Managing Energy and Server Resources in Hosting Centers

Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat *
Department of Computer Science
Duke University
{chase, anderson, prachi, vahdat}@cs.duke.edu

Ronald P. Doyle†
Application Integration and Middleware
IBM Research Triangle Park
rdoyle@us.ibm.com

ABSTRACT

Internet hosting centers serve multiple service sites from a common hardware base. This paper presents the design and implementation of an architecture for resource management in a hosting center operating system, with an emphasis on *energy* as a driving resource management issue for large server clusters. The goals are to provision server resources for co-hosted services in a way that automatically adapts to offered load, improve the energy efficiency of server clusters by dynamically resizing the active server set, and respond to power supply disruptions or thermal events by degrading service in accordance with negotiated Service Level Agreements (SLAs).

Our system is based on an economic approach to managing shared server resources, in which services "bid" for resources as a function of delivered performance. The system continuously monitors load and plans resource allotments by estimating the value of their effects on service performance. A greedy resource allocation algorithm adjusts resource prices to balance supply and demand, allocating resources to their most efficient use. A reconfigurable server switching infrastructure directs request traffic to the servers assigned to each service. Experimental results from a prototype confirm that the system adapts to offered load and resource availability, and can reduce server energy usage by 29% or more for a typical Web workload.

1. INTRODUCTION

The Internet buildout is driving a shift toward server-based computing. Internet-based services host content and applications in data centers for networked access from diverse client devices. Service providers are adding new data center capacity for Web hosting, application services, outsourced storage, electronic markets, and other network services. Many of these services are co-hosted in shared data centers managed by third-party hosting providers. Managed hosting in shared centers offers economies of scale and a potential for dynamic capacity provisioning to respond to request traffic, quality-of-service specifications, and network conditions. The services lease resources from the hosting provider on a "pay as you go" basis; the provider multiplexes the shared resources (e.g., servers, storage, and network bandwidth) to insulate its customers from demand surges and capital costs for excess capacity.

Hosting centers face familiar operating system challenges common to any shared computing resource. The center's operating system must provide a uniform and secure execution environment, isolate services from unintended consequences of resource sharing, share resources fairly in a way that reflects priority, and degrade gracefully in the event of failures or unexpected demand surges. Several research projects address these challenges for networks of servers; Section 7 surveys related research.

This paper investigates the *policies* for allocating resources in a hosting center, with a principal focus on energy management. We present the design and implementation of a flexible resource management architecture—called Muse—that controls server allocation and routing of requests to selected servers through a reconfigurable switching infrastructure. Muse is based on an economic model in which customers "bid" for resources as a function of service volume and quality. We show that this approach enables *adaptive resource provisioning* in accordance with flexible Service Level Agreements (SLAs) specifying dynamic tradeoffs of service quality and cost. In addition, Muse promotes energy efficiency of Internet server clusters by balancing the cost of resources (e.g., energy) against the benefit realized by employing them. We show how this *energy-conscious provisioning* allows the center to automatically adjust on-power capacity to scale with load, yielding a significant energy savings for typical Internet service workloads.

This paper is organized as follows. Section 2 outlines the motivation for adaptive resource provisioning and the importance of energy in resource management for large server clusters. Section 3 gives an overview of the Muse architecture. Section 4 presents the economic resource allocation scheme in detail, including the bidding model, load estimation techniques, and a greedy resource allocation algorithm that we call Maximize Service Revenue and Profit (MSRP) to set resource prices and match resource supply and demand. Section 5 outlines the Muse prototype, and Section 6 presents experimental results. Section 7 sets our approach in context with related work.

*This work is supported by the U.S. National Science Foundation through grants CCR-00-82912 and EIA-9972879. Anderson is supported by a U.S. Department of Education GAANN fellowship. Vahdat is supported by NSF CAREER award CCR-9984328.

†R. Doyle is also a PhD student in the Computer Science department at Duke University.

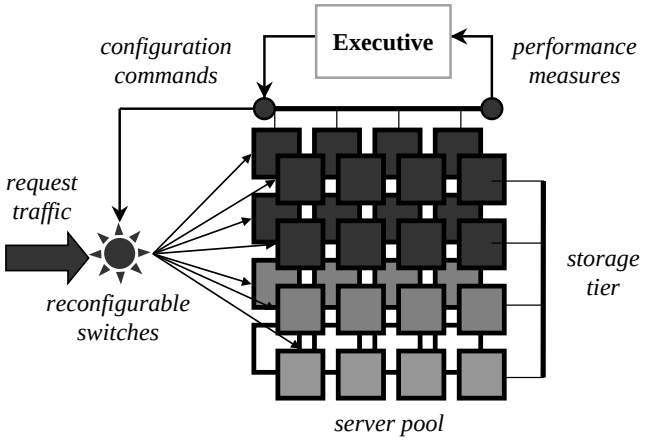

Figure 1: Adaptive resource provisioning in Muse.

2. MOTIVATION

Hosting utilities provision server resources for their customers—co-hosted server applications or *services*—according to their resource demands at their expected loads. Since outsourced hosting is a competitive business, hosting centers must manage their resources efficiently. Rather than overprovisioning for the worst-case load, efficient admission control and capacity planning policies may be designed to limit rather than eliminate the risk of failing to meet demand [8, 2]. An efficient resource management scheme would automatically allocate to each service the minimal server resources needed for acceptable service quality, leaving surplus resources free to deploy elsewhere. Provisioning choices must adapt to changes in load as they occur, and respond gracefully to unanticipated demand surges or resource failures. For these reasons managing server resources automatically is a difficult challenge.

This paper describes a system for adaptive resource management that incorporates power and energy as primary resources of a hosting center. A data center is effectively a large distributed computer comprising an "ecosystem" of heterogeneous components including edge servers, application servers, databases, and storage, assembled in a building with an infrastructure to distribute power, communications, and cooling. Viewed from the outside, the center is a "black box" with common external connections to the Internet and the electrical grid; it generates responses to a stream of requests from the network, while consuming power and producing waste heat. Energy management is critical in this context for several inter-related reasons:

- Data centers are vulnerable to overloading of the thermal system due to cooling failures, external conditions, or high service load. Muse can respond to these threats by automatically scaling back power demand (and therefore waste heat), rather than shutting down or risking damage to expensive components. Similarly, power supply disruptions may force the service to reduce power draw, e.g., to run longer on limited backup power. We refer to this as *browndown*—a new partial failure mode specific to "data center computers". The effect of browndown is to create a resource shortage, forcing the center to degrade service quality.
- Power supply (including backup power) and thermal systems are a significant share of capital outlay for a hosting center. Capacity planning for the worst case increases this cost. Muse enables the center to size for expected loads and scale back power (browndown) when exceptional conditions exceed energy or thermal limits. This is analogous to *dynamic thermal management* for individual servers [13].
- The Internet service infrastructure is a major energy consumer, and its energy demands are growing rapidly. One projection is that US data centers will consume 22 TWh of electricity in 2003 for servers, storage, switches, power conditioning, and cooling [29]. This energy would cost $2B annually at a common price of $100 per MWh; price peaks of $500 per MWh have been common on the California spot market. Energy will make up a growing share of operating costs as administration for these centers is increasingly automated [5, 12]. Moreover, generating this electricity would release about 12M tons of new CO_2 annually. Some areas are zoning against data centers to protect their local power systems [29]. Improving energy efficiency for data centers will yield important social and environmental benefits, in addition to reducing costs.
- Fine-grained balancing of service quality and resource usage—including power—gives businesses control over quality and price tradeoffs in negotiating SLAs. For example, energy suppliers offer cheaper power to customers who can reduce consumption on demand. We show how SLAs for a hosting utility may directly specify similar tradeoffs of price and quality. For example, customers might pay a lower hosting rate to allow for degraded service during power disruptions.

This paper responds to these needs with a resource management architecture that adaptively provisions resources in a hosting center to (1) avoid inefficient use of energy and server resources, (2) provide a capability to adaptively scale back power demand, and (3) respond to unexpected resource constraints in a way that minimizes the impact of service disruption.

3. OVERVIEW OF MUSE

Muse is an operating system for a hosting center. The components of the hosting center are highly specialized, governed by their own internal operating systems and interacting at high levels of abstraction. The role of the center's operating system is to establish and coordinate these interactions, supplementing the operating systems of the individual components.

Figure 1 depicts the four elements of the Muse architecture:

- *Generic server appliances.* Pools of shared servers act together to serve the request load of each co-hosted service. Server resources are generic and interchangeable.
- *Reconfigurable network switching fabric.* Switches dynamically redirect incoming request traffic to eligible servers. Each hosted service appears to external clients as a single virtual server, whose power grows and shrinks with request load and available resources.
- *Load monitoring and estimation modules.* Server operating systems and switches continuously monitor load; the system combines periodic load summaries to detect load shifts and to estimate aggregate service performance.

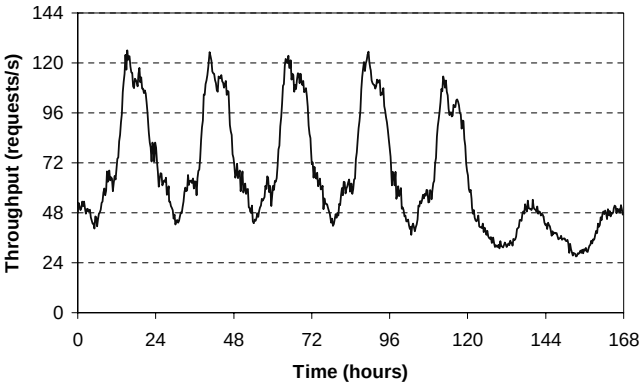

Figure 2: Request rate for the *www.ibm.com* site over February 5-11, 2001.

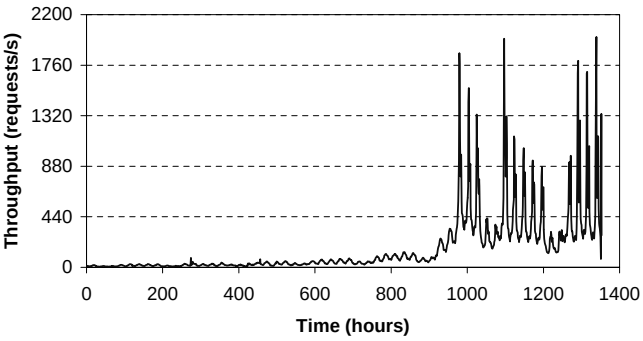

Figure 3: Request rate for the World Cup site for May-June 1998.

- *The executive.* The "brain" of the hosting center OS is a policy service that dynamically reallocates server resources and reconfigures the network to respond to variations in observed load, resource availability, and service value.

This section gives an overview of each of these components, and outlines how the combination can improve the energy efficiency of server clusters as a natural outgrowth of adaptive resource provisioning. Various aspects of Muse are related to many other systems; we leave a full treatment of related work to Section 7.

3.1 Services and Servers

Each co-hosted service consists of a body of data and software, such as a Web application. As in other cluster-based services, a given request could be handled by any of several servers running the software, improving scalability and fault-tolerance [7, 36]. Servers may be grouped into pools with different software configurations, but they may be reconfigured and redeployed from one pool to another. To enable this reconfiguration, servers are stateless, e.g., they are backed by shared network storage. The switches balance request traffic across the servers so it is acceptable for servers to have different processing speeds.

Muse allocates to each service a suitable share of the server resources that it needs to serve its load, relying on support for resource principals such as *resource containers* [10, 9] in the server operating systems to ensure performance isolation on shared servers.

3.2 Redirecting Switches

Large Web sites utilize commercial redirecting *server switches* to spread request traffic across servers using a variety of policies. Our system uses reconfigurable server switches as a mechanism to support the resource assignments planned by the executive. Muse switches maintain an *active set* of servers selected to serve requests for each network-addressable service. The switches are dynamically reconfigurable to change the active set for each service. Since servers may be shared, the active sets of different services may overlap. The switches may use load status to balance incoming request traffic across the servers in the active set.

3.3 Adaptive Resource Provisioning

Servers and switches in a Muse hosting center monitor load conditions and report load measures to the executive. The executive gathers load information, periodically determines new resource assignments, and issues commands to the servers and switches to adjust resource allotments and active sets for each service. The executive employs an economic framework to manage resource allocation and provisioning in a way that maximizes resource efficiency and minimizes unproductive costs. This defines a simple, flexible, and powerful means to quantify the value tradeoffs embodied in SLAs. Section 4 describes the resource economy in detail.

One benefit of the economic framework is that it defines a metric for the center to determine when it is or is not worthwhile to deploy resources to improve service quality. This enables a center to adjust resource allocations in a way that responds to load shifts across multiple services. Typical Internet service loads vary by factors of three or more through the day and through the week. For example, Figure 2 shows request rates for the *www.ibm.com* site over a typical week starting on a Monday and ending on a Sunday. The trace shows a consistent pattern of load shifts by day, with a weekday 4PM EST peak of roughly 260% of daily minimum load at 6AM EST, and a traffic drop over the weekend. A qualitatively similar pattern appears in other Internet service traces, with daily peak-to-trough ratios as high as 11:1 and significant seasonal variations [20]. To illustrate a more extreme seasonal load fluctuation, Figure 3 depicts accesses to the World Cup site [6] through May and June, 1998. In May the peak request rate is 30 requests per second, but it surges to nearly 2000 requests per second in June. Adaptive resource provisioning can respond to these load variations to multiplex shared server resources.

3.4 Energy-Conscious Provisioning

This ability to dynamically adjust server allocations enables the system to improve energy-efficiency by matching a cluster's energy consumption to the aggregate request load and resource demand. *Energy-conscious provisioning* configures switches to concentrate request load on a minimal active set of servers for the current aggregate load level. Active servers always run near a configured utilization threshold, while the excess servers transition to low-power idle states to reduce the energy cost of maintaining surplus capacity during periods of light load. Energy savings from the off-power servers is compounded in the cooling system, which consumes power to remove the energy dissipated in the servers as waste heat. Thus energy-conscious provisioning can also reduce fixed capacity costs for cooling, since the cooling for short periods of peak load may extend over longer intervals.

A key observation behind this policy is that today's servers are less

Architecture	Machine	Disks	Operating System	Level of Usage			
				Boot	Max	Idle	Hibernate
PIII 866MHz	SuperMicro 370-DER	1	FreeBSD 4.0	136	120	93	—
PIII 866MHz	SuperMicro 370-DER	1	Windows 2000	134	120	98	5.5
PIII 450MHz	ASUS P2BLS	1	FreeBSD 4.0	66	55	35	4
PIII Xeon 733MHz	PowerEdge 4400	8	FreeBSD 4.0	278	270	225	—
PIII 500MHz	PowerEdge 2400	3	FreeBSD 4.0	130	128	95	2.5
PIII 500MHz	PowerEdge 2400	3	Windows 2000	127	120	98	2.5
PIII 500MHz	PowerEdge 2400	3	Solaris 2.7	129	124	127	2.5

Table 1: Power draw in watts for various server platforms and operating systems.

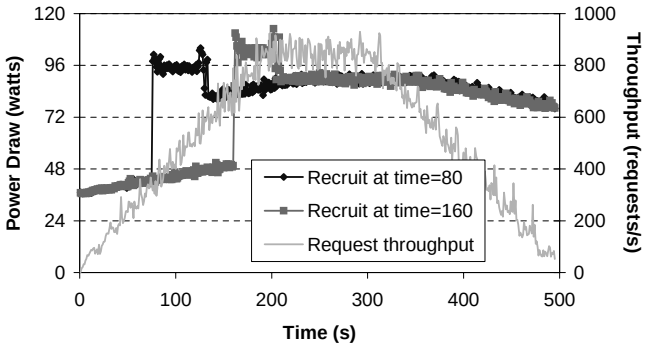

Figure 4: A comparison of power draw for two experiments serving a synthetic load swell on two servers.

energy-efficient at low CPU utilization due to fixed energy costs for the power supply. To illustrate, Figure 4 shows request throughput for two separate experiments serving a load swell on two servers through a redirecting switch, and server power draw as measured by a Brand Electronics 21-1850/CI digital power meter. The experimental configuration and synthetic Web workload are similar to other experiments reported in Section 6. Only one of the servers is powered and active at the start of each experiment. In the first experiment, the second server powers on at $t = 80$, creating a power spike as it boots, and joins the active set at $t = 130$. In the second experiment, the second server powers on later at time $t = 160$ and joins the active set at $t = 210$. Note that from $t = 130$ to $t = 160$ the second configuration serves the same request load as the first, but the first configuration draws over 80 watts while the second draws about 50, an energy savings of 37%. Section 4.1 discusses the effects on latency and throughput.

The effect is qualitatively similar on other server/OS combinations. Table 1 gives the power draw for a selection of systems, all of which pull over 60% of their peak power when idle, even when the OS halts the CPU between interrupts in the idle loop. This is because the power supply transformers impose a fixed energy cost—even if the system is idle—to maintain charged capacity to respond rapidly to demand when it occurs. When these systems are active their power draw is roughly linear with system load, ranging from the base idle draw rate up to some peak. In today's servers the CPU is the next most significant power consumer (e.g., up to 38 watts for a 600 MHz Intel Pentium-III); power draw from memory and network devices is negligible in comparison, and disks consume from 50-250 watts per terabyte of capacity (this number is dropping as disk densities increase) [12].

This experiment shows that the simplest and most effective way to reduce energy consumption in large server clusters is to turn servers off, concentrating load on a subset of the servers. In large clusters this is more effective than adaptively controlling power draw in the server CPUs, which consume only a portion of server energy. Muse defines both a mechanism to achieve this—the reconfigurable switching architecture—and policies for adaptively varying the active server sets and the number of on-power servers. Recent industry initiatives for advanced power management allow the executive to initiate power transitions remotely (see Section 5.2). An increasing number of servers on the market support these features.

One potential concern with this approach is that power transitions (e.g., disk spin-down) may reduce the lifetime of disks on servers that store data locally, although it may extend the lifetime of other components. One solution is to keep servers stateless and leave the network storage tier powered. A second concern is that power transitions impose a time lag ranging from several seconds to a minute. Neither of these issues is significant if power transitions are damped to occur only in response to daily load shifts, such as those illustrated in Figure 2 and Figure 3. For example, typical modern disk drives are rated for 30,000 to 40,000 start/stop cycles.

4. THE RESOURCE ECONOMY

This section details the system's framework for allocating resources to competing co-hosted services (customers). The basic challenge is to determine the resource demand of each customer at its current request load level, and to allocate resources to their most efficient and productive use. Resources are left idle if the marginal cost to use them (e.g., energy) is less than the marginal value of deploying them to improve performance at current load levels.

To simplify the discussion, we consider a single server pool with a common unit of hosting service resource. This unit could represent CPU time or a combined measure reflecting a share of aggregate CPU, memory, and storage resources. Section 4.6 discusses the problem of provisioning multiple resource classes.

Consider a hosting center with a total of μ_{max} discrete units of hosting resource at time t. The average cost to provide one unit of resource per unit of time is given by $cost(t)$, which may account for factors such as equipment wear or energy prices through time. All resource units are assigned equivalent (average) cost at any time. This represents the *variable* cost of service; the center pays the cost for a resource unit only when it allocates that unit to a customer. The cost and available resource μ_{max} may change at any time (e.g., due to browndown).

Each customer i is associated with a *utility function* $U_i(t, \mu_i)$ to model the value of allocating μ_i resource units to i at time t. Since hosting is a contracted service with economic value, there is an

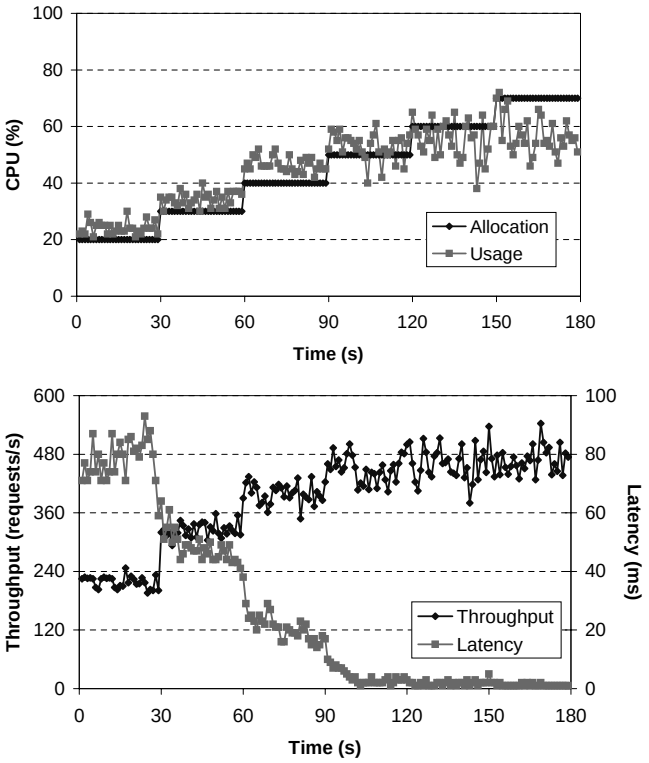

Figure 5: Trading off CPU allocation, throughput, and latency for a synthetic Web workload.

economic basis for evaluating utility [35]. We consider each customer's utility function as reflecting its "bid" for the service volume and service quality resulting from its resource allotment at any given time. Our intent is that utility derives directly from negotiated or offered prices in a usage-based pricing system for hosting service, and may be compared directly to $cost$. We use the economic terms *price*, *revenue*, *profit*, and *loss* to refer to utility values and their differences from cost. However, our approach does not depend on any specific pricing model, and the utility functions may represent other criteria for establishing customer priority (see Section 4.5). Without loss of generality suppose that cost and utility are denominated in "dollars" ($).

The executive's goal of maximizing resource efficiency corresponds to maximizing profit in this economic formulation. Crucially, customer utility is defined in terms of *delivered performance*, e.g., as described in Section 4.1 below. The number of resource units consumed to achieve that performance is known internally to the system, which plans its resource assignments by estimating the the value of the resulting performance effects as described in Section 4.3. Section 4.4 addresses the problem of obtaining stable and responsive load estimates from bursty measures.

4.1 Bids and Penalties

While several measures could act as a basis for utility, we select a simple measure that naturally captures the key variables for services with many small requests and stable average-case service demand per request. Each customer's bid for each unit of time is a function bid_i of its delivered request throughput λ_i, measured in hits per minute ($\$/hpm$). As a simple example, a customer might bid 5 cents per thousand *hpm* up to 10K *hpm*, then 1 cent for each additional thousand *hpm* up to a maximum of one dollar per minute. If 400 requests are served during a minute, then the customer's bid is 2 cents for that minute.

It is left to the system to determine the resources needed to achieve a given throughput level so that it may determine the value of deploying resources for service i. The delivered throughput is some function of the allotment and of the active client load through time: $\lambda_i(t, \mu_i)$. This function encapsulates user population changes, popularity shifts, the resource-intensity of the service, and user think time. The throughput can be measured for the current t and μ_i, but the system must also predict the value of changes to the allotment; Section 4.3 discusses techniques to approximate $\lambda_i(t, \mu_i)$ from continuous performance measures. The system computes bids by composing bid_i and λ_i: the revenue from service i at time t with resource allotment μ_i is $bid_i(\lambda_i(t, \mu_i))$.

Note that bid prices based on $\$/hpm$ reflect service quality as well as load. That is, the center may increase its profit by improving service quality to deliver better throughput at a given load level. Figure 5 illustrates this effect with measurements from a single CPU-bound service running a synthetic Web workload with a fixed client population (see Section 6). The top graph shows the CPU allotment increasing as a step function through time, and the corresponding CPU utilization. (Note that a service may temporarily exceed its allotment on a server with idle resources.) The bottom graph shows the effect on throughput and latency. Figure 5 illustrates the improved service quality (lower latency) from reduced queuing delays as more resources are deployed. As latency drops, the throughput λ increases. This is because the user population issues requests faster when response latencies are lower, a well-known phenomenon modeled by closed or finite population queuing systems. Better service may also reduce the number of users who "balk" and abandon the service. However, as the latency approaches zero, throughput improvements level off because user think time increasingly dominates the request generation rate from each user. At this point, increasing the allotment further yields no added value; thus the center has an incentive to deploy surplus resources to another service, or to leave them idle to reduce cost.

SLAs for hosting service may also impose specific bounds on service quality, e.g., latency. To incorporate this factor into the utility functions, we may include a penalty term for failure to deliver service levels stipulated in an SLA. For example, suppose customer i leases a virtual server with a specific resource reservation of r_i units from the hosting center at a fixed rate. This corresponds to a flat bid_i at the fixed rate, or a fixed minimum bid if the bid_i also includes a load-varying charge as described above to motivate the center to allot $\mu_i > r_i$ when the customer's load demands it. If the request load for service i is too low to benefit from its full reservation, then the center may overbook the resource and deliver $\mu_i < r_i$ to increase its profit by deploying its resources elsewhere: this occurs when $\lambda_i(t, \mu_i) \approx \lambda_i(t, r_i)$ for $\mu_i < r_i$ at some time t. However, since the center collects i's fixed minimum bid whether or not it allots any resource to i, the bid_i may not create sufficient incentive to deliver the full reservation on demand. Thus a penalty is needed.

Since utilization is an excellent predictor of service quality (see Figure 5), the SLA can specify penalties in a direct and general way by defining a maximum target utilization level ρ_{target} for the allotted resource. Suppose that ρ_i is the portion of its allotment μ_i that customer i uses during some interval, which is easily mea-

sured. If $\mu_i < r_i$ and $\rho_i > \rho_{target}$ then the customer is underprovisioned and the center pays a penalty. For example, the $penalty_i$ function could increase with the degree of the shortfall r_i/μ_i, or with the resulting queuing delays or stretch factor [47]. The center must balance the revenue increase from overbooking against the risk of incurring a penalty. The penalty may be a partial refund of the reservation price, or if the penalty exceeds the bid then U_i may take on negative values reflecting the customer's negative utility of reserving resources and not receiving them. Similar tradeoffs are made for seat reservations on American Airlines [37].

4.2 MSRP Resource Allocation

We now describe how the executive allocates resources using an incremental greedy algorithm that we call Maximize Service Revenue and Profit (MSRP). Execution of MSRP is divided into *epochs*. New epochs are declared at regular time intervals, or when some change in the state of the center triggers a reassessment of resource allotments. Examples include load changes beyond some tolerance threshold or changes to available resources μ_{max} due to capacity expansion, failures, or degraded service modes (browndown).

At each epoch the algorithm determines a vector of resource allotments μ_i for N services i. The objective is to maximize profit at each time t:

$$profit(t) = \sum_i^N (U_i(t, \mu_i) - \mu_i * cost(t))$$

The utility for each customer i at time t is given by:

$$U_i(t, \mu_i) = bid_i(\lambda_i(t, \mu_i)) - penalty_i(t, \mu_i)$$

The solution is constrained by the center's maximum capacity μ_{max} at any given time:

$$\mu = \sum_i^N \mu_i \leq \mu_{max}$$

This is a linearly constrained nonlinear optimization problem. To make the problem tractable, our solution assumes that at fixed t each utility function U_i is concave in the following sense. The marginal utility of increasing an allotment μ_i by one resource unit is given by $U_i(t, \mu_i + 1) - U_i(t, \mu_i)$. This is the marginal value or *price* offered by i for its last unit of resource at allotment μ_i: we denote this as $price_i(\mu_i)$ at time t. This price is equivalent to the *utility gradient* or partial derivative $\frac{\delta}{\delta \mu_i} U_i(t, \mu_i)$ if U_i is viewed as continuous over a real-valued domain. The concavity assumption is that this utility gradient must be positive and monotonically nonincreasing. This assumption is common in economic resource allocation [19], and it is natural given our definition of utility. For example, the marginal performance improvement from adding a resource unit declines as μ_i increases to approach and exceed the maximum resource demand for the offered load at time t. Moreover, it is rational for a customer to assign declining marginal value to these declining incremental improvements. Note that the utility functions are concave if the bid and penalty functions are concave.

We specify MSRP using two helper functions $grow(i, target)$ and $shrink(i, target)$. These functions follow the estimated utility gradients up or down, growing or shrinking the planned allotment μ_i for service i. As $shrink$ reclaims resources from a service, the marginal value $price_i(\mu_i)$ increases; as $grow$ assigns more resources the price decreases. These functions shift resources until $price_i(\mu_i) \approx target$ or the resources are exhausted. The next section discusses techniques to estimate these prices when planning resource assignments.

MSRP maximizes revenue and profit by "selling" the available μ_{max} resource units for the highest offered prices at or above cost. The concavity assumption admits a simple and efficient gradient-climbing solution. During each epoch the algorithm executes some or all of the following phases in the following order:

1. *Reclaim resources with negative return.* For any service i with $price_i(\mu_i) \leq cost(t)$: $shrink(i, cost(t))$.
2. *Allot more resources to profitable services.* If $\mu < \mu_{max}$, then there are idle resources to deploy. Determine the highest bidder i with maximum $price_i(\mu_i)$ and the next highest bidder j. Greedily allot resources to the highest bidder i: $grow(i, price_j(\mu_j))$. Repeat, adding resources until $\mu = \mu_{max}$, or the highest remaining bidder's $price_i(\mu_i) < cost(t)$.
3. *Reclaim overcommitted resources.* If $\mu > \mu_{max}$, then reclaim the least profitable resources. This may be needed if browndown or some other failure has reduced μ_{max} from the previous epoch. Determine the lowest-loss service i with minimal $price_i(\mu_i)$, and the next lowest-loss service j. Reclaim from i: $shrink(i, price_j(\mu_j))$. Iterate with the next lowest-loss service, reclaiming resources until $\mu = \mu_{max}$.
4. *Adjust for highest and best use.* If any services i and j exist such that $price_i(\mu_i) < price_j(\mu_j)$, then shift resources from i to j until $price_i(\mu_i) \approx price_j(\mu_j)$. This is done by shifting resources from the lowest to the highest bidders until equilibrium is achieved.

When the algorithm terminates, all resource holders value their last unit of resource equivalently: $price_i(\mu_i)$ is equivalent for all i with $\mu_i > 0$. This price is the *equilibrium price* for resources in the center's economy given the available resources μ_{max} and load at epoch t. If there is no resource constraint then the equilibrium price approaches $cost(t)$. Under constraint the equilibrium price matches the resource demand with the available supply. Establishment of the equilibrium price in step 4 results in a distribution of resources defined as *pareto-optimal*. It can be shown that the pareto-optimal solution is unique and maximizes global utility given our concavity assumption for the utility functions. Equivalently, the nonlinear optimization problem has a global maximum and no false local maxima.

A key property of MSRP and the economic model is that it conserves resources during periods of low demand. Resources are not sold at a price below cost; no resource unit is consumed unless it is expected to yield a meaningful performance improvement.

4.3 Estimating Performance Effects

It remains to show how to estimate the request throughput $\lambda_i(t, \mu_i)$ or resource utilization ρ_i for a planned allotment μ_i to service i in epoch t. These estimates are based on continuous observations of each service i's current throughput λ_i, its aggregate request queue length q_i, and its utilization ρ_i for its current allotment. Section 4.4 describes the technique we use to smooth these observations.

Our scheme uses a common target utilization level ρ_{target} for each ρ_i. Broadly, if $\rho_i < \rho_{target}$ then resources may be conserved by

reducing the allotment μ_i without affecting λ_i. Since utilization is linear with throughput below saturation, we can safely reclaim $\mu_i(\rho_{target} - \rho_i)$ resource units with zero expected loss of revenue. Choosing $\rho_{target} < 1$ leaves surplus capacity for absorbing transient load bursts, and it enables the system to recognize when the service is underprovisioned: if $\rho_i > \rho_{target}$, then it is likely that increasing μ_i will also increase λ_i.

When $\rho_i > \rho_{target}$ the magnitude of the improvement from increasing μ_i may be estimated using a variety of techniques if the model parameters (e.g., population size and think time) are known, which they are not in this case. However, a simple adaptive solution exists: optimistically assume that any new resources allotted to the service will increase throughput linearly with its mean per-request service demand, which is easily determined as $\rho_i\mu_i/\lambda_i$. The system estimates a linear improvement from resource units needed to bring utilization down to ρ_{target}. The risk is that the system may "overshoot" and allot resources beyond the knee of the $\lambda_i(\mu_i)$ throughput curve, i.e., beyond what is needed to bring the service out of saturation. If service i is allotted more resource than it can use productively, then $\rho_i < \rho_{target}$ in a subsequent epoch, leading the algorithm to compensate by reclaiming the unused resources.

Once the system has estimated how adjustments to the resource allotments will affect λ_i, it directly applies the bid_i and $penalty_i$ functions to determine the utility gradients. For example, consider two resource-constrained services A and B. If A bids twice B's bid for throughput increases, but A consumes more than twice as much resource per request, then the system prefers to allocate scarce resources to B. If there is no penalty, then the utility gradient (price) in dollars per resource unit is the incremental bid price per hit per second, divided by the per-request resource demand $\rho_i\mu_i/\lambda_i$.

Although our approach linearly approximates the performance effects of resource shifts in each epoch, the optimization problem is in general nonlinear unless the utility functions are also linear. It is interesting to note that linear utility functions correspond almost directly to a priority-based scheme, with a customer's priority given by the slope of its utility function scaled by its average per-request resource demand. The relative priority of a customer declines as its allotment approaches its full resource demand; this property allocates resources fairly among customers with equivalent priority.

4.4 Feedback and Stability

The executive's predictions depend upon noisy values, in particular the service throughput (λ) and its sister value utilization (ρ). Smoothing these measures to obtain an accurate picture of load is challenging given that Internet service request loads show self-similar bursty behavior [41]. A good load estimator must balance stability—to avoid overreacting to transient load fluctuations—with agility to adapt quickly to meaningful load shifts [31, 25]. In this case, the problem is compounded because the observations constitute a feedback signal. That is, these noisy inputs reflect the effects of resource adjustments in the previous epoch as well as changing load conditions. The system may oscillate if the epochs are too short for the servers to stabilize after each round of adjustments, or if the executive overreacts to noise in the feedback signal. Abdelzaher et. al. [1, 2] use control theory to address stability and responsiveness in a closely related feedback-controlled Web server.

Fortunately, persistent load swells and troughs caused by shifts in population tend to occur on the scale of hours rather than seconds, although service popularity may increase rapidly due to advertising, new content, or other factors. Our current approach uses a principle similar to the *flip-flop filter* devised by Kim and Noble [25] to balance stability and agility. The flip-flop passes a sequence of observations through two exponentially weighted moving average (EWMA) filters with different gain α. Each filter produces an estimate E_t for time t by combining the observation O_t with the previous estimate: $E_t = \alpha E_{t-1} + (1-\alpha)O_t$. Flip-flop selects the result of an agile (low-gain) filter when O_t falls within some tolerance of recent observations, else it selects the output of a stable (high-gain) filter. Flip-flop adapts quickly to load changes and it effectively damps noisy inputs, but it does not smooth the signal sufficiently under normal conditions for our purposes. Our solution—which we call *flop-flip*—holds a stable estimate $E_t = E_{t-1}$ until that estimate falls outside some tolerance of a moving average of recent observations, then it switches the estimate to the current value of the moving average.

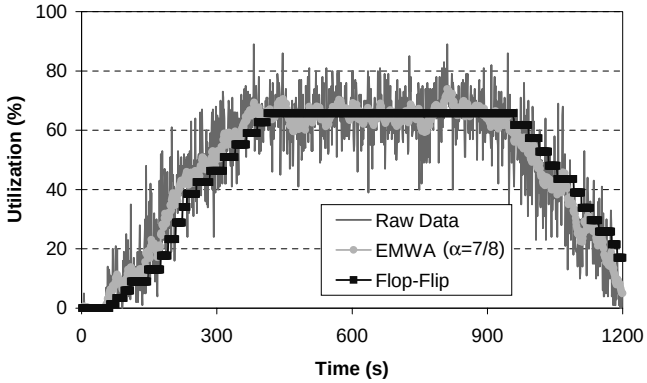

Figure 6: Smoothing a noisy signal with the flop-flip filter.

Figure 6 illustrates an instance of the flop-flip filter smoothing Web server CPU utilization measurements under a synthetic Web load swell (see Section 6). Even during steady load, this bursty signal varies by as much as 40% between samples. The flop-flip filter uses a 30-second sliding window moving average and a step threshold of one standard deviation: it maintains a nearly flat signal through transient load oscillations, and responds to persistent load changes by stepping to another stable level. For comparison, we plot the output from an EWMA filter with a heavily damped $\alpha = 7/8$, the filter used to estimate TCP round-trip time [21]. A flip-flop filter behaves similarly to the EWMA for the "flat" portion of this signal. The flop-flip filter is less agile than the EWMA or flip-flop; its signal is shifted slightly to the right on the graph. However, the step effect reduces the number of unproductive reallocations in the executive, and yields stable, responsive behavior in our environment.

4.5 Pricing

The Muse resource economy defines a general and flexible means to represent business priorities for a hosting service. For example, even if service is offered at a fixed rate, the utility functions can represent priority for valuable customers, e.g., those with longer contract periods. At the system level, utility functions enable a resource management approach based on economic principles, which offers an effective solution to the basic challenges for hosting centers set out in Section 2: conservation of unproductive resources and efficient rationing under constraint caused by power brown-down or load surges beyond aggregate capacity.

Our model is most natural when the utility functions directly reflect prices in a usage-sensitive pricing model. Pricing systems for communication services tend toward simple fixed or tiered pricing schemes, in part because they are simple for consumers and they stimulate demand for resources that are abundant due to advancing technology and deployment [28]. For example, it is common today for customers to pay for network and content transit at a fixed rate based on smoothed expected peak demand over each interval. Customers prefer fixed pricing in part because charges are predictable. However, customers are accustomed to metered pricing for services with high variable cost, such as electrical power or financial services for e-commerce (e.g., credit card processing, which is typically priced as a percentage of the customer's revenue). Relative to communication, hosting has a high variable cost of service for server energy and storage. However, usage-sensitive pricing creates a need for verification services similar to those in place for Internet advertising.

The utility functions offer useful flexibility independent of the pricing model. For usage-sensitive pricing, bid functions may specify a bound to cap costs independent of load; load increases beyond that point cause a decline in service quality. The bid may grow more rapidly at low throughputs and then level off; this reduces the likelihood that a low-value service starves during a period of scarcity, and it reflects economies of scale for high-volume services. With fixed tiered pricing, utility functions can represent refund rates for poor-quality service, or surcharges for service beyond the contract reservation, providing an incentive to upgrade to a higher tier.

An important limitation of our framework as defined here is that customers do not respond to the price signals by adjusting their bids or switching suppliers as the resource congestion level varies. Thus, the system is "economic" but not "microeconomic". Utility functions are revealed in full to the supplier and the bids are "sealed", i.e., they are not known to other customers. While this allows a computationally efficient "market", there is no meaningful competition among customers or suppliers. Our approach could incorporate competition by allowing customers to change their bid functions in real time; if the new utility function meets the concavity assumption then the system will quickly converge on a new utility-maximizing resource assignment.

4.6 Multiple Resources

As implemented in our prototype, Muse manages only one resource: server CPU. We adjusted the performance estimator to control I/O resources in a minimal way for the special case where I/O is to a local replica of stored content, as described in 5.1. This is the only support for controlling I/O resources in our current prototype.

Our approach could extend to manage multiple resources given support in the server node OS for enforcing assignments of disk, memory, and network bandwidth [44, 40, 30]. Economic problems involving multiple complementary goods are often intractable, but the problem is simplified in Muse because customer utility functions specify value indirectly in terms of delivered performance; the resource allotment to achieve that performance need not be visible to customers. Only adjustments to the bottleneck resource (the resource class with the highest utilization) are likely to affect performance for services with many concurrent requests. Once the bottleneck resource is provisioned correctly, other resources may be safely reclaimed to bring their utilization down to ρ_{target} without affecting estimated performance or revenue.

5. PROTOTYPE

The Muse prototype includes a user-level executive server and two loadable kernel modules for the FreeBSD operating system, implementing a host-based redirecting server switch and load monitoring extensions for the servers. In addition, the prototype uses the Resource Containers kernel modules from Rice University [10, 8] as a mechanism to allocate resources to service classes on individual servers. The following subsections discuss key elements of the prototype in more detail.

5.1 Monitoring and Estimation

A kernel module for the servers reports utilization ρ_i, request queue length q_i, and request throughput λ_i for each service i, as described in Section 4.3. A *monitor* module in the executive combines, smooths, and interprets these measures before passing them to the policy controller. The monitor consists of 800 lines of C code compiled into the executive. The prototype measures q_i and λ_i in units of TCP connections: q_i is the aggregate TCP accept queue length, and λ_i is measured in TCP FIN-ACKs per second. These correspond to requests in HTTP 1.0. Precise measures for HTTP 1.1 or other service protocols using persistent connections may require extensions to application server software.

The q_i observations may be taken as a measure of service quality or to detect variations in the balance between CPU and I/O for each service. If a service is not reaching the target utilization level for its resource allotment ($\rho_i < \rho_{target}$), but smoothed queue lengths exceed a threshold, then the monitor assumes that the service is I/O bound. Our prototype adapts by dropping the ρ_{target} for that service to give it a larger share of each node and avoid contention for I/O. Once the ρ_{target} is reached, the system gradually raises it back to the default, unless more queued requests appear. Queued requests may also result from transient load bursts or sluggish action by the executive to increase the allotment when load is in an uptrend. In this case, the adjustment serves to temporarily make the executive more aggressive.

5.2 The Executive

The executive runs as a user-level server on a dedicated machine. In addition to the load monitor code, it includes 300 lines of C code implementing the policies described in Section 4 for a single pool of homogeneous servers. In our experiments the executive spreads request loads from all services evenly across the entire active server pool. The resource container allotment for service i is set on every active server as a percentage of total μ allotted to all services in that epoch. In practice, it may be desirable to partition the server pool for stronger security or fault isolation [35, 5].

The executive emits a sequence of commands to control the active server sets, server power states, and resource allotments. A separate *actuator* program triggers the state changes by passing commands to the switches and servers. For example, the actuator changes resource allotments by issuing *rsh* to remotely execute tools from the resource container release. Decoupling the executive from the actuator allows us to test and debug the executive by running it in trial mode or using saved load information as its input.

The actuator uses Advanced Power Management (APM) tools for Intel-based systems to retire excess servers by remotely transitioning them to a low-power state. The current prototype recruits and retires servers by rank; we have not investigated policies to select target servers to spread thermal load or evenly distribute the start/stop cycles. The low-power state leaves the network adapter

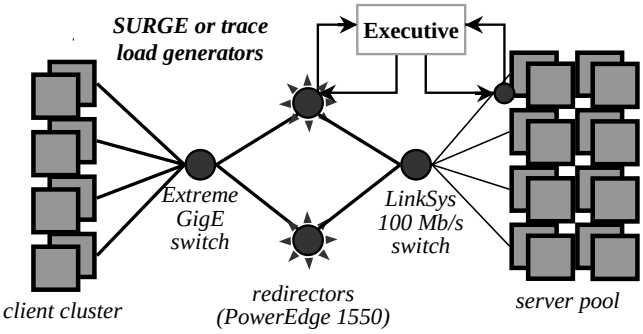

Figure 7: The hosting center testbed.

powered; off-power servers are awakened with a tool that sends a predefined *Wake-On-LAN* packet pattern to the Ethernet MAC address, which initiates a boot in the BIOS. Power draw in the low-power state is 3-5 watts on our systems, equivalent to powering off the machine (power draw only ceases after physically unplugging the machine). It takes about a minute to bring a recruited server on line, but we expect that future ACPI-capable servers will support warm low-power states to reduce this delay.

The resource allocation algorithm runs in at worst $N + \mu N$ time, where μ is the number of resource units reallocated in the epoch and N is the number of services. (This assumes that the utility functions execute in constant time, such as a table lookup.) Thus the executive is fast when the cluster is stable, but it is relatively expensive to adapt to significant changes in the load, resource availability (μ_{max}), or utility functions. One potential source of runtime cost is evaluating the utility functions themselves, which are loaded into the executive in our prototype. The algorithm evaluates these functions to determine the highest and lowest bidders for each reallocated resource unit. The base resource unit granularity is 1% of a server CPU; an initialization parameter allows grouping of resource units to reduce computational cost at the expense of precision.

Future work will explore decentralizing the executive functions to eliminate it as a bottleneck or single point of failure. The cluster adapts only when the executive is functioning, but servers and switches run autonomously if the executive fails. The executive may be restarted at any time; it observes the cluster to obtain adequately smoothed measures before taking action.

5.3 The Request Redirector

The reconfigurable load-balancing switches in the prototype are host-based, implemented by a *redirector* module that interposes on the kernel TCP handler. It intercepts TCP packets addressed to registered logical service ports on the switch, and redirects them to selected servers. The redirector uses Network Address Translation and incremental checksum computation to adjust packets from each flow before redirecting them. All response traffic also passes through the redirector. This behavior is typical of a commercial "Layer-4" load-balancing switch serving virtual TCP/IP endpoints. For our experiments we used a simple round-robin policy to select servers from the active set.

The redirector also gathers load information (packet and connection counts) from the redirected connections, and maintains active server sets for each registered service endpoint. The active sets are lists of *(IPaddr, TCPport)* pairs, allowing sharing of an individual server by multiple services. Active set membership is controlled by the executive and its actuator, which connects to the redirector through a TCP socket and issues commands using a simple protocol to add or remove servers. This allows the executive to dynamically reconfigure the active sets for each service, for example, to direct request traffic away from retired (off-power) servers and toward recruited (active) servers.

6. EXPERIMENTAL RESULTS

This section presents experimental results from the Muse prototype to show the behavior of dynamic server resource management. These experiments consider the simple case of a symmetric cluster with a single server pool.

6.1 Experimental Setup

Figure 7 depicts the data center testbed, which consists of a server pool driven by traffic-generating clients through two redirecting switches. The servers are 450 MHz Pentium-III systems (Asus P2B/440BX). These servers run FreeBSD 4.3 and multiple instances of Apache version 1.3.12, each in its own resource container. The redirectors are Dell PowerEdge 1550 rackmounts with 1GHz Pentium-III CPUs. These systems have dual Gigabit Ethernet NICs connected to gigabit ports on the server's switch, a Linksys Gigaswitch, and the client's switch, an Extreme Summit 7i. Each of the redirectors' NICs is connected to a separate 64-bit 66 MHz PCI bus. The servers are connected to the Linksys switch through 100 Mb/s links. Each redirector can carry sufficient traffic to saturate the server pool under a standard HTTP 1.0 Web workload, but latencies increase significantly near redirector saturation.

Our experiments use a combination of synthetic traffic and real request traces. We use a server log trace from IBM Corporation's main Web server (www.ibm.com), with 43M requests received during February 5-11, 2001. To show the impact with multiple services under varying loads, we use a modified version of the SURGE [11] synthetic Web service load generator. SURGE generates highly bursty traffic characteristic of Web workloads, with heavy-tailed object size distributions so that per-request service demand is highly variable. These factors stress the prototype's load estimation and resource allocation algorithms. One advantage of synthetic traffic is that the load generator is closed, meaning that the request arrival rate is sensitive to the response latency in the manner discussed in Section 4.1. Also, we can generate synthetic load "waves" with any amplitude and period by modulating the number of generators.

The experiments presented are designed to illustrate various aspects of the system's behavior in simple scenarios. This version of resource containers controls only the CPU resource, thus our experiments vary only the CPU allotments. All servers hold a complete replica of the Web service file set, and our workloads serve a large majority of requests from memory so that the CPU is the critical resource. Section 4.6 discusses the problem of multiple resources. All experiments use 10-second epochs, fixed $cost = 1$, and one-second measurement intervals.

6.2 Allocation Under Constraint

The first two experiments demonstrate competitive allocation of server resources in a cluster that is inadequately provisioned for unexpected load swells. These experiments offer synthetic request traffic to two services (**s0** and **s1**) with linear utility as a function of throughput. Both services have equivalent file sets, so they have the same average per-request service demand. They receive different load signals, generated by a set of eight machines running SURGE

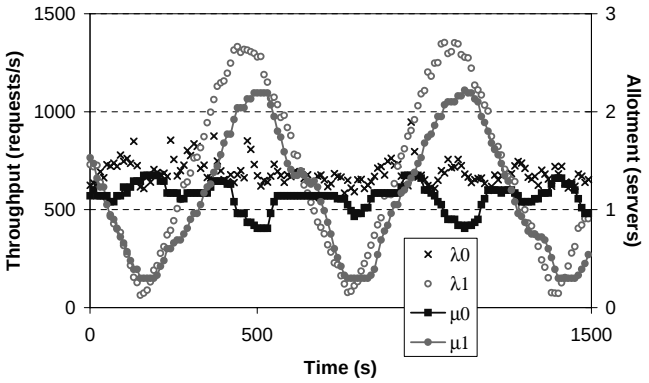

Figure 8: Competitive bidding with two services, favoring s1.

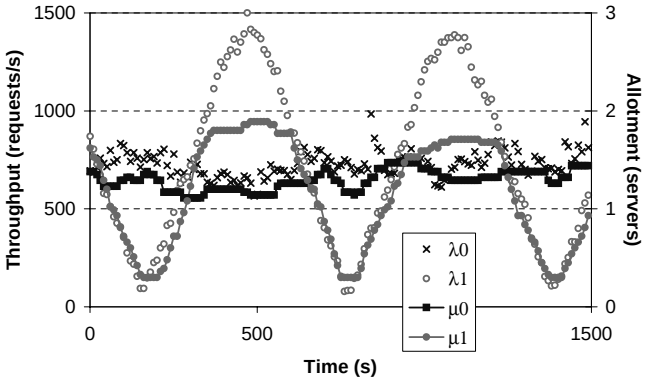

Figure 9: Competitive bidding with two services, favoring s0.

load generators. Service **s0** receives a fixed load, while service **s1** experiences rapid load variations with two major swells over a 25-minute period peaking at double the load for **s0**. The experiment is constrained to three servers, and the load is comfortably handled by two servers during the troughs for **s1**. These experiments use $\rho_{target} = 0.8$.

Figure 8 and Figure 9 demonstrate competitive resource rationing under this scenario. Each figure plots total service throughput (λ) on one y axis and total resource allotment (μ) on the other. The figures report allotments in units of complete servers; the allocation grain is 1% of a server ($\mu_{max} = 300$).

In Figure 8, the oscillating **s1** bids higher than **s0** for each unit of its throughput. When the **s1** load swells, the executive responds to a resource shortage by reducing the allotment for **s0**, pushing its throughput down. Initially each service receives enough resources to handle its load, until the oscillating load swells near time index 300. At that point, **s0** is depressed until the swell reverses and the executive reclaims unused resources from **s1**.

The second experiment in Figure 9 is identical to the first, except that the utility functions are reversed so that the fixed **s0** bids higher than **s1**. The behavior is similar when there are sufficient resources to handle the load. During the **s1** swells the executive preserves the allocation for the higher-bidding **s0**, doling out most of the remaining resources to **s1**.

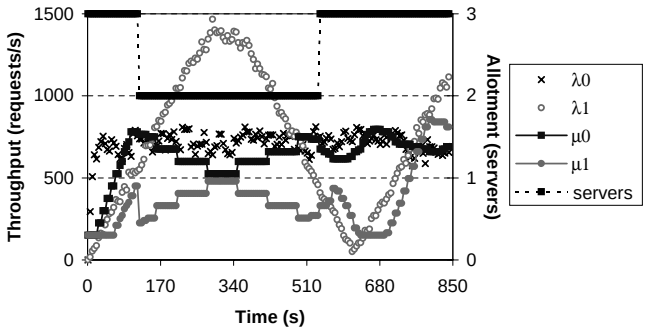

Figure 10: Competitive bidding during browndown.

6.3 Browndown

The next experiment, reduces μ_{max} for an active cluster hosting competing services, emulating a server failure or browndown. We use loads for services **s0** and **s1** similar to the experiments above, but favoring the fixed **s0**. Initially both services are idle with the minimum allotment and there are three servers available. As load builds, each service allotment increases until time 120 when the maximum cluster size shrinks to two servers ($\mu_{max} = 200$).

Figure 10 illustrates the behavior. Muse responds to the loss by shifting load to the remaining servers and degrading service for **s1**. However, shortly after the failure, at $t = 170$, the request load for **s1** exceeds **s0**. Although **s0** bids higher *per request*, the executive shifts resources away from **s0** because the same resources will satisfy more hits for the badly saturated **s1**, earning higher overall utility.

The third server is restored at $t = 540$. Muse allocates additional resources to both services until **s1** offered load drops and the system reclaims its excess allocation. The subsequent swell starting at $t = 680$ is similar to Figure 9, favoring **s0** as the swell peaks.

6.4 Varying Load and Power

The next experiments evaluate the energy savings from energy-conscious provisioning for a typical Web workload. Figure 11 shows throughput, energy consumption, and request latency for the full week of the IBM trace, played at 16x speedup using open trace replay software [18] that is faithful to the logged request arrival times. These 10-hour experiments used five servers and one redirector. A Brand Electronics 21-1850/CI digital power meter logs average aggregate server power draw once per second. In the baseline experiment (left), power levels vary within a narrow range of 190-240 watts proportionally to request load. The total energy consumption for the run was 2.38 KWh.

The second experiment (right) shows a similar run with energy-conscious provisioning enabled and $\rho_{target} = 0.5$. Total energy consumption for this run was 1.69 KWh, a savings of 29%. The right-hand graph reflects the executive's more conservative policies for retiring machines during periods of declining load, yielding somewhat higher energy consumption than necessary. The average load for this trace is 1010 *hits per second* with a peak load of 2043 *hps*. Thus the idealized energy savings for the full week is bounded above by 50%; the Muse prototype yields a significant share of this savings, even with the trace sped up by 16x. Response latencies are slightly higher under light load, reflecting the resizing of server sets to maintain server utilization at ρ_{target}.

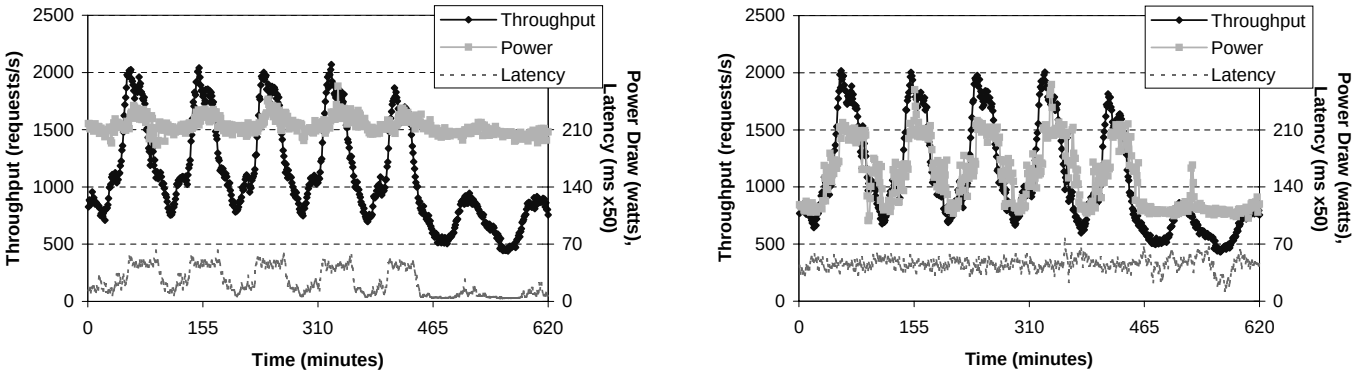

Figure 11: Throughput, power, and latency for the IBM trace at 16x with (right) and without (left) energy-conscious provisioning.

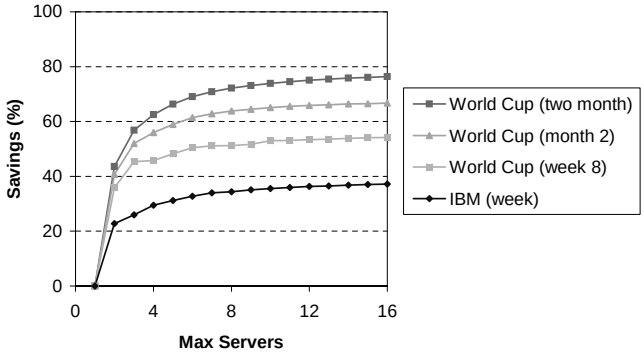

Figure 12: Potential energy savings for the IBM and World Cup traces with increasing server requirements.

While the absolute energy savings of 690 watt-hours is modest for the IBM trace experiment in Figure 11, a similar relative savings can be expected for more intensive workloads over longer periods. The overall savings from energy-conscious server management is increased by overprovisioning and seasonal load fluctuations. Also, the potential relative savings increases in larger clusters, because dynamic resizing of the cluster may occur on a finer relative granularity to more closely approximate the load curve.

To illustrate, Figure 12 compares the potential energy savings for the IBM and World Cup load curves as a function of cluster size. This determines the savings available for a service with identical relative load fluctuations but a higher peak resource requirement due to more resource-intensive requests, larger client populations and higher request rates, or a lower ρ_{target}. These results were obtained by passing the per-second request rates in the trace through the flop-flip filter, then computing the number of active servers required to serve the trace load at ten-minute intervals, under the assumption of stable average per-request resource demand and the peak resource demand given on the x-axis. The y-axis gives the corresponding approximate energy savings relative to static provisioning. The projections assume power draw linear with load between the *idle* and *max* measures for the SuperMicro 370-DER servers given in Table 1.

Server energy savings approaches 38% for the IBM trace and up to 78% for the full World Cup trace, realizing higher savings as the cluster size increases. In essence, the given load is discretized according to the number of servers. With larger clusters, the "rounding error" for excess capacity declines. Figure 12 shows that managing energy at the granularity of complete servers is sufficient for clusters of a few dozen nodes or more.

These results approximate the potential overall savings in energy-related operating costs from this technique. Most energy consumed for data center machine rooms goes to servers and cooling [29]. The results presented here do not reflect the energy savings from reduced cooling load; while ventilation and power conditioning impose modest fixed costs, energy consumption in chillers is proportional to thermal loading from active servers. Savings may be partially offset by off-peak maintenance activities or geographical load-shifting from time zones in peak load toward areas with light load. Although more data is needed, the results justify an expected potential overall energy savings of 25% or more for Web data centers.

7. RELATED WORK

Cluster-based network services. Many systems have used server clusters and redirecting front-ends or switches to virtualize network services (e.g., [32, 36, 7, 4]) for scalability, availability, or both. Muse uses reconfigurable redirecting switches as a mechanism to support adaptive resource provisioning in network server pools; related projects considering this approach are DDSD [47] and Oceano [5]. The switch features needed are similar to the support for virtual LANs, request load balancing, and server failover found on commercial switches for server traffic management, which are now widely used in large Internet server sites.

Service hosting. Our focus on hosting services targets environments in which physical servers are shared among competing server applications. Many projects have addressed the related problem of providing a configurable and secure execution environment for application components on generic servers. WebOS [42] framed the problem and proposed solutions for a wide-area hosting utility; Oceano [5] is one project addressing this challenge for hosting centers. One solution gaining commercial acceptance is to encapsulate service components in separate virtual machines (e.g., VMware). Our approach to server provisioning is applicable to each of these contexts. Although this paper focuses on a single data center, a similar approach could manage resources for a wide-area hosting network or content services network.

Resource principals. Muse associates each service with a resource principal recognized by the operating system, and allocates resources to these principals to apply provisioning choices securely and with-

out burdening server applications. Decoupling resource principals from the kernel's process and thread abstractions improves generality: examples in the literature include resource groups in Opal [14], activities in Rialto [24], Software Performance Units [44], scheduling domains in Nemesis [30], and the resource containers [10] implemented for FreeBSD by Mohit Aron [8] and used in our prototype. Aron has shown how to extend resource principals to *cluster reserves* spanning nodes [9, 8].

Scheduling for performance isolation. To ensure performance isolation, resource allotments to resource principals are enforced by kernel-based schedulers for each node's physical resources. For example, CPU time allotments may be expressed and enforced using reservations (e.g., as in Rialto [24]) or a proportional share discipline such as the lottery scheduling [46] used for FreeBSD resource containers. Several groups have shown how to implement performance-isolating schedulers for other resources including memory, network bandwidth, and storage access [44, 40, 30]. Recent work addresses performance isolation for co-hosted virtual services built above shared database servers or other supporting tiers [34].

Software feedback. Our work uses these resource control mechanisms to adaptively allocate system resources to competing consumers whose resource demands are not known *a priori*. This approach is related to feedback scheduling, which is used to dynamically characterize the resources needed to maintain specified rates of application progress [38]. Previous work on feedback scheduling has focused primarily on CPU scheduling for continuous media and other real-rate applications, while we use feedback to adaptively provision resources for hosted network services whose resource demands vary dynamically with request load. Aron [8] uses kernel-based feedback schedulers to meet latency and throughput goals for network services running on a shared server. Abdelzaher et. al. [1, 2] are also investigating this approach and have addressed the control-theoretical aspects. IBM mainframe systems use feedback to optimize resource usage to meet a range of progress and quality goals for mixed workloads on clusters [3]; their approach could also apply to network services, as in Oceano [5]. In Muse, the feedback and adaptation policies are decoupled from the node operating systems and applied across the entire server pool, using reconfigurable switches to enable dynamic assignment of active server sets for each service.

Admission control and QoS. A substantial body of related work addresses predictable or guaranteed quality-of-service (QoS) bounds for scheduling shared resources — primarily CPUs and networks — among consumers with varying demands. Continuously monitoring of load levels is related to measurement-based admission control, which has been used to maintain predictive service levels for network flows [22], clients of a Web server [27], and services on a shared server [8]. The approach for co-hosted Web services in [2] relies on *a priori* characterizations of peak demands for each service, together with feedback-based degraded content fidelity when a server is overloaded. Muse currently addresses QoS bounds only indirectly through utility functions that place a premium on higher throughput and impose a penalty when a customer is starved. Even with admission control, transient resource constraints may occur as a result of failures or power "browndown" events.

Economic resource allocation. Use of continuous utility functions as a basis for planning resource allocation under constraint is ubiquitous in markets and auctions. Reference [16] and [19] survey earlier computer science research on market-based resource allocation. Recent systems have experimented with market-based models for wide-area database systems (Mariposa [39]), distributed computation (e.g., Spawn [45]) and energy management for mobile systems [30]. Muse extends these ideas to enable continuous, fine-grained service quality control for co-hosted network services. Service resource demands vary with time and user population, while computational workloads in principle have unbounded resource demand rates until an answer is produced. The last few years have seen an explosion of interest in market-based allocation of network bandwidth during the Internet's transition to a commercial service; reference [28] is a collection of perspectives. The economics of service hosting are related but distinct, as discussed in Section 4.5.

Energy management. A key distinction from previous work is that our approach is directed not just at scheduling shared resources effectively, but also at reducing on-power server capacity to match demand, as a natural outgrowth of a comprehensive resource management architecture for hosting utilities. Energy-conscious switching was recently proposed independently [12, 33] and appears to be new in the literature. It adjusts on-power capacity at the coarse granularity of entire servers, as a complementary but simpler and more effective alternative to fine-grained techniques (e.g., frequency scaling or variable voltage scaling on new CPU architectures such as Transmeta Crusoe), which reduce CPU power demand under light load but do not address energy losses in the power supply. Our approach also enables the system to support flexible tradeoffs of service quality for power under constraint (e.g., cooling failure) in a manner similar to dynamic thermal management [13] within an individual server.

Previous research on power management focuses on mobile systems, which are battery-constrained. We apply similar concepts and goals to Internet server clusters; in this context, energy-conscious policies are motivated by cost, energy efficiency, and the need to manage power supply disruptions or thermal events. The focus on servers adds a new dimension to *power-aware resource management* [43], which views power as a first-class resource to be managed by the operating system [15, 26]. One aspect we have not investigated in the server context is the role of application-specific adaptation to resource constraints [31, 17, 30, 2]. If it is necessary to degrade service quality, our approach merely increases response time by reducing resource allotments without participation from server applications.

Load estimation. Fundamental to server resource management is the load estimation problem: how much resource is needed to meet service quality goals for the current load level? This is related to the problem of characterizing network flows. Most approaches used exponential moving averages of point samples or average resource utilization over some sample period [23, 27, 2, 9, 8]. In [2], resource demands are derived from request counts and bytes transferred, using a model of a static Web server. Kim and Noble have compared several statistical methods for estimating available bandwidth on a noisy link [25]; their study laid the groundwork for the "flop-flip" filter in Muse.

8. CONCLUSION

This paper describes the design and implementation of Muse, a resource management architecture for hosting centers. Muse defines policies for *adaptive resource provisioning* in hosting centers using an economic approach. A principal objective is to incorporate energy management into a comprehensive resource manage-

ment framework for data centers. This enables a center to improve energy efficiency under fluctuating load, dynamically match load and power consumption, and respond gracefully to transient resource shortages caused by exceptional load surges or "brown-down" events that disrupt the power supply or thermal systems.

The Muse resource economy is based on executable utility functions that quantify the value of performance for each co-hosted service. The system plans resource assignments to maximize "profit" by balancing the cost of each resource unit against the estimated marginal utility or "revenue" that accrues from allocating that resource unit to a service. To predict the effects of planned resource adjustments, the system estimates resource demands based on continuous performance observations. This enables the center to use available resources efficiently, set resource "prices" to balance supply and demand, and minimize the impact of shortages. It also provides a foundation for a market-based approach to competitive third-party application hosting in the wide area.

This paper makes the following contributions:

- It shows how to use reconfigurable network-level request redirection to route incoming request traffic toward dynamically provisioned server sets. This also enables *energy-conscious provisioning*, which concentrates request load on a subset of servers. Muse leverages server power management to control the center's overall power level by automatically transitioning the power states of idle servers.
- It illustrates a simple adaptive policy to dynamically allot sufficient resources for each service to serve its current load at some selected quality level, reducing the need for static overprovisioning. This enables a center to balance short-term load shifts that may be out-of-phase across multiple services, and respond automatically to long-term popularity shifts.
- It shows how an economic framework for server resource allocation allows informed tradeoffs of service quality during shortages. The utility functions capture the key performance criteria embodied in Service Level Agreements (SLAs), enabling direct expression of a continuum of SLAs balancing service quality and cost. The price-setting algorithm determines efficient resource assignments by "selling" the available resources at a profit to the highest bidders. The system may choose not to sell idle capacity when it is economically preferable to step down that capacity to reduce costs.
- It describes a Muse prototype for experimentation in data center testbeds. Experimental results from the prototype demonstrate its potential to adapt service provisioning to respond to dynamically varying resource availability and cost in a server cluster. The prototype can reduce server energy consumption by 29%-78% for representative Web workloads.

Further research is needed to expand the generality of our approach. Examples of desirable capabilities for a complete resource management system are: a decentralized market executive, partitioning of the server set, integration with externally controlled cooling systems, rapid response to load shifting across centers, market competition among customers and suppliers, automatic handling of diverse software configurations in multiple server pools, performance and utility measures for a wider range of server applications, and coordinated management of other resources including storage.

Acknowledgements

We thank our shepherd Mike Jones, Terence Kelly, Rolf Neugebauer, Geoff Voelker, and the anonymous reviewers for their comments on earlier drafts. The paper also benefited from discussions with Ricardo Bianini, Ed Bugnion, Dave Farber, Rich Friedrich, Kevin Jeffay, Brian Noble, Ron Parr, Chandrakant Patel, Timothy Roscoe, Mike Spreitzer, Kishor Trivedi, Jeff Vitter, and John Wilkes. We also thank Alister Lewis-Bowen and Andrew Frank-Loron for their help in obtaining the 2001 IBM server logs, Mootaz Elnozahy for providing preprints of related papers on server energy at IBM, Andrew Gallatin and David Becker for helping out with experiments, and Mohit Aron for sharing his code for Resource Containers in FreeBSD.

9. REFERENCES

[1] Tarek F. Abdelzaher and Chenyang Lu. Modeling and Performance Control of Internet Servers. In *39th IEEE Conference on Decision and Control*, December 2000.

[2] 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*, June 2001.

[3] Jeffrey Aman, Catherine K. Eilert, David Emmes, Peter Yocom, and Donna Dillenberger. Adaptive Algorithms for Managing a Distributed Data Processing Workload. *IBM Systems Journal*, 36(2), 1997.

[4] Darrell C. Anderson, Jeffrey S. Chase, and Amin M. Vahdat. Interposed Request Routing for Scalable Network Storage. In *Proceedings of the Fourth Symposium on Operating System Design and Implementation (OSDI)*, October 2000.

[5] Karen Appleby, Sameh Fakhouri, Liana Fong, Germán Goldszmidt, Michael Kalantar, Srirama Krishnakumar, Donald Pazel, John Pershing, and Benny Rochwerger. Oceano - SLA Based Management of a Computing Utility. In *Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management*, May 2001.

[6] Martin Arlitt and Tai Jin. Workload Characterization of the 1998 World Cup Web Site. Technical Report HPL-1999-35R1, HP Laboratories, September 1999. The trace is available from the Internet Traffic Archive at ita.ee.lbl.gov.

[7] Armando Fox and Steven D. Gribble and Yatin Chawathe and Eric A. Brewer and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles (SOSP)*, pages 78–91, October 1997.

[8] Mohit Aron. *Differentiated and Predictable Quality of Service in Web Server Systems*. PhD thesis, Department of Computer Science, Rice University, October 2000.

[9] Mohit Aron, Peter Druschel, and Willy Zwaenepoel. Cluster Reserves: A Mechanism for Resource Management in Cluster-based Network Servers. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS 2000)*, pages 90–101, June 2000.

[10] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)*, February 1999.

[11] Paul Barford and Mark E. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '98)*, pages 151–160, June 1998.

[12] Pat Bohrer, Elmootazbellah N. Elnozahy, Tom Keller, Michael Kistler, Charles Lefurgy, and Ray Rajamony. The Case for Power Management in Web Servers. In *Power-Aware Computing (Robert Graybill and Rami Melhem, editors)*. Kluwer/Plenum series in Computer Science, to appear, January 2002.

[13] David Brooks and Margaret Martonosi. Dynamic Thermal Management for High-Performance Microprocessors. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture (HPCA-7)*, January 2001.

[14] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and Protection in a Single Address Space Operating System. *ACM Transactions on Computer Systems*, 12(4), November 1994.

[15] Fred Douglis, P. Krishnan, and Brian Bershad. Adaptive Disk Spin-down Policies for Mobile Computers. In *2nd USENIX Symposium on Mobile and Location-Independent Computing*, April 1995. Monterey CA.

[16] Donald F. Ferguson, Christos Nikolaou, Jakka Sairamesh, and Yechiam Yemini. Economic Models for Allocating Resources in Computer Systems. In *Market-Based Control: A Paradigm for Distributed Resource Allocation (Scott H. Clearwater, editor)*. World Scientific, 1996.

[17] Jason Flinn and Mahadev Satyanarayanan. Energy-aware Adaptation for Mobile Applications. In *Proceedings of the Seventeenth Symposium on Operating Systems Principles (SOSP)*, pages 48–63, December 1999.

[18] Syam Gadde. The Proxycizer Web Proxy Tool Suite. `http://www.cs.duke.edu/ari/Proxycizer/`.

[19] Toshihide Ibaraki and Naoki Katoh, editors. *Resource Allocation Problems: Algorithmic Approaches*. MIT Press, Cambridge, MA, 1988.

[20] Arun Iyengar, Jim Challenger, Daniel Dias, and Paul Dantzig. High-performance Web Site Design Techniques. *IEEE Internet Computing*, 4(2):17–26, March 2000.

[21] Van Jacobson. Congestion Avoidance and Control. *ACM Computer Communication Review: Proceedings of the SIGCOMM Symposium*, 18(4):314–329, August 1988.

[22] Sugih Jamin, Peter B. Danzig, Scott J. Shenker, and Lixia Zhang. A Measurement-based Admission Control Algorithm for Integrated Services Packet Networks. *IEEE/ACM Transactions on Networking*, 5(1):56–70, February 1997.

[23] Sugih Jamin, Scott J. Shenker, and Peter B. Danzig. Comparison of Measurement-based Admission Control Algorithms for Controlled-Load Service. In *Proceedings of IEEE Infocom 1997*, April 1997.

[24] Michael B. Jones, Daniela Rosu, and Marcel-Catalin Rosu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 198–211, October 1997.

[25] Minkyong Kim and Brian Noble. Mobile Network Estimation. In *Proceedings of the Seventh Annual Conference on Mobile Computing and Networking*, July 2001.

[26] Alvin R. Lebeck, Xiaobo Fan, Heng Zeng, and Carla S. Ellis. Power-Aware Page Allocation. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, 2000.

[27] Kelvin Li and Sugih Jamin. A Measurement-Based Admission-Controlled Web Server. In *Proceedings of IEEE Infocom 2000*, March 2000.

[28] Lee W. McKnight and Joseph P. Bailey, editors. *Internet Economics*. MIT Press, Cambridge, MA, 1997.

[29] Jennifer D. Mitchell-Jackson. Energy Needs in an Internet Economy: A Closer Look at Data Centers. Master's thesis, Energy and Resources Group, University of California at Berkeley, July 2001.

[30] Rolf Neugebauer and Derek McAuley. Energy is Just Another Resource: Energy Accounting and Energy Pricing in the Nemesis OS. In *Proceedings of the Eighth IEEE Workshop on Hot Topics in Operating Systems HotOS-VIII*, pages 59–64, May 2001.

[31] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile Application-Aware Adaptation for Mobility. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 276–287, Saint Malo, France, October 1997.

[32] Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenopoel, and Erich Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, October 1998.

[33] 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.

[34] John Reumann, Ashish Mehra, Kang G. Shin, and Dilip Kandlur. Virtual Services: A New Abstraction for Server Consolidation. In *Proceedings of the USENIX 2000 Technical Conference*, June 2000.

[35] Timothy Roscoe and Prashant Shenoy. New Resource Control Issues in Shared Clusters. In *Proceedings of the Eight International Workshop on Interactive Distributed Multimedia Systems (IDMS'01)*, September 2001.

[36] Yasushi Saito, Brian N. Bershad, and Henry M. Levy. Manageability, Availability and Performance in Porcupine: A Highly Scalable Cluster-Based Mail Service. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–15, Kiawah Island, December 1999.

[37] Barry C. Smith, John F. Leimkuhler, and Ross M. Darrow. Yield Management at American Airlines. *Interfaces*, 22(1), January 1992.

[38] David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A Feedback-driven Proportion Allocator for Real-Rate Scheduling. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)*, pages 145–158, February 1999.

[39] Michael Stonebraker, Paul M. Aoki, Witold Litwin, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin, and Andrew Yu. Mariposa: A Wide-Area Distributed Database System. *VLDB Journal: Very Large Data Bases*, 5(1):48–63, 1996.

[40] David G. Sullivan and Margo I. Seltzer. Isolation with Flexibility: A Resource Management Framework for Central Servers. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 337–350, June 2000.

[41] Kevin Thompson, Gregory J. Miller, and Rick Wilder. Wide-Area Internet Traffic Patterns and Characteristics. In *IEEE Network*, November 1997.

[42] Amin Vahdat, Thomas Anderson, Michael Dahlin, Eshwar Belani, David Culler, Paul Eastham, and Chad Yoshikawa. WebOS: Operating System Services for Wide-Area Applications. In *Proceedings of the Seventh IEEE Symposium on High Performance Distributed Computing (HPDC)*, Chicago, Illinois, July 1998.

[43] Amin Vahdat, Alvin R. Lebeck, and Carla S. Ellis. Every Joule is Precious: The Case for Revisiting Operating System Design for Energy Efficiency. In *Proceedings of the 9th ACM SIGOPS European Workshop*, September 2000.

[44] Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Performance Isolation: Sharing and Isolation in Shared Memory Multiprocessors. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1998.

[45] Carl A. Waldspurger, Tad Hogg, Bernardo A. Huberman, Jeffrey O. Kephart, and W. Scott Stornetta. Spawn: A Distributed Computational Economy. *IEEE Transactions on Software Engineering*, 18(2):103–117, February 1992.

[46] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–11, November 1994.

[47] Huican Zhu, Hong Tang, and Tao Yang. Demand-driven Service Differentiation in Cluster-Based Network Servers. In *Proceedings of IEEE Infocom 2001*, April 2001.