters. In *ISPASS '03: Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 111–122, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7803-7756-7.

[70] Parthasarathy Ranganathan, Phil Leech, David Irwin, and Jeffrey Chase. Ensemble-level power management for dense blade servers. *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Computer Architecture*, pages 66–77, 2006. ISSN 1063-6897. doi: 10.1109/ISCA.2006.20.

[71] Red Hat, Inc. JBoss application server. URL `http://www.jboss.org/`.

[72] Cosmin Rusu, Alexandre Ferreira, Claudio Scordino, and Aaron Watson. Energy-efficient real-time heterogeneous server clusters. In *RTAS '06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 418–428, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2516-4. doi: 10.1109/RTAS.2006.16.

[73] Lui Sha, Xue Liu, Ying Lu, and Tarek Abdelzaher. Queuing model based network server performance control. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, Austin, TX, December 2002.

[74] Ratnesh K. Sharma, Cullen E. Bash, Chandrakant D. Patel, Richard J. Friedrich, and Jeffrey S. Chase. Balance of power: Dynamic thermal management for internet data centers. *IEEE Internet Computing*, 9(1):42–49, 2005. ISSN 1089-7801. doi: 10.1109/MIC.2005.10.

[75] Vivek Sharma, Arun Thomas, Tarek Abdelzaher, Kevin Skadron, and Zhijian Lu. Power-aware QoS management in web servers. In *Proceedings of the IEEE International Real-Time Systems Symposium*, pages 63–72, Cancun, Mexico, December 2003. ISBN 0-7695-2044-8.

[76] Tajana Simunic, Luca Benini, Andrea Acquaviva, Peter Glynn, and Giovanni De Micheli. Dynamic voltage scaling and power management for portable systems. In *Proceedings of the 38th Conference on Design Automation*, pages 524–529, Las Vegas, Nevada, United States, 2001. ACM Press. ISBN 1-58113-297-2. doi: 10.1145/378239.379016.

[77] Preeti Bhoj Srinivas. Web2k: Bringing qos to web servers, 2000.

[78] The Apache Software Foundation. The Apache HTTP server. URL `http://www.apache.org/`.

[79] William Thies, Vikram Chandrasekhar, and Saman Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 356–369, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-3047-8. doi: 10.1109/MICRO.2007.7.

[80] Matthew E. Tolentino, Joseph Turner, and Kirk W. Cameron. Memory-miser: a performance-constrained runtime system for power-scalable clusters. In *CF '07: Proceedings of the 4th international conference on Computing frontiers*, pages 237–246, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-683-7. doi: 10.1145/1242531.1242566.

[81] Alexander A. Totok. J2EE-based implementation of the TPC-W benchmark, February 2004. URL `http://www.cs.nyu.edu/~totok/professional/software/tpcw/tpcw.html`. New York University.

[82] Transaction Processing Performance Council (TPC). TPC benchmark W (web commerce) specification version 1.8, February 2002. URL `http://www.tpc.org/tpcw/spec/tpcw_V1.8.pdf`.

[83] Osman S. Unsal and Israel Koren. System-level power-aware design techniques in real-time systems. *Proceedings of the IEEE*, 91(7):1055–1069, July 2003. Special Issue on Real-Time Systems.

[84] Ankush Varma, Brinda Ganesh, Mainak Sen, Suchismita Roy Choudhury, Lakshmi Srinivasan, and Jacob Bruce. A control-theoretic approach to dynamic voltage scheduling. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 255–266, San Jose, California, USA, 2003. ACM Press. ISBN 1-58113-676-5. doi: 10.1145/951710.951744.

[85] Mark Weiser, Brent Welch, Alan J. Demers, and Scott Shenker. Scheduling for reduced CPU energy. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'94)*, pages 13–23, Monterey, CA, November 1994.

[86] Andreas Weissel and Frank Bellosa. Dynamic thermal management for distributed systems. In *Proceedings of the First Workshop on Temperature-Aware Computer Systems (TACS'04)*, Munich, Germany, June 2004.

[87] Adepele Williams, Martin Arlitt, Carey Williamson, and Ken Barker. *Web Workload Characterization: Ten Years Later*. Springer, August 2005. ISBN 0-387-24356-9.

[88] Wanghong Yuan and Klara Nahrstedt. Energy-efficient soft real-time CPU scheduling for mobile multimedia systems. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 149–163, Bolton Landing, NY, USA, 2003. ACM Press. ISBN 1-58113-757-5. doi: 10.1145/945445.945460.

[89] Qi Zhang, Ludmila Cherkasova, and Evgenia Smirni. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *ICAC '07: Proceedings of the Fourth International Conference on Autonomic Computing*, page 27, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2779-5. doi: 10.1109/ICAC.2007.1.

[90] Wensong Zhang. Linux virtual server for scalable network services, July 2000. URL `http://www.LinuxVirtualServer.org/`.

[91] Dakai Zhu, Rami Melhem, and Bruce Childers. Scheduling with dynamic voltage/speed adjustment using slack reclamation in multi-processor real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS'01)*, page 84, 2001. ISBN 0-7695-1420-0.

[92] Yifan Zhu and Frank Mueller. Feedback EDF scheduling exploiting dynamic voltage scaling. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 84–93, May 2004.

[93] John G. Ziegler and Nathaniel B. Nichols. Optimum settings for automatic controllers. *Transactions of ASME*, 64:759–768, November 1942.