USENIX Association

Proceedings of USITS '03: 4th USENIX Symposium on Internet Technologies and Systems

Seattle, WA, USA
March 26–28, 2003

© 2003 by The USENIX Association All Rights Reserved For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738 Email: office@usenix.org WWW: http://www.usenix.org
Rights to individual papers remain with the author or the author's employer.
Permission is granted for noncommercial reproduction of the work for educational or research purposes.
This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

Model-Based Resource Provisioning in a Web Service Utility

Ronald P. Doyle*
IBM
Research Triangle Park
rdoyle@us.ibm.com

Jeffrey S. Chase
Omer M. Asad†
Wei Jin
Amin M. Vahdat
Department of Computer Science‡
Duke University
{chase, jin, vahdat}@cs.duke.edu

Abstract

Internet service utilities host multiple server applications on a shared server cluster. A key challenge for these systems is to provision shared resources on demand to meet service quality targets at least cost. This paper presents a new approach to utility resource management focusing on coordinated provisioning of memory and storage resources. Our approach is *model-based*: it incorporates internal models of service behavior to predict the value of candidate resource allotments under changing load. The model-based approach enables the system to achieve important resource management goals, including differentiated service quality, performance isolation, storage-aware caching, and proactive allocation of surplus resources to meet performance goals. Experimental results with a prototype demonstrate model-based dynamic provisioning under Web workloads with static content.

1 Introduction

The hardware resources available to a network service determine its maximum request throughput and—under typical network conditions—a large share of the response delays perceived by its clients. As hardware performance advances, emphasis is shifting from server software performance (e.g., [19, 26, 27, 37]) to improving the manageability and robustness of large-scale services [3, 5, 12, 31]. This paper focuses on a key subproblem: automated on-demand resource provisioning for multiple competing services hosted by a shared server infrastructure—a utility. It applies to Web-based services in a shared hosting center or a Content Distribution Network (CDN).

The utility allocates each service a *slice* of its resources, including shares of memory, CPU time, and available throughput from storage units. Slices provide performance isolation and enable the utility to use its resources efficiently. The slices are chosen to allow each hosted service to meet service quality targets (e.g., response time) negotiated in Service Level Agreements (SLAs) with the utility. Slices vary dynamically to respond to changes in load and resource status. This paper addresses the *provisioning* problem: how much resource does a service need to meet SLA targets at its projected load level? A closely related aspect of utility resource allocation is *assignment*: which servers and storage units will provide the resources to host each service?

Previous work addresses various aspects of utility resource management, including mechanisms to enforce resource shares (e.g., [7, 9, 36]), policies to provision shares adaptively [12, 21, 39], admission control with probabilistically safe overbooking [4, 6, 34], scheduling to meet SLA targets or maximize yield [21, 22, 23, 32], and utility data center architectures [5, 25, 30].

The key contribution of this paper is to demonstrate the potential of a new *model-based* approach to provisioning multiple resources that interact in complex ways. The premise of model-based resource provisioning (MBRP) is that internal models capturing service workload and behavior can enable the utility to predict the effects of changes to the workload intensity or resource allotment. Experimental results illustrate model-based dynamic provisioning of memory and storage shares for hosted Web services with static content. Given adequate models, this approach may generalize to a wide range of services including complex multi-tier services [29] with interacting components, or services with multiple functional stages [37]. Moreover, model-based provisioning is flexible enough to adjust to resource constraints or surpluses exposed during assignment.

This paper is organized as follows. Section 2 motivates the work and summarizes our approach. Section 3 out-

*R. Doyle is also a PhD student at Duke Computer Science.

†O. Asad is currently at Sun Microsystems.

‡This work is supported by the U.S. National Science Foundation through grants CCR-00-82912, ANI-126231, and EIA-9972879, and by IBM, Network Appliance, and HP Laboratories.

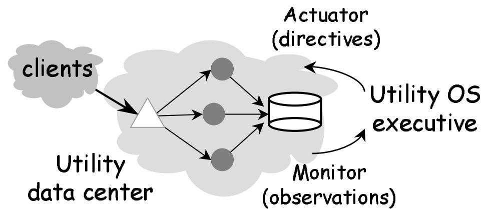


Figure 1: A utility OS. A feedback-controlled policy module or *executive* monitors load and resource status, determines resource slices for each service, and issues directives to configure server shares and direct request traffic to selected servers.

lines simple models for Web services; Section 4 describes a resource allocator based on the models, and demonstrates its behavior in various scenarios. Section 5 describes our prototype and presents experimental results. Section 6 sets our approach in context with related work, and Section 7 concludes.

2 Overview

Our utility provisioning scheme is designed to run within a feedback-controlled resource manager for a server cluster, as depicted in Figure 1. The software that controls the mapping of workloads to servers and storage units is a *utility operating system*; it supplements the operating systems on the individual servers by coordinating traditional OS functions (e.g. resource management and isolation) across the utility as a whole. An executive policy component continuously adjusts resource slices based on smoothed, filtered observations of traffic and resource conditions, as in our previous work on the Muse system [12]. The utility enforces slices by assigning complete servers to services, or by partitioning the resources of a physical server using a resource control mechanism such as resource containers [9] or a virtual machine monitor [36]. Reconfigurable redirecting switches direct request traffic toward the assigned servers for each service.

This paper deals with the executive's policies for provisioning resource slices in this framework. Several factors make it difficult to coordinate provisioning for multiple interacting resources:

- **Bottleneck behavior.** Changing the allotment of resources other than the primary bottleneck has little or no effect. On the other hand, changes at a bottleneck affect demand for other resources.
- **Global constraints.** Allocating resources under constraint is a zero-sum game. Even when data centers are well-provisioned, a utility OS must cope with constraints when they occur, e.g., due to abnormal load surges or failures. Memory is often constrained; in the general case there is not enough to cache the entire data set for every service. Memory management can dramatically affect service performance [26].
- **Local constraints.** Assigning service components (*capsules*) to specific nodes leads to a bin-packing problem [3, 34]. Solutions may expose local resource constraints or induce workload interference.
- **Caching.** Sizing of a memory cache in one component or stage can affect the load on other components. For example, the OS may overcome a local constraint at a shared storage unit by increasing server memory allotment for one capsule's I/O cache, freeing up storage throughput for use by another capsule.

The premise of MBRP is that network service loads have common properties that allow the utility OS to predict their behavior. In particular, service loads are streams of requests with stable average-case behavior; the model allows the system to adapt to changing resource demands at each stage by continuously feeding observed request arrival rates to the models to predict resource demands at each stage. Moreover, the models enable the system to account for resource interactions in a comprehensive way, by predicting the effects of planned resource allotments and placement choices. For example, the models can answer questions like: "how much memory is needed to reduce this service's storage access rate by 20%?". Figure 2 depicts the use of the models within the utility OS executive.

MBRP is a departure from traditional resource management using reactive heuristics with limited assumptions about application behavior. Our premise is that MBRP is appropriate for a utility OS because it hosts a smaller number of distinct applications that are both heavily resource-intensive and more predictable in their average per-request resource demands. In many cases the workloads and service behavior have been intensively studied. We propose to use the resulting models to enable dynamic, adaptive, automated resource management.

For example, we show that MBRP can provision for average-case response time targets for Web services with static content under dynamically varying load. First, the system uses the models to generate initial *candidate* resource allotments that it predicts will meet response time targets for each service. Next, an *assignment* phase maps service components to specific servers and storage units in the data center, balancing affinity, migration cost, competing workloads, and local constraints on individual servers and storage units. The system may

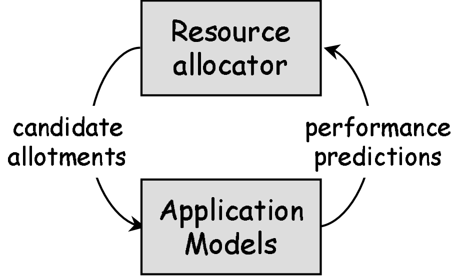


Figure 2: Using models to evaluate candidate resource allotments in a utility OS. The executive refines the allotments until it identifies a resource assignment that meets system goals.

adjust candidate allotments to compensate for local constraints or resource surpluses discovered during the assignment phase. In this context, MBRP meets the following goals in an elegant and natural way:

- **Differentiated service quality**. Since the system can predict the effects of candidate allotments on service quality, it can plan efficient allotments to meet response time targets, favoring services with higher value or more stringent SLAs.
- **Resource management for diverse workloads**. Each service has a distinct *profile* for reference locality and the CPU-intensity of requests. The model parameters capture these properties, enabling the executive to select allotments to meet performance goals. These may include allocating surplus resources to optimize global metrics (e.g., global average response time, yield, or profit).
- **Storage-aware caching**. Storage units vary in their performance due to differences in their configurations or assigned loads. Models capture these differences, enabling the resource scheduler to compensate by assigning different amounts of memory to reduce the dependence on storage where appropriate. Recent work [16] proposed reactive kernel heuristics with similar goals.
- **On-line capacity planning**. The models can determine aggregate resource requirements for all hosted services at observed or possible load levels. This is useful for admission control or to vary hosting capacity with offered load.

This flexibility in meeting diverse goals makes MBRP a powerful basis for proactive resource management in utilities.

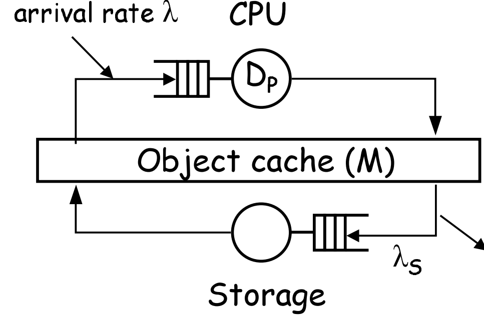


Figure 3: A simple model for serving static Web content. Requests arrive at rate λ (which varies with time) and incur an average service demand D_P in a CPU. An in-memory cache of size M absorbs a portion H of the requests as hits; the misses generate requests to storage at rate λ_S.

Parameter	*Meaning*
α	Zipf locality parameter
λ	Offered load in requests/s
S	Average object size
T	Total number of objects
M	Memory size for object cache
D_P	Average per-request CPU demand
μ_S, ϕ	Peak storage throughput in IOPS

Table 1: Input parameters for Web service models.

3 Web Service Models

This section presents simplified analytical models to predict performance of Web services with static content, based on the system model depicted in Figure 3. Table 1 summarizes the model parameters and other inputs. Table 2 summarizes the model outputs, which guide the choices of the resource allocator described in Section 4.

The models are derived from basic queuing theory and recent work on performance modeling for Web services. They focus on average-case behavior and abstract away much detail. For example, they assume that the network within the data center is well-provisioned. Each of the models could be extended to improve its accuracy; what is important is the illustration they provide of the potential for model-based resource provisioning.

3.1 Server Memory

Many studies indicate that requests to static Web objects follow a Zipf-like popularity distribution [11, 38]. The probability p_x of a request to the xth most popular object is proportional to $1/x^\alpha$, for some parameter α. Many requests target the most popular objects, but the distribution has a heavy tail of unpopular objects with poor reference locality. Higher α values increase the concentra-

Parameter	*Meaning*
R_P	CPU response time
H	Object cache hit ratio
λ_S	Storage request load in IOPS
R_S	Average storage response time
R	Average total response time

Table 2: Performance measures predicted by Web models.

tion of requests on the most popular objects. We assume that object size is independent of popularity [11], and that size distributions are stable for each service [35].

Given these assumptions, a utility OS can estimate hit ratio for a memory size M from two parameters: α, and T, the total size of the service's data (consider M and T to be in units of objects). If the server effectively caches the most popular objects (i.e., assuming perfect Least Frequently Used or LFU replacement), and ignoring object updates, the predicted object hit ratio H is given by the portion of the Zipf-distributed probability mass that is concentrated in the M most popular objects. We can closely approximate this H by integrating over a continuous form of the Zipf probability distribution function [11, 38]. The closed form solution reduces to:

$$H = \frac{1 - M^{1-\alpha}}{1 - T^{1-\alpha}} \quad (1)$$

Zipf distributions appear to be common in a large number of settings, so this model is more generally applicable. While pure LFU replacement is difficult to implement, a large body of Web caching research has yielded replacement algorithms that approximate LFU; even poor schemes such as LRU are qualitatively similar in their behavior.

3.2 Storage I/O Rate

Given an estimate of cache hit ratio H, the service's storage I/O load is easily derived: for a Web load of λ requests per second, the I/O throughput demand λ_S for storage (in IOPS, or I/O operations per second) is:

$$\lambda_S = \lambda S(1 - H) \quad (2)$$

The average object size S is given as the number of I/O operations to access an object on a cache miss.

3.3 Storage Response Time

To determine the performance impact of disk I/O, we must estimate the average response time R_S from storage. We model each storage server as a simple queuing

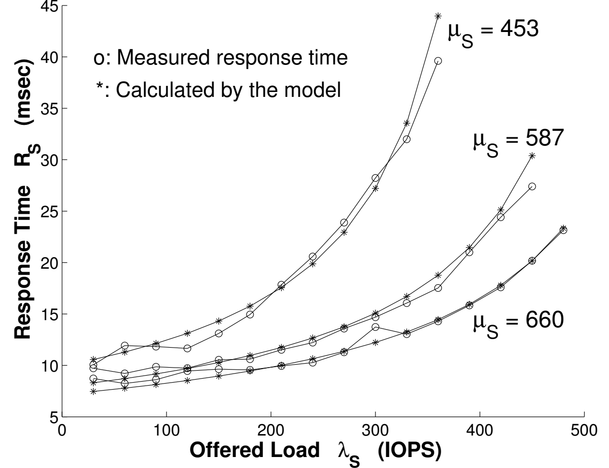


Figure 4: Predicted and observed R_S for an NFS server (Dell 4400, $k = 4$ Seagate Cheetah disks, 256MB, BSD/UFS) under three synthetic file access workloads with different fileset sizes.

center, parameterized for k disks; to simplify the presentation, we assume that the storage servers in the utility are physically homogeneous. Given a stable request mix, we estimate the utilization of a storage server at a given IOPS request load λ_S as λ_S/μ_S, where μ_S is its saturation throughput in IOPS for that service's file set and workload profile. We determine μ_S empirically for a given request mix. A well-known response time formula then predicts the average storage response time as:

$$R_S = \frac{k/\mu_S}{1 - (\lambda_S/\mu_S)} \quad (3)$$

This model does not reflect variations in cache behavior within the storage server, e.g., from changes to M or λ_S. Given the heavy-tailed nature of Zipf distributions, Web server miss streams tend to show poor locality when the Web server cache (M) is reasonably provisioned [14].

Figure 4 shows that this model closely approximates storage server behavior at typical utilization levels and under low-locality synthetic file access loads (random reads) with three different fileset sizes (T). We used the FreeBSD Concatenated Disk Driver (CCD) to stripe the filesets across $k = 4$ disks, and the *fstress* tool [2] to generate the workloads and measure R_S.

Since storage servers may be shared, we extend the model to predict R_S when the service receives an allotment ϕ of storage server resources, representing the maximum storage throughput in IOPS available to the service within its share.

$$R_S = \frac{k/\phi}{1 - (\lambda_S/\phi)} \quad (4)$$

This formula assumes that some scheduler enforces proportional shares ϕ/μ_S for storage. Our prototype uses *Request Windows* [18] to approximate proportional sharing. Other approaches to differentiated scheduling for storage [23, 33] are also applicable.

3.4 Service Response Time

We combine these models to predict average total response time R for the service, deliberately ignoring congestion on the network paths to the clients (which the utility OS cannot control). Given a measure D_P of average per-request service demand on the Web server CPU, CPU response time R_P is given by a simple queuing model similar to the storage model above; previous work [12] illustrates use of such a model to adaptively provision CPU resources for Web services. The service's average response time R is simply:

$$R = R_P + R_S(1 - H) \quad (5)$$

This ignores the CPU cost of I/O and the effects of prefetching for large files. The CPU and storage models already account (crudely) for these factors in the average case.

3.5 Discussion

These models are cheap to evaluate and they capture the key behaviors that determine application performance. They were originally developed to improve understanding of service behavior and to aid in static design of server infrastructures; since they predict how resource demands change as a function of offered load, they can also act as a basis for dynamic provisioning in a shared hosting utility. A key limitation is that the models assume a stable average-case per-request behavior, and they predict only average-case performance. For example, the models here are not sufficient to provision for probabilistic performance guarantees. Also, since they do not account for interference among workloads using shared resources, MBRP depends on performance isolation mechanisms (e.g., [9, 36]) that limit this interference. Finally, the models do not capture overload pathologies [37]; MBRP must assign sufficient resources to avoid overload, or use dynamic admission control to prevent it.

Importantly, the models are independent of the MBRP framework itself, so it is possible to replace them with more powerful models or extend the approach to a wider range of services. For example, it is easy to model simple dynamic content services with a stable average-case service demand for CPU and memory. I/O patterns for database-driven services are more difficult to model, but a sufficient volume of requests will likely reflect a stable average-case behavior.

MBRP must parameterize the models for each service with the inputs from Table 1. T and S parameters and average-case service demands are readily obtainable (e.g., as in Muse [12] or Neptune [32]), but it is an open question how to obtain α and μ_S from dynamic observations. The system can detect anomalies by comparing observed behavior to the model predictions, but MBRP is "brittle" unless it can adapt or reparameterize the models when anomalies occur.

4 A Model-Based Allocator

This section outlines a resource provisioning algorithm that plans least-cost resource slices based on the models from Section 3. The utility OS executive periodically invokes it to adjust the allotments, e.g., based on filtered load and performance observations. The output is an *allotment vector* for each service, representing a CPU share together with memory and storage allotments $[M, \phi]$. The provisioning algorithm comprises three primitives designed to act in concert with an assignment planner, which maps the allotted shares onto specific servers and storage units within the utility. The resource provisioning primitives are as follows:

- *Candidate* plans initial candidate allotment vectors that it predicts will meet SLA response time targets for each service at its load λ.
- *LocalAdjust* modifies a candidate vector to adapt to a local resource constraint or surplus exposed during assignment. For example, the assignment planner might place some service on a server with insufficient memory to supply the candidate M; *LocalAdjust* constructs an alternative vector that meets the targets within the resource constraint, e.g., by increasing ϕ to reduce R_S.
- *GroupAdjust* modifies a set of candidate vectors to adapt to a resource constraint or surplus exposed during assignment. It determines how to reduce or augment the allotments to optimize a global metric, e.g., to minimize predicted average response time. For example, the assignment planner might assign multiple hosted services to share a network storage unit; if the unit is not powerful enough to meet the aggregate resource demand, then *GroupAdjust* modifies each vector to conform to the constraint.

We have implemented these primitives in a prototype executive for a utility OS. The following subsections discuss each of these primitives in turn.

4.1 Generating Initial Candidates

To avoid searching a complex space of resource combinations to achieve a given performance goal, *Candidate* follows a simple principle: *build a balanced system*. The allocator configures CPU and storage throughput (ϕ) allotments around a predetermined average utilization level ρ_{target}. The ρ_{target} may be set in a "sweet spot" range from 50-80% that balances efficient use of storage and CPU resources against queuing delays incurred at servers and storage units. The value for ρ_{target} is a separate policy choice. Lower values for ρ_{target} provide more "headroom" to absorb transient load bursts for the service, reducing the probability of violating SLA targets. The algorithm generalizes to independent ρ_{target} values for each $(service, resource)$ pair. ,

The *Candidate* algorithm consists of the following steps:

- **Step 1**. Predict CPU response time R_P at the configured ρ_{target}, as described in Section 3.4. Select initial $\phi = \mu_S$.
- **Step 2.** Using ϕ and ρ_{target}, predict storage response time R_S using Equation (4). Note that the allocator does not know λ_S at this stage, but it is not necessary because R_S depends only on ϕ and the ratio of λ_S/ϕ, which is given by ρ_{target}.
- **Step 3.** Determine the required server memory hit ratio (H) to reach the SLA target response time R, using Equation (5) and solving for H as:

$$H = 1 - \frac{R - R_P}{R_S} \tag{6}$$

- **Step 4.** Determine the storage arrival rate λ_S from λ and H, using Equation (2). Determine and assign the resulting candidate storage throughput allotment $\phi = \lambda_S/\rho_{target}$.
- **Step 5.** Loop to step 2 to recompute storage response time R_S with the new value of ϕ. Loop until the difference of ϕ values is within a preconfigured percentage of μ_S.
- **Step 6.** Determine and assign the memory allotment M necessary to achieve the hit ratio H, using Equation (1).

Note that the candidate M is the minimum required to meet the response time target. Given reasonable targets, *Candidate* leaves memory undercommitted. To illustrate, Figure 5 shows candidate storage and memory allotments $[M, \phi]$ for an example service as offered Web load λ increases along the x-axis. *Candidate* responds to increasing load by increasing ϕ rather than M. This is because increasing λ does not require a higher hit ratio H to meet a fixed response time target. For a fixed M and corresponding H, λ_S grows linearly with λ, and so the storage allotment ϕ also grows linearly following $\lambda_S/\phi = \rho_{target}$.

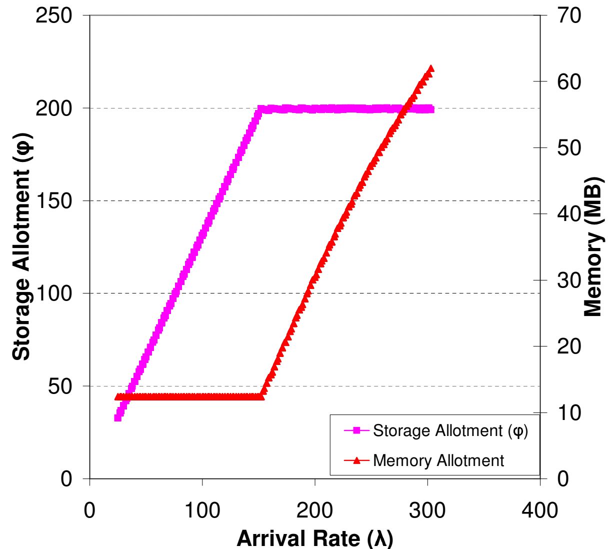
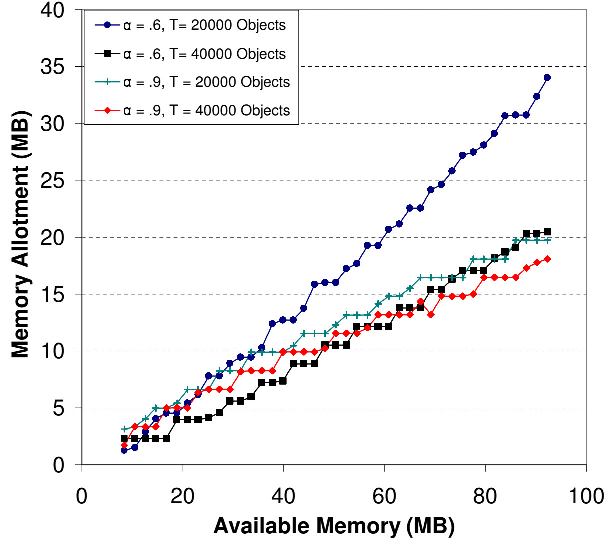


Figure 5: Using *Candidate* and *LocalAdjust* to determine memory and storage allotments for a service. As service load (λ) increases, *Candidate* increases the storage share ϕ to meet response time targets. After encountering a resource constraint at $\phi = 200$ IOPS, *LocalAdjust* transforms the candidate allotments to stay on target by adding memory instead.

Figure 5 also shows how the provisioning scheme adjusts the vectors to conform to a resource constraint. This example constrains ϕ to 200 IOPS, ultimately forcing the system to meet its targets by increasing the candidate M. *Candidate* itself does not consider resource constraints; the next two primitives adapt allotment vectors on a local or group basis to conform to resource constraints, or to allocate surplus resources to improve performance according to system-wide metrics.

4.2 Local Constraint or Surplus

The input to *LocalAdjust* is a candidate allotment vector and request arrival rate λ, together with specified constraints on each resource. The output of *LocalAdjust* is an adjusted vector that achieves a predicted average-case response time as close as possible to the target, while conforming to the resource constraints. Since this paper focuses primarily on memory and storage resources, we ignore CPU constraints. Specifically, we assume that the expected CPU response time R_P for a given ρ_{target}



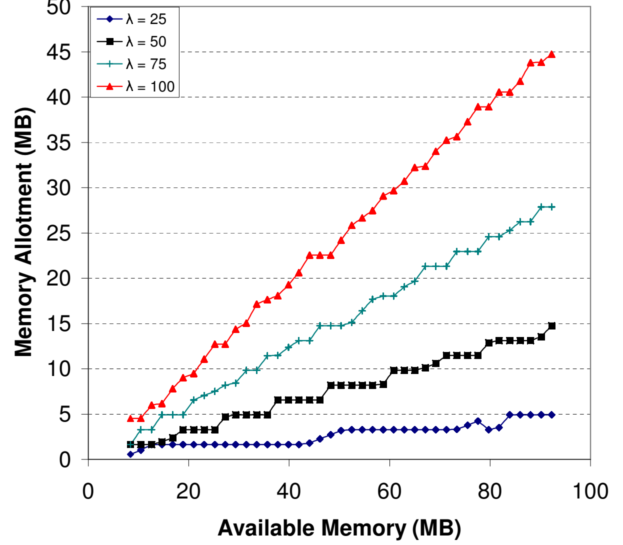


Figure 6: Memory allotments by *GroupAdjust* for four competing services with different caching profiles (left) and different request arrival rates (right).

is fixed and achievable. CPU allotments are relatively straightforward because memory and storage allotments affect per-request CPU demand only minimally. For example, if the CPU is the bottleneck, then the allotments for other resources are easy to determine: set λ to the saturation request throughput rate, and provision other resources as before.

If the storage constraint falls below the candidate storage allotment ϕ, then *LocalAdjust* assigns the maximum value to ϕ, and rebalances the system by expanding the memory allotment M to meet the response time target given the lower allowable request rate λ_S for storage. Determine the allowable $\lambda_S = \phi\rho_{target}$ at the preconfigured ρ_{target}. Determine the hit ratio H needed to achieve this λ_S using Equation (2), and the memory allotment M to achieve H using Equation (1).

Figure 5 illustrates the effect of *LocalAdjust* on the candidate vector under a storage constraint at $\phi = 200$ IOPS. As load λ increases, *LocalAdjust* meets the response time target by holding ϕ to the maximum and growing M instead. The candidate M varies in a (slightly) nonlinear fashion because H grows as M increases, so larger shares of the increases to λ are absorbed by the cache. This effect is partially offset by the dynamics of Web content caching captured in Equation (1): due to the nature of Zipf distributions, H grows logarithmically with M, requiring larger marginal increases to M to effect the same improvement in H.

If memory is constrained, *LocalAdjust* sets M to the maximum and rebalances the system by expanding ϕ (if possible). The algorithm is as follows: determine H and λ_S at M using Equations (1) and (2), and use λ_S to determine the adjusted storage allotment as $\phi = \lambda_S/\rho_{target}$. Then compensate for the reduced H by increasing ϕ further to reduce storage utilization levels below ρ_{target}, improving the storage response time R_S.

If both ϕ and M are constrained, assign both to their maximum values and report the predicted response time using the models in the obvious fashion. *LocalAdjust* adjusts allotments to consume a local surplus in the same way. This may be useful if the service is the only load component assigned to some server or set of servers. Surplus assignment is more interesting when the system must distribute resources among multiple competing services, as described below.

4.3 Group Constraint or Surplus

GroupAdjust adjusts a set of candidate allotment vectors to consume a specified amount of memory and/or storage resource. *GroupAdjust* may adapt the vectors to conform to resource constraints or to allocate a resource surplus to meet system-wide goals.

For example, *GroupAdjust* can reprovision available memory to maximize hit ratio across a group of hosted services. This is an effective way to allocate surplus memory to optimize global response time, to degrade service fairly when faced with a memory constraint, or to reduce storage loads in order to adapt to a storage constraint. Note that invoking *GroupAdjust* to adapt to constraints on specific shared storage units is an instance of storage-aware caching [16], achieved naturally as a side effect of model-based resource provisioning.

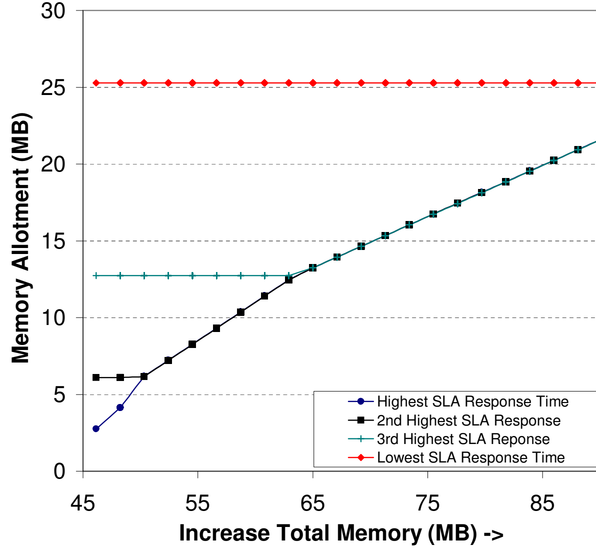


Figure 7: Using *Candidate* and *GroupAdjust* to allocate memory to competing services; in this example, the services are identical except for differing SLA response time targets. *Candidate* determines the minimum memory to meet each service's target; *GroupAdjust* allocates any surplus to the services with the least memory.

GroupAdjust assigns available memory to maximize hit ratio across a group as follows. First, multiply Equation (1) for each service by its expected request arrival rate λ. This gives a weighted value of service hit rates (hits or bytes per unit of time) for each service, as a function of its memory allotment M. The derivative of this function gives the marginal benefit for allocating a unit of memory to each service at its current allotment M. A closed form solution is readily available, since Equation (1) was obtained by integrating over the Zipf probability distribution (see Section 3.1). *GroupAdjust* uses a gradient-climbing approach (based on the Muse MSRP algorithm [12]) to assign each unit of memory to the service with the highest marginal benefit at its current M. This algorithm is efficient: it runs in time proportional to the product of the number of candidate vectors to adjust and the number of memory units to allocate.

To illustrate, Figure 6 shows the memory allotments of *GroupAdjust* to four competing services with different cache behavior profiles (α, S, T), as available memory increases on the x-axis. The left-hand graph varies the α and T parameters, holding offered load λ constant across all services. Services with higher α concentrate more of their references on the most popular objects, improving cache effectiveness for small M, while services with lower T can cache a larger share of their objects with a given M. The graph shows that for services with the same α values, *GroupAdjust* prefers to allocate memory to services with lower T. In the low-memory cases, it prefers higher-locality services (higher α); as total memory increases and the most popular objects of higher α services fit in cache, the highest marginal benefit for added memory moves to lower α services. These properties of Web service cache behavior result in the crossover points for services with differing α values.

The right-hand graph in Figure 6 shows four services with equal α and T but varying offered load λ. Higher λ increases the marginal benefit of adding memory for a service, because a larger share of requests go to that service's objects. *GroupAdjust* effectively balances these competing demands. When the goal is to maximize global hit ratio, as in this example, a global perfect-LFU policy in the Web servers would ultimately yield the same M shares devoted to each service. Our model-based partitioning scheme produces the same result as the reactive policy (if the models are accurate), but it also accommodates other system goals in a unified way. For example, *GroupAdjust* can use the models to optimize for global response time in a storage-aware fashion; it determines the marginal benefit in response time for each service as a function of its M, factoring in the reduced load on shared storage. Similarly, it can account for service priority by optimizing for "utility" [12, 21] or "yield" [32], composing the response time function for each service with a yield function specifying the value for each level of service quality.

4.4 Putting It All Together

Candidate and *GroupAdjust* work together to combine differentiated service with other system-wide performance goals. Figure 7 illustrates how they allocate surplus memory to optimize global response time for four example services. In this example, the hosted services have identical λ and caching profiles (α, S, T), but their SLAs specify different response time targets. *Candidate* allocates the minimum memory (total 46 MB) to meet the targets, assigning more memory to services with more stringent targets. *GroupAdjust* then allocates surplus memory to the services that offer the best marginal improvement in overall response time. Since the services have equivalent behavior, the surplus goes to the services with the smallest allotments.

Figure 8 further illustrates the flexibility of MBRP to adapt to observed changes in load or system behavior. It shows allotments $[M, \phi]$ for three competing services **s0**, **s1**, and **s2**. The system is configured to consolidate all loads on a shared server and storage unit, meet minimal response time targets when possible, and use any surplus resources to optimize global response time. The services begin with identical arrival rates λ, caching profiles (α, S, T), and response time targets. The experi-

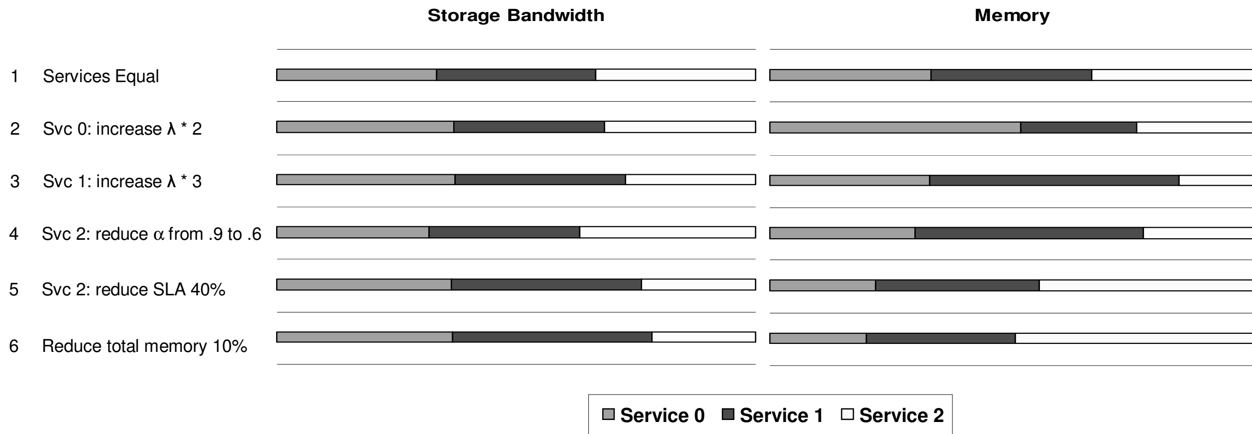


Figure 8: Allotments $[M, \phi]$ for three competing services in a multiple step experiment. For each step, the graphs represent the allotment percentage of each resource received by each service.

ment modifies a single parameter in each of six steps and shows its effect on the allotments; the changes through the steps are cumulative.

- In the base case, all services receive equal shares.
- Step 2 doubles λ for **s0**. The system shifts memory to **s0** to reflect its higher share of the request load, and increases its ϕ to rebalance storage utilization to ρ_{target}, compensating for a higher storage load.
- Step 3 triples the arrival rate for **s1**. The system shifts memory to **s1**; note that the memory shares match each service's share of the total request load. The system also rebalances storage by growing **s1**'s share at the expense of the lightly loaded **s2**.
- Step 4 reduces the cache locality of **s2** by reducing its α from .9 to .6. This forces the system to shift resources to **s2** to meet its response time target, at the cost of increasing global average response time.
- Step 5 lowers the response time target for **s2** by 40%. The system further compromises its global response time goal by shifting even more memory to **s2** to meet its more stringent target. The additional memory reduces the storage load for **s2**, allowing the system to shift some storage resource to the more heavily loaded services.
- The final step in this experiment reduces the amount of system memory by 10%. Since **s2** holds the minimum memory needed to meet its target, the system steals memory from **s0** and **s1**, and rebalances storage by increasing **s1**'s share slightly.

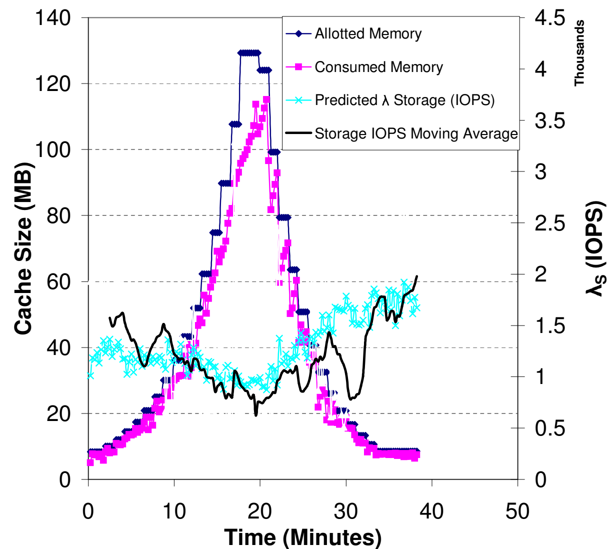


Figure 9: Predicted and observed storage I/O request rate vary with changing memory allotment for a Dash server under a typical static Web load (a segment of a 2001 trace of *www.ibm.com*).

5 Prototype and Results

To evaluate the MBRP approach, we prototyped key components of a Web service utility (as depicted in Figure 1) and conducted initial experiments using Web traces and synthetic loads. The cluster testbed consists of load generating clients, a reconfigurable L4 redirecting switch (from [12]), Web servers, and network storage servers accessed using the Direct Access File System protocol (DAFS [13, 24]), an emerging standard for network storage in the data center. We use the DAFS implementation from [24] over an Emulex cLAN network.

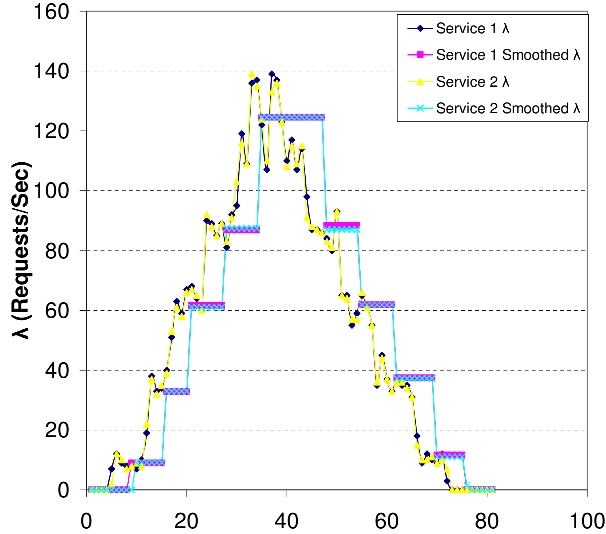


Figure 10: Arrival rate λ for two services handling synthetic load swells under the Dash Web server.

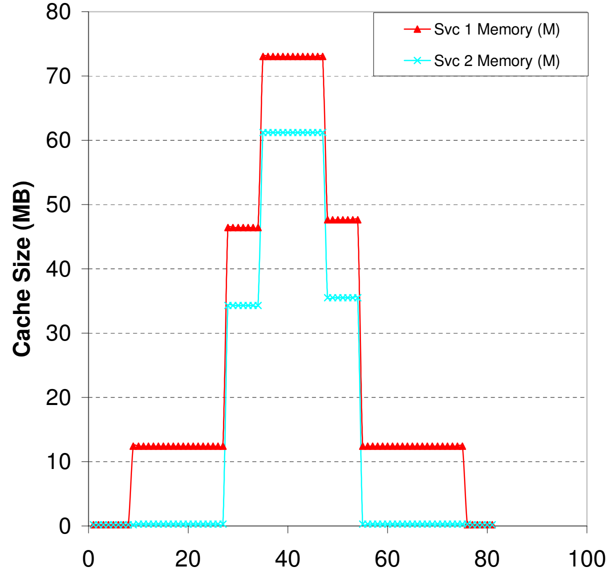


Figure 11: Memory allotments M for two services handling synthetic load swells under the Dash Web server. As load increases, *LocalAdjust* provisions additional memory to the services to keep the response time within SLA limits. Service 1 is characterized by higher storage access costs and therefore receives more memory to compensate.

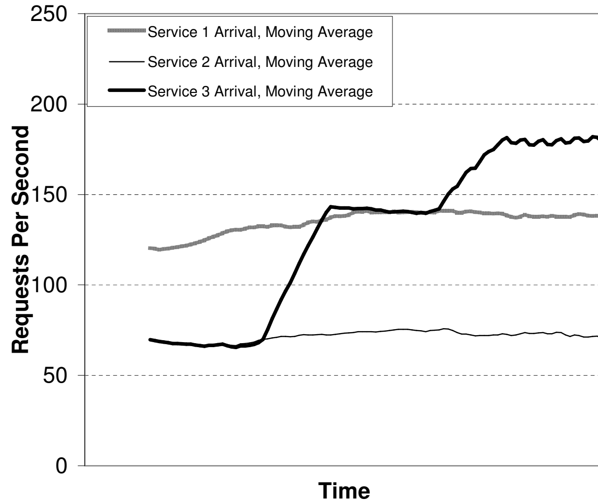
The prototype utility OS executive coordinates resource allocation as described in Section 4. It periodically observes request arrival rates (λ) and updates resource slices to adapt to changing conditions. The executive implements its actions through two mechanisms. First, it issues directives to the switch to configure the active server sets for each hosted service; the switch distributes incoming requests for each service evenly across its active set. Second, it controls the resource shares allocated to each service on each Web server.

To allow external resource control, our prototype uses a new Web server that we call Dash [8]. Dash acts as a trusted component of the utility OS; it provides a protected, resource-managed execution context for services, and exports powerful resource control and monitoring interfaces to the executive. Dash incorporates a DAFS user-level file system client, which enables user-level resource management in the spirit of Exokernel [19], including full control over file caching and and data movement [24]. DAFS supports fully asynchronous access to network storage, enabling a single-threaded, event-driven Web server structure as proposed in the Flash Web server work [27]—hence the name Dash. In addition, Dash implements a decentralized admission control scheme called Request Windows [18] that approximates proportional sharing of storage server throughput. The details and full evaluation of Dash and Request Windows are outside the scope of this paper.

For our experiments, the Dash and DAFS servers run on SuperMicro SuperServer 6010Hs with 866 MHz Pentium-III Xeon CPUs; the DAFS servers use one 9.1 GB 10,000 RPM Seagate Cheetah drive. Dash controls memory usage as reported in the experiments. Web traffic originates from a synthetic load generator ([10]) or Web trace replay as reported; the caching profiles (α, S, T) are known a priori and used to parameterize the models. All machines run FreeBSD 4.4.

We first present a simple experiment to illustrate the Dash resource control and to validate the hit ratio model (Equation (2)). Figure 9 shows the predicted and observed storage request rate λ_S in IOPS as the service's memory allotment M varies. The Web load is an accelerated 40-minute segment of a 2001 IBM trace [12] with steadily increasing request rate λ. Larger M improves the hit ratio for the Dash server cache; this tends to reduce λ_S, although λ_S reflects changes in λ as well as hit ratio. The predicted λ_S approximates the observed I/O load; the dip at $t = 30$ minutes is due to a transient increase in request locality, causing an unpredicted transient improvement in cache hit ratio. Although the models tend to be conservative in this example, the experiment demonstrates the need for a safety margin to protect against transient deviations from predicted behavior.

To illustrate the system's dynamic behavior in storage-aware provisioning, we conducted an experiment with two services with identical caching profiles (α, S, T) and response time targets, serving identical synthetic load swells on a Dash server. The peak IOPS through-



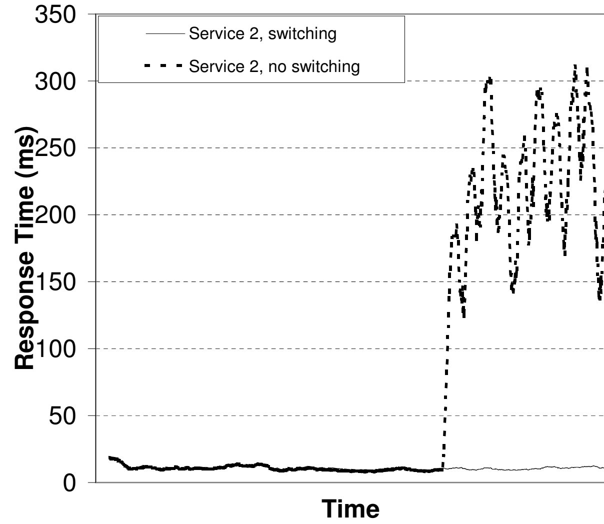


Figure 13: Arrival rates for competing services (left) and client response time with and without a bin-packing assignment phase to switch Web servers (right) handling the service.

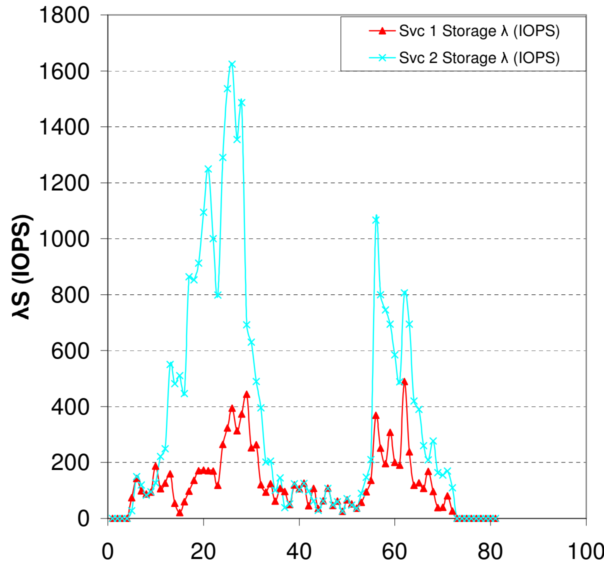


Figure 12: I/O rate (λ_S) for two services handling synthetic load swells under the Dash Web server. As additional memory is allocated to the services to meet SLA targets, the storage arrival rate decreases. Service 1 storage load reduces at a greater rate due to the additional memory allocated to this service.

puts available at the storage server for each service (reflected in the μ_s parameters) are constrained at different levels, with a more severe constraint for service 1. Figure 10 shows the arrival rates λ and the values smoothed by a "flop-flip" stepped filter [12] for input to the executive. Figure 11 shows the memory allotments for each service during the experiments, and Figure 12 shows the resulting storage loads λ_S. The storage constraints force the system to assign each service more memory to meet its target; as load increases, it allocates proportionally more memory to service 1 because it requires a higher H to meet the same target. As a result, service 1 shows a lower I/O load on its more constrained storage server. This is an example of how the model-based provisioning policies (here embodied in *LocalAdjust*) achieve similar goals to storage-aware caching [16].

The last experiment uses a rudimentary assignment planner to illustrate the role of assignment in partitioning cluster resources for response time targets. We compared two runs of three services on two Dash servers under the synthetic loads shown on the left-hand side of Figure 13, which shows a saturating load spike for service 3. In the first run, service 1 is bound to server A and services 2 and 3 are bound to server B. This results in a response time jump for service 2, shown in the right-hand graph in Figure 13; since the system cannot meet targets for both services, it uses *GroupAdjust* to provision B's resources for the best average-case response time. The second run employs a simple bin-packing scheme to assign the provisioned resource slices to servers. In this run, the system reassigns service 2 to A when the load spike for service 3 exposes the local resource constraint on B; this is possible because *Candidate* determines that there are sufficient resources on A to meet the response time targets for both services 1 and 2. To implement this choice, the executive directs the switch to route requests for service 2 to A rather than B. This allows service 2 to continue meeting its target. This simple example shows the power of the model-based provisioning primitives as a foundation for comprehensive resource management for cluster utilities.

6 Related Work

Adaptive resource management for servers. Aron [6] uses kernel-based feedback schedulers to meet latency and throughput goals for network services running on a single shared server. Abdelzaher et. al. [1] have addressed the control-theoretical aspects of feedback-controlled Web server performance management.

SEDA [37] reactively provisions CPU resources (by varying the number of threads) across multiple service stages within a single server. SEDA emphasizes request admission control to degrade gracefully in overload. A utility can avoid overload by expanding resource slices and recruiting additional servers as traffic increases. SEDA does not address performance isolation for shared servers.

Resource management for cluster utilities. Several studies use workload profiling to estimate the resource savings of multiplexing workloads in a shared utility. They focus on the probability of exceeding performance requirements for various degrees of CPU overbooking [4, 6, 34]. Our approach varies the degree of overbooking to adapt to load changes, but our current models consider only average-case service quality within each interval. The target utilization parameters (ρ_{target}) allow an external policy to control the "headroom" to handle predicted load bursts with low probability of violating SLA targets.

There is a growing body of work on adaptive resource management for cluster utilities under time-varying load. Our work builds on Muse [12], which uses a feedback-controlled policy manager and redirecting switch to partition cluster resources; [39] describes a similar system that includes a priority-based admission control scheme to limit load. Levy [21] presents an enhanced framework for dynamic SOAP Web services, based on a flexible combining function to optimize configurable class-specific and cluster-specific objectives. Liu [22] proposes provisioning policies (SLAP) for e-commerce traffic in a Web utility, and focuses on maximizing SLA profits. These systems incorporate analytical models of CPU behavior; MBRP extends them to provision for multiple resources including cluster memory and storage throughput.

Several of these policy-based systems rely on resource control mechanisms—such as Resource Containers [9] or VMware ESX [36]— to allow performance-isolated server consolidation. Cluster Reserves [7] extends a server resource control mechanism (Resource Containers) to a cluster. These mechanisms are designed to enforce a provisioning policy, but they do not define the policy. Cluster Reserves adjusts shares on individual servers to bound aggregate usage for each hosted service. It is useful for utilities that do not manage request traffic within the cluster. In contrast, our approach uses server-level (rather than cluster-level) resource control mechanisms in concert with redirecting switches.

The more recent Neptune [32] work proposes an alternative to provisioning (or partitioning) cluster resources. Neptune maximizes an abstract SLA yield metric, similarly to other utility resource managers [12, 21, 22]. Like SLAP, each server schedules requests locally to maximize per-request yield. Neptune has no explicit provisioning; it distributes requests for all services evenly across all servers, and relies on local schedulers to maximize global yield. While this approach is simple and fully decentralized, it precludes partitioning the cluster for software heterogeneity [5, 25], memory locality [14, 26], replica control [17, 31], or performance-aware storage placement [3].

Virtual Services [29] proposes managing services with multiple tiers on different servers. This paper shows how MBRP coordinates provisioning of server and storage tiers; we believe that MBRP can extend to multi-tier services.

Memory/storage management. Kelly [20] proposes a Web proxy cache replacement scheme that considers the origin server's value in its eviction choices. Storage-Aware Caching [16] develops kernel-based I/O caching heuristics that favor blocks from slower storage units. MBRP shares the goal of a differentiated caching service, but approaches it by provisioning memory shares based on a predictive model of cache behavior, rather than augmenting the replacement policy. MBRP supports a wide range of system goals in a simple and direct way, but its benefits are limited to applications for which the system has accurate models.

Several systems use application-specific knowledge to manage I/O caching and prefetching. Patterson et. al. [28] uses application knowledge and a cost/benefit algorithm to manage resources in a shared I/O cache and storage system. Faloutsos [15] uses knowledge of database access patterns to predict the marginal benefit of memory to reduce I/O demands for database queries.

Hippodrome [3] automatically assigns workload components to storage units in a utility data center. Our work is complementary and uses a closely related approach. Hippodrome is model-based in that it incorporates detailed models of storage system performance, but it has no model of the applications themselves; in particular, it cannot predict the effect of its choices on application service quality. Hippodrome employs a backtracking assignment planner (Ergastulum); we believe that a similar approach could handle server assignment in a utility data center using our MBRP primitives.

7 Conclusion

Scalable automation of large-scale network services is a key challenge for computing systems research in the next decade. This paper deals with a software framework and policies to manage network services as utilities whose resources are automatically provisioned and sold according to demand, much as electricity is today. This can improve robustness and efficiency of large services by multiplexing shared resources rather than statically overprovisioning each service for its worst case.

The primary contribution of this work is to demonstrate the potential of *model-based resource provisioning* (MBRP) for resource management in a Web hosting utility. Our approach derives from established models of Web service behavior and storage service performance. We show how MBRP policies can adapt to resource availability, request traffic, and service quality targets. These policies leverage the models to predict the performance effects of candidate resource allotments, creating a basis for informed, policy-driven resource allocation. MBRP is a powerful way to deal with complex resource management challenges, but it is only as good as the models. The model-based approach is appropriate for utilities running a small set of large-scale server applications whose resource demands are a function of load and stable, observable characteristics.

Acknowledgments

Darrell Anderson and Sara Sprenkle contributed to the ideas in this paper, and assisted with some experiments. We thank Mike Spreitzer, Laura Grit, and the anonymous reviewers for their comments, which helped to improve the paper.

References

[1] T. F. Abdelzaher, K. G. Shin, and N. Bhatti. Performance guarantees for Web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, June 2001.

[2] D. C. Anderson and J. S. Chase. Fstress: A flexible network file service benchmark. Technical Report CS-2002-01, Duke University, Department of Computer Science, January 2002.

[3] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: running circles around storage administration. In *Proceedings of the First Usenix Conference on File and Storage Technologies (FAST)*, January 2002.

[4] A. Andrzejak, M. Arlitt, and J. Rolia. Bounding the resource savings of utility computing models. Technical Report HPL-2002-339, HP Labs, November 2002.

[5] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. Pazel, J. Pershing, and B. 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] M. Aron. *Differentiated and Predictable Quality of Service in Web Server Systems*. PhD thesis, Department of Computer Science, Rice University, October 2000.

[7] M. Aron, P. Druschel, and W. 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.

[8] O. M. Asad. Dash: A direct-access Web server with dynamic resource provisioning. Master's thesis, Duke University, Department of Computer Science, December 2002.

[9] G. Banga, P. Druschel, and J. 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.

[10] P. Barford and M. E. Crovella. Generating representative Web workloads for network and server performance evaluation. In *Proceedings of Performance '98/ACM SIGMETRICS '98*, pages 151–160, June 1998.

[11] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of IEEE Infocom '99*, March 1999.

[12] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centers. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP)*, pages 103–116, October 2001.

[13] M. DeBergalis, P. Corbett, S. Kleiman, A. Lent, D. Noveck, T. Talpey, and M. Wittle. The Direct Access File System. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, March 2003.

[14] R. P. Doyle, J. S. Chase, S. Gadde, and A. M. Vahdat. The trickle-down effect: Web caching and server request distribution. *Computer Communications: Selected Papers from the Sixth International Workshop on Web Caching and Content Delivery (WCW)*, 25(4):345–356, March 2002.

[15] C. Faloutsos, R. T. Ng, and T. K. Sellis. Flexible and adaptable buffer management techniques for database management systems. *IEEE Transactions on Computers*, 44(4):546–560, April 1995.

[16] B. C. Forney, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Storage-aware caching: Revisiting caching for heterogeneous storage systems. In *Proceedings of the First Usenix Conference on File and Storage Technologies (FAST)*, January 2002.

[17] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for Internet service construction. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI)*, October 2000.

[18] W. Jin, J. S. Chase, and J. Kaur. Proportional sharing for a storage utility. Technical Report CS-2003-02, Duke University, Department of Computer Science, January 2003.

[19] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on Exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[20] T. P. Kelly, S. Jamin, and J. K. MacKie-Mason. Variable QoS from shared Web caches: User-centered design and value-sensitive replacement. In *Proceedings of the MIT Workshop on Internet Service Quality Economics (ISQE 99)*, December 1999.

[21] R. Levy, J. Nagarajarao, G. Pacifici, M. Spreitzer, A. Tantawi, and A. Youssef. Performance management for cluster-based Web services. In *Proceedings of the 8th International Symposium on Integrated Network Management (IM 2003)*, March 2003.

[22] Z. Liu, M. S. Squillante, and J. L. Wolf. On maximizing service-level-agreement profits. In *Proceedings of the ACM Conference on Electronic Commerce (EC'01)*, October 2001.

[23] C. R. Lumb, A. Merchant, and G. A. Alvarez. Facade: Virtual storage devices with performance guarantees. In *Proceedings of the 2nd Usenix Conference on File and Storage Technologies (FAST)*, March 2003.

[24] K. Magoutis, S. Addetia, A. Fedorova, M. Seltzer, J. Chase, R. Kisley, A. Gallatin, R. Wickremisinghe, and E. Gabber. Structure and performance of the Direct Access File System. In *USENIX Technical Conference*, pages 1–14, June 2002.

[25] J. Moore, D. Irwin, L. Grit, S. Sprenkle, and J. Chase. Managing mixed-use clusters with Cluster-on-Demand. Technical report, Duke University, Department of Computer Science, November 2002.

[26] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenopoel, and E. 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.

[27] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the 1999 Annual Usenix Technical Conference*, June 1999.

[28] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 79–95, Copper Mountain, CO, December 1995. ACM Press.

[29] J. Reumann, A. Mehra, K. G. Shin, and D. Kandlur. Virtual Services: A new abstraction for server consolidation. In *Proceedings of the Usenix 2000 Technical Conference*, June 2000.

[30] J. Rolia, S. Singhal, and R. Friedrich. Adaptive Internet data centers. In *Proceedings of the International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet (SSGRR '00)*, July 2000.

[31] Y. Saito, B. N. Bershad, and H. 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.

[32] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated resource management for cluster-based Internet services. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, December 2002.

[33] P. Shenoy and H. M. Vin. Cello: A disk scheduling framework for next-generation operating systems. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 1998.

[34] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, December 2002.

[35] A. Venkataramani, P. Yalagandula, R. Kokku, S. Sharif, and M. Dahlin. The potential costs and benefits of long-term prefetching for content distribution. *Computer Communications: Selected Papers from the Sixth International Workshop on Web Caching and Content Delivery (WCW)*, 25(4):367–375, March 2002.

[36] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, December 2002.

[37] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proceedings of the Eighteenth Symposium on Operating Systems Principles (SOSP-18)*, Banff, Canada, October 2001.

[38] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. Levy. On the scale and performance of cooperative Web proxy caching. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, December 1999.

[39] H. Zhu, H. Tang, and T. Yang. Demand-driven service differentiation in cluster-based network servers. In *Proceedings of IEEE Infocom 2001*, April 2001.