Online Web Cluster Capacity Estimation and Its Application to Energy Conservation

Chang-Hao Tsai, *Student Member*, *IEEE Computer Society*, Kang G. Shin, *Fellow*, *IEEE*, John Reumann, *Member*, *IEEE*, and Sharad Singhal, *Member*, *IEEE*

Abstract—Designers of data centers and Web servers aim to make on-demand allocation of resources to clients in order to lower the deployment cost of hosted services. Moreover, they must also minimize operating costs, such as energy consumption, by matching service-capacity demand with resource supply. However, since the term "capacity" is typically defined vaguely or inadequately, it is difficult to assess resource needs and, hence, servers, which are several times larger than needed at runtime, are usually deployed. The time-varying nature of the workload model further complicates the problem and necessitates an online capacity-estimation solution. To address this overprovisioning problem, we first define the *capacity* of a server cluster as the sustainable throughput subject to a request retransmission ratio constraint and then analyze different approaches to capacity estimation in a running system. Various capacity-estimation mechanisms, such as offline benchmarking and CPU-utilization evaluation, are discussed and compared with our queue-monitoring method. We employ several different data-collection methods (application instrumentation, user-space tools, Simple Network Management Protocol (SNMP), and kernel modules) to compare their effects on estimation accuracy. Of these, queue monitoring is found to provide a good and stable estimate of server capacity. To validate this finding, we propose a simple cluster-resizing mechanism and evaluate the energy-conservation performance. A good combination of data collection and online capacity estimation is found to make significantly more energy savings than traditional approaches (that is, static estimation and scheduled capacity). Our experimental results show that more than 40 percent of energy can be saved for regular daily usage patterns without any prior knowledge of the workload and that long start-up and shutdown delays affect energy savings considerably.

Index Terms—Server cluster, Web servers and clients, service-capacity estimation and on-demand resource allocation, cluster resizing and energy savings.

1 Introduction

Clustered servers are commonly used to provide highly available and scalable services. The *capacity*, or maximum sustainable throughput, of a server cluster is approximately the sum of all individual servers' capacities. However, each server's capacity is typically unknown, workload dependent, and highly variable. This is due mainly to the unpredictability of resource demands, content change, and the nonlinear scaling within individual servers.

The capacity of a server system is determined by several factors. First, the request arrival process and the target of each individual request determine both instantaneous and average resource requirements. Second, users' performance expectations and patience define acceptable service levels, which, in turn, affect the meaning of the term "sustainable throughput" and, hence, real system capacity. Third, server hardware and software configurations directly influence performance and resource requirements. Furthermore, configured queuing limits on the server affect the maximal surge that can be absorbed by the server. Depending on the request arrival process, this can affect average client-perceived delays because a request packet drop results in a time-out due to slow Transmission Control Protocol (TCP) SYN packet retransmission. All of these interrelated factors make accurate capacity estimation a very difficult problem. These issues have not been fully explored because servers are often tested using relatively predictable workloads (for example, typical Web benchmarks) or measured in terms of long running averages that hide inaccuracies in system capacity estimation.

For the abovementioned reasons, the current engineering approach to capacity estimation is to conservatively underestimate the capacity of each server based on a coarse-grained (weekly or monthly) utilization analysis that describes user demands as well as server performance. Usually, linear scaling is assumed, and then equipment is purchased if the estimated capacity does not allow the safety margin that practitioners value. Thus, servers are almost always severely underutilized. While the excessive equipment purchase may not present a major problem to data centers, excess capacity allocation generally reduces overall system utilization and directly reduces the profit from a given infrastructure. The operational cost of powering and cooling the excess equipment reduces the profit margin even further. Nonetheless, reducing available capacity to lower operational cost is not an option because

- *C.-H. Tsai is with the Real-Time Computing Lab, Department of Electrical Engineering and Computer Science, The University of Michigan, 2260 Hayward Street, Room 4956 CSE, Ann Arbor, MI 48109-2121. E-mail: chtsai@eecs.umich.edu.*
- *K.G. Shin is with the Department of Electrical Engineering and Computer Science, The University of Michigan, 2260 Hayward Street, Ann Arbor, MI 48109-2121. E-mail: kgshin@eecs.umich.edu.*
- *J. Reumann is with Google Inc., 1440 Broadway, 21st Floor, New York, NY 10018. E-mail: reumann@google.com.*
- *S. Singhal is with Hewlett-Packard Laboratories, 1501 Page Mill Road, M/S 1125, Palo Alto, CA 94002. E-mail: sharad.singhal@hp.com.*

Manuscript received 12 June 2005; revised 12 Feb. 2006; accepted 26 June 2006; published online 9 Jan. 2007.
Recommended for acceptance by A. Sivasubramaniam.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0289-0605.
Digital Object Identifier no. 10.1109/TPDS.2007.1028.

this would not allow the system to handle peak loads. Handling peak loads, however, is an important deployment consideration because every lost request translates into lost revenue, which is typically much greater than the marginal cost of energy expenditure. Furthermore, we argue that an application service provider (ASP) also runs the risk of degrading customer satisfaction without appropriate online capacity estimation and may potentially face service outages. The economic reasons for better estimation of capacity are therefore compelling.

We explore the realities, pitfalls, and techniques of online or dynamic capacity estimation as it is an important building block in capacity planning, on-demand computing, and energy conservation in server systems. After observing the correlation between queue length and client-perceived response time, we define "drop ratio" as a measure of user satisfaction and the true "capacity" of server systems. We propose a new scheme based on queue-length monitoring to estimate server capacity and compare it with other simpler estimation mechanisms.

To evaluate our approach, we implement and compare several different measurement and estimation schemes. Our estimation approach is also demonstrated by its application to the well-known energy conservation problem [1], [2]. In this scenario, we resize a cluster of Web servers by utilizing a relatively simple controller to determine the cluster size and dynamically powering them up and down to conserve energy. Our ability to realize substantial (in excess of 40 percent) energy savings demonstrates the accuracy and applicability of our online capacity estimation. We also show that long start-up and shutdown delays limit the performance of energy conservation.

This paper is organized as follows: We first discuss previous research on server clusters and workload analysis in Section 2. The system model and metrics we used are defined in Sections 3 and 4, respectively. We discuss various estimation mechanisms in Section 5, and Section 6 details implementation with a variety of measurement methods. The performance of capacity estimation is evaluated in Section 7, with energy conservation as an application. Section 8 closes the paper with concluding remarks.

2 Related Work

Server clustering is a commonly used method to build high-performance and high-availability services [3], [4]. Optimizations have been introduced in both architectural design [5], [6], [7] and request distribution schemes [8], [9], [10]. Special-purpose clusters, such as streaming media server clusters [11] and the Océano computing utility [12], have also been proposed.

Significant research efforts have been made on characterizing the workload presented to servers, especially Web servers [13], [14], [15], as understanding the user model can facilitate the use of workload generators [16], [17] and benchmark programs [18] to study server performance problems. Changes of the average HTTP response size over time of day suggest that users' browsing behavior (which might be affected by browser's caching) changes over time of day [19]. Recent workload characterization of dynamic content Web sites also indicates changes in the workload model and parameters [20]. Both evidences suggest that online capacity estimation is necessary as the workload changes dynamically.

To manage cluster resources, Cluster Reserves [21] extend Resource Containers [22] to server clusters and achieve service isolation, while services share every server node. However, as server blades provide many small servers to service providers, we argue that estimating server capacity and allocating an appropriate number of blade servers to each single service is a better approach.

Given limited resources in single-server configurations, control theory has also been applied to optimize server performance by tuning system parameters such as MaxClients and KeepAlive in the Apache Web server [23], [24] or throttling the request admission rate in each stage to manage the response time [25]. In the data center scenario, one can dynamically allocate resources to services violating their quality constraints and can satisfy clients easily.

On the other hand, any performance requirement must be able to relate to end-users' perspectives. A response time larger than 10 seconds is shown to cause users to think that there is an error in the system [26], [27]. Several researchers attempted to measure the client-perceived response time. In [28], HTML document instrumentation is used to assess the response time. However, the result does not include a TCP connection setup time and is only applicable to HTTP requests. Alternatively, in [29], TCP client-server interactions, including packet drops, are used to infer the response time. However, as data centers do not have control of network delay or packet loss in the network, using the client-perceived response time directly as a service quality measure for the server is inappropriate and misleading.

Dynamic Voltage Scaling (DVS) [30] is a well-known technique to reduce energy consumption. A real-time extension [31] guarantees service timeliness while saving energy. Using simulation, Bohrer [32] showed that DVS can reduce CPU energy consumption in Web servers by up to 36 percent. Our cluster-resizing mechanism can be combined with DVS to improve energy conservation.

Rajamani and Lefurgy [33] defined system and workload characteristics that could affect energy savings in server clusters and applied simple control mechanisms to demonstrate that having knowledge of these characteristics, which may not always be available, can improve the result of a simple threshold control approach. Pinheiro [34] also applied power management to cluster-based systems, including a Web server cluster. They estimated the resource utilizations on each server and added them up to predict the total resource requirement. They also showed that mismatches between the decision frequency and the workload-varying rate can affect service quality. However, resource demand and utilization typically exhibit a nonlinear relationship, and performance is not directly managed in the control loop. Elnozahy et al. [35] proposed a request batching policy to reduce energy consumption at the cost of a longer response time. It can be combined with DVS and our method to maximize energy savings.

Chase et al. [1] applied an economic approach, called Muse, to manage energy and other resources in data centers. In Muse, the value of each resource is quantified

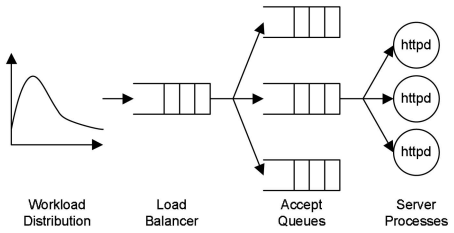

Fig. 1. The system model of a server cluster. In a symmetric configuration, requests are evenly distributed to server processes and queue length can indicate server utilization.

in utility functions and services place bids on resources. They make control decisions by maximizing service revenue and profit, and their results showed that 29 percent or more energy can be saved for a typical workload. However, we argue that resources cannot be priced easily and that the resource price could also be highly time varying. Since resource utilization and performance-resource relationships are difficult to determine, it could make data center operation more complicated and even infeasible. In this paper, we show that similar energy savings can be achieved by a much simpler method.

Chen et al. [2] formulated the cost of operation, including electricity cost and wear-and-tear due to power cycles. With the cost model, they applied both queuing and control-theory-based approaches to minimize the cost while meeting Service Level Agreements (SLAs). Compared to this, our work has put more focus on defining the true service level and the capacity of each server. We presented energy conservation only as an application of accurate capacity estimation. Our definition of service level (that is, request drop rate) has a greater impact on the client-perceived response time and is also clearly defined outside the context of energy or utility cost. Therefore, the estimated capacity can be used in other applications as well.

3 System Model

A typical server cluster consists of a load balancer and a number of servers, as depicted in Fig. 1. Servers are usually always on, but an operator can also take some of them offline for maintenance or energy conservation as in [1] and [2]. Hence, the size of a cluster is defined as the number of active servers in the cluster. The load balancer responds to a single externally accessible Internet Protocol (IP) address and balances the allocation of incoming requests across the servers in the cluster by rewriting the IP addresses [6]. Each individual server usually provides identical services so that client requests can be dispatched to arbitrary back-end servers.

All servers are assumed to have identical hardware and software configurations, thus making service time (nearly) independent of the server used. Servers of different configurations or servers that can be partitioned, such as the IBM Logical Partitioning for zSeries and pSeries servers [36], Sun Dynamic System Domains [37], and HP Partitioning Continuum [38], would only require straightforward extensions to the results presented in this paper and are therefore not specifically addressed here.

The client-server request-reply traffic is assumed to be encapsulated in TCP connections. The load balancer forwards a connection request to one of the servers, which will be responsible for picking up the request. After a TCP connection is established, the client transmits its request and one server process will be awoken to serve it. Depending on the request, the server process may access local files, fetch remote files via Storage Area Networks (SANs), retrieve data from database servers, do computation, and so on. The request arrival process and the implicit resource demand imposed by each request are assumed to follow some probabilistic distribution that is unknown to the server cluster's operator.

Before each request is processed, it may be queued at either the load balancer or one of the real servers. A request would only be queued at the load balancer if the network link toward the chosen real server is occupied by other requests. Since the network traffic volume is usually asymmetric and the network devices can forward at wire speed with very low blocking probabilities, it is highly unlikely that many request packets would be queued at the load balancer unless the load balancer is configured to operate in proxy mode. Therefore, we assume that the load balancer does not drop any request and that its queue length is ignored in our model.

On the other hand, a request is always queued at the server before a process can pick it up due to the design of TCP. The server queue is also referred to as *backlog* in most UNIX systems. A server will drop a request if the queue is full, and the request will be retransmitted after a time-out. Ideally, as the load balancer equally distributes incoming requests to active servers, the queue length in active servers will have the same distributions. With actual queue-length measurements from more than one server, one can estimate the underlying distribution and, therefore server capacity, with higher accuracy.

Although some servers may adapt their service quality (for example, the richness of the content) in certain situations as in [39], we assume that the resource demands by incoming requests are not affected by cluster status. If lower quality content were equally good for business, it would be wise to always provide lower quality content. Otherwise, buying additional equipment is generally a cheaper and more predictable approach to dealing with recurring capacity shortage. While there is some feedback between request arrival and cluster performance (especially in benchmark programs), in a large server with many independent request sources, the effects of such client-server synchronization feedback will be marginal.

4 Metrics

With a resizable server cluster like the one defined above, it becomes possible to adapt the size of a cluster online to match the demand in real time. Data center administrators would like to optimize the allocation of resources to host as many services as possible and, therefore, maximize their revenue. On the other hand, they would always want to provide a high level of service quality to their customers.

The quality of service is often defined in SLAs between service providers and customers as the binding contracts,

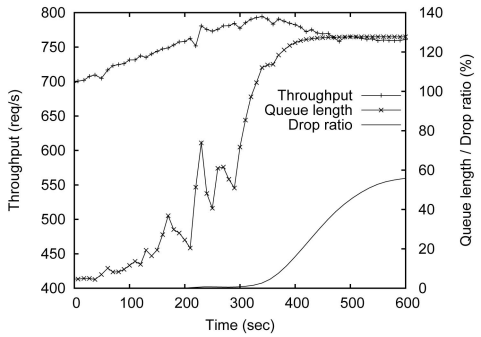

Fig. 2. When the server becomes overloaded, the queue length grows, and packets are dropped when the queue is full.

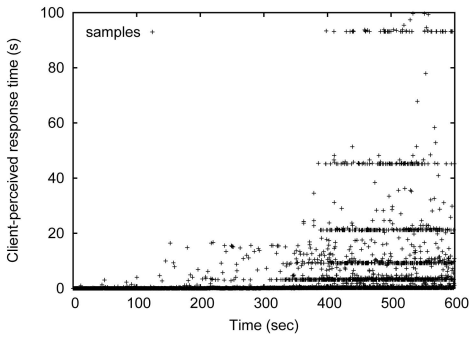

Fig. 3. Client-perceived response time distribution. As workload increases with time and the server gets saturated, TCP handshake packets are retransmitted, and time-out can be easily seen.

which typically specify rough performance numbers below which the service is considered unavailable or unacceptable. Networking SLA requirements are typically phrased in terms of availability, bandwidth, loss rate, latency, and jitter constraints [40]. In a computing utility environment like Océano [12], availability, response time, server load, assigned resources, and output bandwidth are proposed as requirements and goals in SLAs. Such SLAs are, in general, not very precise with respect to server performance. For example, it is simply too difficult to separate network delay from server delay.

Moreover, delays depend on the service time, which, in turn, depends on the type (for example, static or dynamic content) of the service, size and access frequency of a specific object, and many other factors. It is usually the service developer's job to optimize performance. Thus, delay is one of the metrics that a service provider cannot commit, except on a very basic level.

On the other hand, loss rate and failure-based metrics are easier to validate and are much less ambiguous. Therefore, service quality can be expressed in terms of loss-related quantities. Since we assume that requests are encapsulated in reliable TCP connections, the only cause of dropping a request is encountering a full queue at a busy server. Although one can also queue service requests in the user space to avoid drops at kernel TCP implementation (given the kernel can provide such resources), if users lose patience and disconnect due to the prolonged delay in service, it then wastes both network and server resources. Therefore, queuing and dropping requests at the kernel is preferred.

In order to study the relationships among the average queue length, TCP connection request dropping, and client-perceived service quality, we injected identical static HTTP requests with exponentially distributed interarrival times to one server. While increasing the request rate linearly over time, we monitor the accept queue in the Linux kernel and measure the response time at the client side. As shown in Fig. 2, as the request rate grows with time, the queue length starts to increase at around 700 requests/s and continues to grow until the queue is completely filled up with requests. As the request rate increases independently of actual throughput, the request drop ratio also increases with time. Dropped requests will be retransmitted, and the amount of time waiting for retransmission(s) is figured in the client-perceived response time, which is plotted in Fig. 3.

Although the LAN transmission delay is negligible, as requests are dropped by the server, the gap between the response times of requests that experienced no drop and one or more drops is obvious. The gaps actually map to different TCP time-out values of 3, 6, 12, 24, and 48 seconds. With users expecting prompt responses from the servers, especially in Web-based services, even a 3 second delay would deteriorate client-perceived service quality seriously in spite of whatever service time the request may take. Experiencing time-outs of 6 seconds or more (that is, when servers can only respond at least 9 seconds after the request first arrived at the server) can make users lose patience and recede [26].

Based on the abovementioned observations, we define the *request drop ratio* as the number of TCP connections failed to be established divided by the number of connection attempts, where each retry is counted as a separate attempt.

With this definition of request drop ratio, we then define the *capacity* of a server system as the maximum number of requests a server can process within 1 second while keeping the *request drop ratio* under a certain threshold. We choose the threshold of request drop ratio as 1 percent in this paper. We believe that a 1 percent request drop ratio is a reasonable choice, since, statistically, only less than 0.01 percent of request arrivals will experience a delay of 9 seconds or more.

5 Estimation Mechanisms

Server capacity can be estimated in many different ways. We first introduce two commonly used approaches and propose a new queue-monitoring-based algorithm and discuss its rationale.

5.1 Offline Static Estimation

Benchmark programs are commonly used to measure a Web server's ability to handle requests [16], [17], [18]. Some of them reveal the maximum throughput, and others generate fixed workload and report performance metrics, such as request-acceptance ratio and response time. For the latter type of benchmarks, users can set performance requirements and run the program repetitively with heavier workloads until the result violates the requirements.

Although benchmark programs can be used to estimate server capacity, they are also designed with a specific

workload model in mind, such as server functions, user access patterns, content popularity distributions, and so on. However, with new service components introduced frequently and workload shifts during the course of a day, the fixed benchmark model and parameters cannot capture these dynamics. Consequently, benchmark programs are good only for comparing hardware performance in controlled environments, but not for online capacity estimation. With an inaccurate estimation by benchmark programs, one may overestimate or underestimate the actual capacity of the server cluster, depending on the degree of the benchmark's deviation from the real workload.

5.2 CPU-Utilization-Based Estimation

Another usual approach is to log system utilization and estimate achievable maximum throughput from it. System utilization usually includes CPU, network, and disk utilizations. Inside a data center, network bandwidth is abundant, so it is usually not the bottleneck. Previous research showed that the popularity of Web objects follows Zipf-like distributions [15]. Thus, with sufficient memory installed, most Web objects can be cached and served from memory, and the remaining disk accesses are made via a high-speed SAN. Hence, CPU becomes the most probable bottleneck in Web servers. Therefore, one may want to estimate capacity by dividing current throughput by CPU utilization in order to assess the number of requests a fully utilized CPU can serve.

This method is simple and can adapt to changes in the workload model. For example, when the proportion of dynamic content requests increases in the incoming workload, estimation based on CPU utilization can capture the change as an increase in average CPU demand and then adjust the capacity estimation accordingly. However, servers have only a limited queuing capacity to hold a surge of incoming requests. This becomes a problem when utilization approaches 100 percent because the very long predicted queue length will not fit within the system-imposed queuing limits. Therefore, utilization-based estimation typically overestimates server capacity by an unknown workload-dependent factor.

5.3 Queue Monitoring

As the cost of dropping requests from the queue is high and our definition of capacity directly depends on the request drop ratio, we propose to derive capacity estimation from the queue itself, which is called *queue monitoring*. The insight behind queue monitoring is that the queue length grows irrespective of the type of bottleneck resource. As long as there is a mismatch between request arrival and service rates or a change in client access pattern or service configuration, the queue length will reflect it immediately.

We assume that the queue can hold up to K connection requests and is sampled every T seconds. In each sampling period, we count the number of enqueuing attempts n_{arr}, the number of connections accepted by server processes n_{acc}, and the number of connections dropped n_d. (The time index t of each sampled value and its derivatives are omitted for clarity.) The queue length is then increased by $\Delta\ell = n_{arr} - (n_{acc} + n_d)$ at the end of each sampling period. Instead of using the instantaneous queue length at the beginning of each sampling period, we use the average queue length $\overline{\ell}$ seen by incoming requests during each sampling period as it also captures the bursty arrivals and acceptances of Web requests. We also define arrival rate $\lambda = n_{arr}/T$ and service rate $\mu = n_{acc}/T$. The current request drop ratio is defined as $\varepsilon = n_d/n_{arr}$, which will be used to adapt the capacity estimate c and verify the constraint in the definition of capacity in Section 4.

As shown experimentally in Section 4, a queue of average length less than a certain threshold will never cause any requests to be dropped. The value of this threshold depends on the underlying workload characteristics, such as the arrival rate and burstiness. Since an accurate workload model is usually not available, we first estimate the threshold from the results of queue monitoring.

The difference between peak arrival rate, λ_{max}, and current capacity estimation, c, represents the maximum mismatch between the arrival and service rates. It is also the fastest rate that connection requests can accumulate in the queue. The time to fill up the queue, t_f, can be calculated as

$$t_f = \begin{cases} K/(\lambda_{max} - c) & \text{if } \lambda_{max} > c \\ \text{undefined} & \text{otherwise.} \end{cases} \quad (1)$$

With the definition of t_f, we can estimate the threshold of the minimal average queue length, ℓ_{th}, that could lead to queue overflow within a sampling period. If t_f is shorter than a sampling period, the queue could be empty at the start of a sampling period and overflow before the end; otherwise, there must be some requests queued up at the beginning. For the first case, the minimal average queue length is achieved by having an arrival rate c for the first $T - t_f$ seconds in a sampling period and arrival rate λ_{max} for the remaining t_f seconds. Hence, there are $c \cdot (T - t_f)$ requests in the first phase facing an empty queue and $\lambda_{max} \cdot t_f$ requests facing an average queue length $K/2$. Therefore, the average queue length over all requests during the sampling period can be calculated. Similarly, if $t_f > T$, the maximal queue accumulation rate is $\lambda_{max} - c$, and the minimal queue length at the beginning of a sampling period must be at least $K - (\lambda_{max} - c) \cdot T$ to have a full queue at the end of the sampling period. The average queue length is then simply the average of queue lengths at the beginning and the end. If the current average queue length exceeds the threshold, an overflow can occur. The threshold is calculated as

$$\ell_{th} = \begin{cases} \frac{(K/2) \cdot \lambda_{max} \cdot t_f}{c \cdot (T - t_f) + \lambda_{max} \cdot t_f} & \text{if } t_f \le T \\ \frac{[K - (\lambda_{max} - c)T] + K}{2} & \text{if } t_f > T. \end{cases} \quad (2)$$

After determining the threshold, we predict the future queue length by assuming the average queue length to grow/shrink linearly with time. If the current average queue length grows at a rate of ℓ' connections/second and a T_c-second forecast is needed to predict the future queue length due to a server start-up delay, then the predicted average queue length, $\hat{\ell}$, becomes

$$\hat{\ell} = \overline{\ell} + \ell' \cdot T_c. \quad (3)$$

With the queue monitoring data and the threshold derived above, we adapt the capacity estimate when 1) the request

drop ratio constraint, ε_{th}, is violated or 2) the current estimate is too optimistic or too pessimistic. An estimate is too optimistic if both the current or predicted queue length is greater than or equal to the threshold and the current estimated capacity is greater than the service rate. Similarly, an estimate is too pessimistic if both queue lengths are smaller than the threshold and the current estimated capacity is smaller than the service rate. We use a variable I to indicate an optimistic or pessimistic estimate:

$$I = \begin{cases} 1 & \text{if } \begin{cases} min(\overline{\ell}, \hat{\ell}) \geq \ell_{th} \wedge c > \mu \text{ or} \\ max(\overline{\ell}, \hat{\ell}) < \ell_{th} \wedge c < \mu \end{cases} \\ 0 & \text{otherwise.} \end{cases} \quad (4)$$

When the indicator shows a need to adapt the capacity estimate, we derive a new capacity estimate, c'. If the service level constraint is violated, we clamp down the capacity estimate to current service rate μ. If the current estimate is too optimistic or pessimistic, we adjust the estimate toward a more appropriate value with a factor α to prevent rapid changes in estimates. We note that the threshold will also be adjusted in the next sampling period accordingly. If none of these conditions holds, the original estimate is retained. The capacity adjustment equation can be expressed as

$$c' = \begin{cases} \mu & \text{if } \varepsilon \geq \varepsilon_{th} \\ c \cdot \alpha + \mu \cdot (1 - \alpha) & \text{if } \varepsilon < \varepsilon_{th} \wedge I = 1 \\ c & \text{otherwise.} \end{cases} \quad (5)$$

One small addition to this mechanism is a mode bit. The mode bit indicates any prior occurrence of downward adaptation of the capacity estimate. When a new server starts and joins a cluster, it can join with no knowledge of its potential capacity if it is the first server in the cluster or in a hybrid cluster where no other active server is of the same configuration. In that case, the mode bit is OFF and will trigger a "fast-start stage" by updating the capacity estimate c with current peak arrival rate λ_{max} if none of the downward adaptation conditions holds. Also, the estimate is reported as a "lower bound" of capacity as no difficulty has been encountered. Once any downward adaptation happens, the mode bit is set to ON, and the original adaptation behavior is engaged, which is called "adaptation stage." Once the mode bit is ON, it never flips back.

The parameters in this mechanism, including K, T, ε_{th}, and T_c, are all tunable to match system specification and the request drop ratio constraints. After staying for a certain period in the adaptation phase, an accurate capacity estimate can be obtained.

6 Implementation

To evaluate the performance of different estimation mechanisms and the effects of different measurement methods, we implemented various measurement methods from which an adaptive system can derive the needed parameters for online capacity estimates. Due to the dual connection queue structure in the Linux kernel adopted in this paper, we first provide some details of the Linux kernel before detailing the implementation of each of the measurement methods.

6.1 Linux Kernel Internal

A connection is placed in a backlog before it has been established and picked up by server processes. Although the size of the backlog is specified when calling `listen`, there are actually two different queues—SYN and accept queues—each for a listening socket in the Linux kernel.

When the first SYN packet arrives from a network interface, the new connection is only half-opened and placed in the SYN queue. The size of the SYN queue depends on the amount of physical memory available in the server. For example, machines with 512 Mbytes of memory have a SYN queue of size 1,024. However, if either the SYN queue or the accept queue is full at the time of a packet arrival, the SYN packet is dropped quietly, and no SYN + ACK packet will be returned to the client. Therefore, the client who generated the SYN packet will treat it as a packet loss and will retransmit the SYN packet a few more times with certain time-outs in-between. The length of each time-out is determined by the client OS and usually increases from a few hundred milliseconds to 1 minute. A packet dropped in this stage will increase the Simple Network Management Protocol (SNMP) counter `TcpAttemptFails` by 1.

After a SYN + ACK packet is sent and acknowledged by the client with an ACK packet, the connection is fully established and moved to the accept queue. The size of the accept queue, which is the only backlog existing in other OSs, is specified by the server program in the `listen` call with an upper bound of 128 imposed by the glibc 2.x. When the accept queue is full, the kernel quietly drops the ACK packet as if it had never been received. As a result, the server will consider the SYN + ACK packet not received by the client and will retransmit after a server-side time-out. On our platform, the kernel resends the packets up to five times with time-outs of 3, 6, 12, 24, and 48 seconds, respectively. The Linux kernel also counts the number of dropped ACK packets over all accept queues in a variable named `ListenOverflows`, which is not standardized in SNMP.

6.1.1 Kernel-Exported Measurements

The `/proc` pseudo file system in Linux is an interface for user-space programs to retrieve system information and tweak kernel parameters. CPU utilizations in number of jiffies are included in `/proc/stat`; `/proc/net/snmp` and `/proc/net/netstat` contain SNMP-compliant and non-SNMP-compliant network statistics, respectively. Due to the separate queue design, the total number of queue overflows is the sum of TcpAttemptFails (SYN queue overflows) and ListenOverflows (accept queue overflows). With both counter values available, we can obtain an accurate number of TCP retransmissions caused by resource scarcity. A Perl script is used to poll these values, and capacity is estimated from CPU utilization. Unfortunately, this interface is not standardized, but similar interfaces are available in other Unix flavors.

6.1.2 Kernel Instrumentation

In spite of the rich set of information available via the Linux `/proc` file system, the states of system queues, including both SYN and accept queues, are poorly exported. To assess the load of a system, it would be useful to access the

number of enqueuing attempts, number of dequeuing requests (`accept` system calls), number of queue overflows, and average queue length. By exporting more detailed queuing state information than is traditionally done in Unix-like OSs, online capacity management can be done with higher accuracy. (Note that mainframes traditionally export such finer-grain queuing statistics.) Thus, we modified Linux kernel 2.4 to measure the queue length in kernel space and provide the measurements via the `/proc` file system.

In our patched kernel, when a SYN packet is received by the kernel and is to be placed in the SYN queue, a SYN packet count and an accumulative queue length counter are incremented to keep track of the total SYN queue length seen by incoming SYN packets. By periodically polling the values of these two counters and dividing the incremental total queue length by the SYN packet count, we can compute the average SYN queue length seen by SYN packets that arrived during the current sampling period. Similar counters are also placed in the accept queue. Moreover, we count the number of `accept` system calls as they pull connections from the accept queue.

If a queue is full and a request is about to be enqueued, the queue size (that is, current queue length) is added to the counter. When retransmitted packets arrive again at the same queue, they are counted as different requests and the current queue length will be added multiple times. The counter values and current system time are polled atomically to achieve the highest accuracy. With queue-length measurements directly from the kernel, we can utilize the queue monitoring introduced in Section 5 to estimate server capacity.

Kernel measurements are accurate and consistent even under heavy load. Furthermore, the latency for reporting queue lengths and TCP SYN drops is reduced significantly if monitoring is done at the kernel level. Reducing this latency is key to achieving instantaneous capacity estimates.

6.1.3 `vmstat` and `netstat`

An OS typically implements a set of user-space programs for basic system performance reporting and monitoring purposes. Although `vmstat` can generally provide average CPU utilization periodically across different platforms at a 1-Hz frequency, most `netstat` implementations only provide a snapshot of network statistics. Therefore, we redirect the output of the command "`vmstat 1`" to provide CPU utilization measurements every second to the Perl script, and as soon as a new sample is received, we execute "`netstat -s`" to retrieve a snapshot of network statistics. We also log the current system time with each sample so that the actual interval between samples can be learned.

User-space tools generally retrieve measurements from the OS kernel through a predefined proprietary interface, such as the `/proc` file system in Linux, and format data appropriately for direct inspection. The available measurements are limited and vary from one system to another. In the output of "`netstat -s`," we are mainly concerned with the number of incoming connection requests and the number of failed connection attempts. In most cases, a connection attempt fails if the initial TCP SYN packet is dropped by the server due to the full accept or SYN queue. However, if the first packet is accepted but the connection cannot advance to the accept queue when the TCP three-way handshake is complete, it is not counted as a failed attempt in Linux. Moreover, the current lengths of both queues are not available from `netstat` either. Although the execution of user-space programs increases the system overhead, the measured CPU utilization also reflects the cycles consumed by these programs.

6.1.4 SNMP

SNMP is a standard protocol for the management of network devices and, sometimes, servers. With the vendor's management information base (MIB) definitions, the load balancer in our testbed provides many aspects of network statistics on layer 4 (TCP) and below via SNMP queries, although only basic measurements are standardized. The UCD SNMP daemon on servers provides OS-specific counters, including raw CPU usage, which are mapped to the numbers of jiffies the Linux kernel scheduled in various modes (user, nice, system, and idle). The counter values are nominally the same as what we collected using typical user-space measurements. As in the case of network devices, nonnetwork-oriented information is mostly not standardized, thus creating problems similar to those of using platform-dependent user-space tools.

The accuracy of SNMP counters depends on the vendor's SNMP implementation (for example, update rate), which is outside the control of our capacity estimation. Moreover, nonstandardized information limits the ability to define a standard methodology for online estimation of system capacity. However, capacity estimates can be derived from periodic queries to load balancers and servers' SNMP agents in our testbed. We collect drop counts and other error indicators, such as queue overflow and buffer shortage indicators from the load balancer. As expected, the load balancer is fast enough to always forward all packets; not a single drop occurred during our experimentation.

6.1.5 Apache

Application instrumentation is obviously the gold standard for system monitoring. For example, in a monitoring framework such as Tivoli [41] or Openview [42], the server process can timestamp each incoming request and record start and end times. Logging is standardized in the *application response time measurement* (ARM) standard, which defines the API of source-code level logging. Unfortunately, Apache does not have an ARM module, and it is not always possible to modify application source code. Instead of using ARM, an Apache module, `mod_status`, provides server status reports, including some performance details, through a dynamic HTML document. With extended status turned on, performance metrics, including but not limited to total requests and CPU usage, are provided in the server status report. The CPU usage is obtained by calling the `times` system call and includes the time spent by HTTP daemon (httpd) processes (and their child processes) in both user and system modes. By periodically polling these variables, we can calculate average throughput and CPU utilization and, therefore, estimate throughput when the CPU is fully utilized. However, since the CPU will never be fully utilized

solely by server processes without any overhead, we expect this method to overestimate system capacity.

7 Evaluation

There are two common practices of evaluating server systems: 1) using synthetic workloads and 2) replaying real traffic traces. Synthetic workload generators, such as Surge [16], HTTPerf [17], and Spec Web [18], are usually designed to mimic user behavior and assess server performance. Therefore, they can be used for evaluating the accuracy of capacity estimation, which is the main goal of this paper. One can also tweak parameters to reflect different scenarios.

On the other hand, using traces such as Web server logs can only lead to results specific to the traces used. Furthermore, because TCP time-outs and packet transmission times cannot be changed easily, replaying a trace at an accelerated speed actually changes the client model. These effects are particularly significant in stress-testing a server system. For example, if a trace is played back at a 10x speed, a connection experiencing a 3 second time-out resulting from an accept queue overflow will wait 30 seconds in the trace time scale before it retries. In contrast, by changing the number of emulated users, a synthetic workload generator can maintain the same client model while stress-testing the server.

To evaluate various estimation mechanisms and measurement methods, we set up a testbed that consists of a Foundry Server-Iron XL load balancer, five identical PC servers with 512-Mbytes RAM each, and another five workload-generating PCs. Servers are connected to the load balancer via 100-megabit-per-second Ethernet links, and the load balancer has a 1-Gbps Ethernet interface to clients. We built a client emulation program, which is based on the Surge user model, to generate workload with an arbitrary time-varying number of emulated users. In Surge, each emulated user first generates a request to an HTML document and then fetches its embedded objects. The user then sleeps for a period of time and repeats this sequence of actions. We follow the distribution and parameters in [16]. The load balancer is configured to use the round-robin algorithm. Note that, since the capacity-estimation process is running at the server level instead of the cluster level, the choice of a load-balancing algorithm does not affect our results.

In this section, we first evaluate the accuracy and overhead of each measurement method. Second, the queuing behavior inside a server system is presented, and three capacity-estimation mechanisms are compared with each other under different types of workload. Finally, we apply the capacity-estimation schemes to realize an energy-efficient server cluster where the percentage of energy savings under different scenarios demonstrates the effectiveness of our online capacity estimation and its application.

7.1 Measurement Accuracy

Before estimating server capacity, we first compare the accuracy of the five measurement implementations in Section 6. This is required as a low overhead yet accurate measurement implementation can help improve capacity estimation as well. We set up all of them to simultaneously measure one running server at a 1-Hz sampling frequency

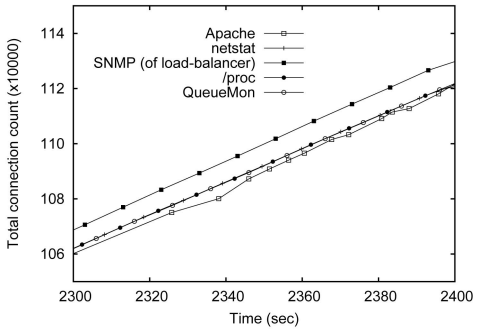

Fig. 4. Measuring the total number of connections obtained by different methods. As the server gets saturated, SNMP counts more connections than others, and Apache has big jumps in increments.

and compare the thus obtained results. The server is fed with increasing loads, which are based on the Surge workload model with an increasing number of users so as to assess the accuracy of measuring both a normal server and an overloaded server. Although running multiple measurement methods simultaneously increases the overhead of the server under test, this is the only way to make a direct comparison of all of them.

To estimate capacity, one of the most important parameters to measure is the total number of connections, which is also the total number of requests in our model. Before the server became saturated, all five methods yielded exactly the same result. However, as the server became saturated, it took more than 1 second, which is our sampling period, to retrieve the sample. We will discuss the cause of this later. Also, the total number of connections reported by Apache has some jumps, causing the zigzags in Fig. 4.

Moreover, the total number of connections obtained by SNMP grows faster than others after the server is saturated. This inconsistency is due to the interpretation of SNMP specification in the Linux kernel. In Linux 2.4, the kernel responds to SYN packets by sending SYN + ACK packets without guaranteeing that the connection will be queued in the accept queue when it is fully opened. The load balancer regards each three-way handshake as the creation of a connection and, hence, increases the SNMP `TotalConnections` counter by one as soon as the client responds with an ACK packet. However, the Linux kernel increases the `tcpPassiveOpens` counter only after the ACK packet is received *and* there is room in the accept queue. Therefore, as the accept queue is full for a nonnegligible period of time, many connections may not be able to move to the accept queue even after a few retries, and the inconsistency between counters arises. The way the Linux kernel handles the counter is less accurate according to the definition in RFC 1213 (also as STD 17) [43]: the number of times TCP connections have made direct transitions to the SYN-RCVD state from the LISTEN state.

We also compare the CPU utilization, which is defined as the fraction of time the CPU is not idle, measured by four of the five methods, with the exception of kernel instrumentation that is built for queue monitoring only. The user-space program `vmstat` represents CPU utilization in percentages, whereas all other methods express CPU utilization in total number of jiffies (of 10 ms each) spent in each mode. With

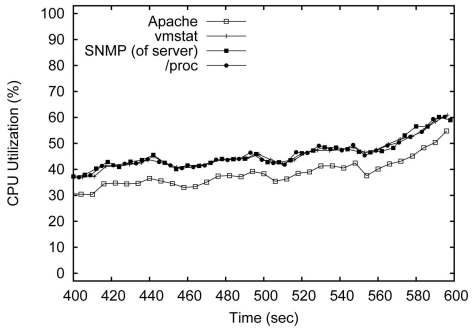

Fig. 5. Measuring CPU utilization using different methods. Apache only counts CPU time used by itself and, therefore, underestimates CPU utilization.

the actual sampling time of each point, we convert all CPU performance counters to percentages.

The CPU utilizations obtained by using different methods are plotted in Fig. 5. At first, it was even impossible to retrieve CPU performance counters via SNMP when the server was saturated because the SNMP agent had too little CPU time; this problem was solved by raising the SNMP agent's priority. The result shows that CPU utilization grows with the request rate (which increases linearly with time). Compared with the other three approaches, Apache consistently underestimates CPU utilization as it only counts the CPU time used by itself.

As the server under test becomes saturated, we also noticed that some of the measurement methods cannot keep up with the 1-Hz sampling frequency. The unpredictability of the sampling interval is due to the OS scheduler. We tagged each sample with the current system time when the sample is taken. Compared with the "before-sampling" time, we found that the "after-sampling" time is closer to the time when remote measurement entities (Apache and SNMP agent) actually processed the counter-retrieval request. The inter-sample intervals of these measurement methods are plotted in Fig. 6.

Similar to the SNMP agent, we also raise the priority of other measurement-related processes. As retrieving measurement results from the `/proc` file system does not create any new process, once the measurement-retrieval process is scheduled, the result collection can be done immediately. If we did not adjust the priority, there would be a fluctuating

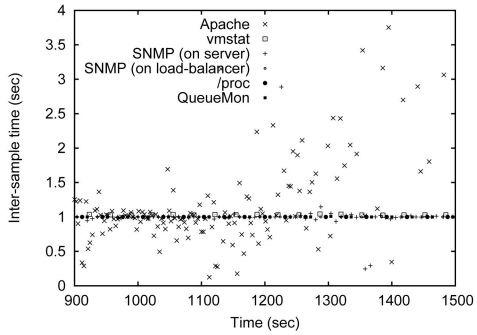

Fig. 6. Although we retrieve measurements every second, the actual intersample interval ranges between 0 and 2 seconds.

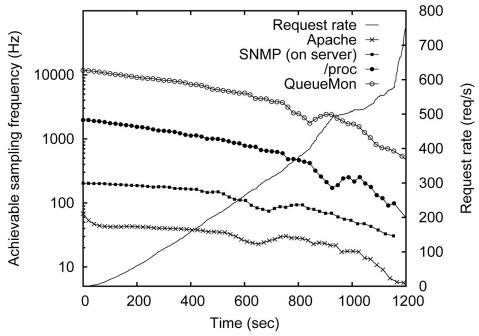

Fig. 7. The maximum sampling frequency of each method. The overhead of each measurement method differs from the others by two orders of magnitude. Thus, the achievable sampling frequency ranges from a few hertz to a few hundred hertz when the server becomes overloaded.

delay between 0.8 and 1.2 seconds. For latter experiments, all measurement processes are running at a higher priority. The only method that suffered from frequent deviations around a fixed sampling delay is Apache status retrieving, as it must be scheduled with other Apache processes.

With these results, we conclude that reading from the `/proc` file system with the reader process staying in the memory and priority promoted can provide the most stable measurements of a running system. Although the interfaces provided by SNMP and Apache are more generic, they are not designed for frequent retrieval, especially under overload situation.

7.2 Measurement Overhead

Using the same evaluation setup as above, we also compare the overheads of the measurement methods. Instead of periodically sampling the server, we execute one method at a time and measure the system continuously. The maximum sampling frequency of each method is plotted in Fig. 7.

As the figure shows, when there is little load on the server, four of the five measurement methods can acquire samples at a rate of 50 Hz or much higher, with the exception of user-space measurement tool `vmstat`, which has a limit of 1-Hz sampling frequency by design. However, as the request rate increases linearly with time, the maximum achievable sampling frequency drops exponentially. A high-overhead measurement method like Apache can only achieve a sampling rate of 5 Hz. On the other hand, we can still retrieve queue-length measurement results at 500 Hz. Although we do not require a high sampling frequency to estimate capacity, it shows that kernel instrumentation is a low-overhead measurement method.

7.3 Queuing Behavior

Before applying capacity estimation using the queue-length monitoring proposed in Section 5, we first examine how various queues in a server system interact with each other and when incoming packets are dropped.

As Fig. 8 shows, as the request rate increases, the run queue length first reflects the increasing workload. After we reached the limit of the allowable number (256) of httpd processes, incoming requests start to be accumulated in the

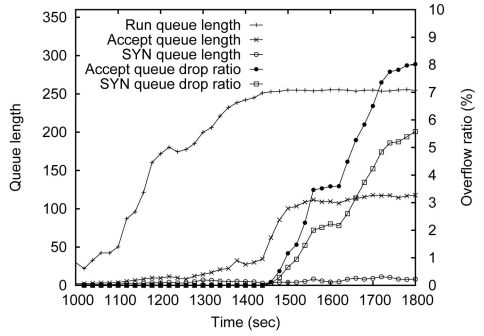

Fig. 8. As the request rate increases, the average run queue length first reflects the increased resource demand, and then the accept queue grows until it becomes full. Packets are dropped when the queue is full and, therefore, SYN packets seldom build up in the SYN queue.

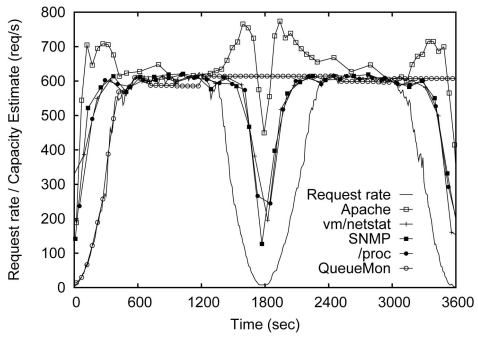

Fig. 9. Capacity estimates using five different measurement methods.

accept queue. No request will be dropped because of the httpd processes count limit. As the accept queue length grew, we observed that SYN packets were dropped even without any queue filled up. This was due to the queue size limit of an outgoing packet scheduler. Although the packet scheduler normally does not drop any packet, the SYN + ACK packet is an exception in Linux. If the SYN + ACK packet cannot be scheduled to send, the Linux kernel does not retry before dropping it. As the default packet scheduler has a queue of only 100 packets, bursty traffic can easily fill up the queue before the device driver has a chance to drain some of the packets. With the limit increased to 1,000, no more SYN packets were dropped at this stage.

Before the average accept queue length approaches its limit (128), the instantaneous accept queue length, which is highly fluctuating and which we choose not to plot in the figure for clarity, has reached the limit every now and then. Not only does a full accept queue block establishing TCP connections out of the queue (when the ACK packet is received from the client), but also the first SYN packet from the client will be dropped. Therefore, we can see that both the accept queue and SYN queue drop ratios increase dramatically; almost 10 percent of new connections cannot be established if the accept queue is full for a few minutes. Since many SYN packets are rejected, the SYN queue length grows at a very moderate pace and is far from its limit (1,024). From this result, we can see that the accept queue does reflect server load.

7.4 Capacity Estimation

Using the measurement results, we apply all three capacity-estimation mechanisms to the server under test. To determine the actual capacity of the server, we feed a linearly increasing Surge workload to the server and monitor the request drop ratio over a 1 minute window. Once the service level is violated, we call the average throughput the capacity of the server. The process is repeated 10 times and determines that the server has a capacity of 595 requests/second under this workload.

Instead of using the simple linearly increasing workload, we create a sinusoid-like workload, where the number of Surge user-emulated changes like a sine wave, to evaluate the capacity-estimation mechanisms under fluctuating workloads. When the workload is CPU bound, simple estimation like the CPU-utilization-based mechanism we discussed in Section 5 can effectively approximate system capacity. Although this method is simple, the quality of its inputs (that is, measured data) can seriously affect its prediction. To this end, we compare the accuracy of four measurement mechanisms that can be used as input to any capacity-estimation method, including the simple one mentioned above. The result of capacity estimation by queue monitoring is also included for the purpose of comparison.

Fig. 9 shows how these mechanisms perform relative to each other. Use of Apache's status yields a poor estimate of server capacity. It typically overestimates load when the workload is low. This is due mainly to various layers of indirection and inaccuracies introduced by each of them; for example, the CPU time used by background processes is not taken into account. Estimates by using the output of `vmstat` and `netstat` commands using SNMP counters and measurements in the Linux's `/proc` file system are all very close to each other. They all produce a reasonable estimate when there is a medium-level load on the server, for example, during 200 and 400 seconds in this experiment. Once the server is overloaded, the estimate essentially equals the current throughput while a large number of incoming packets are dropped.

The capacity estimated by the queue-monitoring scheme grows at first with the request rate (the fast-start stage). As soon as the server gets overloaded, it actually estimates a capacity lower than the current throughput, and the estimate, 586 reqs/second, is actually very close to the one we obtained by static estimation, 595 reqs/second. After the overloaded situation disappears, it yields an estimate of 614 reqs/second as the queue length drops to a safe region. Most importantly, in between the two overloaded periods (from 1,200 to 1,800 seconds), since there are no new triggers to adapt capacity estimation, it keeps the current estimate. As a stable estimate is very important for making control decisions, this queue-monitoring scheme can provide better estimates than the other schemes.

The CPU is not the only potential bottleneck of a system, albeit the easiest to measure. Estimating server capacity when a resource other than CPU is the bottleneck is much more difficult because I/O utilization is fuzzier than CPU utilization. To simulate an I/O-bound workload, before the completion of each request, we artificially inserted a 0.5 second sleep to simulate an I/O wait, such as the time a front-end Web server waiting for back-end database committing transactions, which is common in multitier server design. This design moves the bottleneck from CPU to the configured maximum number of server processes in

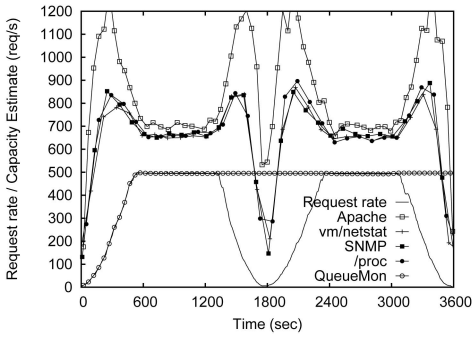

Fig. 10. Capacity estimation by five different methods. Only the proposed queue-monitoring scheme can estimate the server capacity under an I/O-bound workload.

Apache or maximum number of processes in OS, whichever is smaller. Although a fixed I/O time is unrealistic, it reflects that a batch of requests causing some I/O delay can easily change the workload from CPU bound to I/O bound.

Estimates of server capacity when an I/O-bound workload is present are plotted in Fig. 10. Obviously, all estimates based on CPU utilizations are overly optimistic by a large margin. Only our queue-monitoring scheme provides a stable output irrespective of the bottleneck shift. That is, the server capacity estimation based on the proposed queue monitoring is adaptive and can handle a wide variety of workloads.

7.5 Application: Energy Conservation

With an accurate capacity estimate, adapting the size of a server cluster to match its workload becomes straightforward. Traditionally, sizing a server cluster is either based on a fixed schedule or static capacity estimation by executing benchmark programs. A utility-function-based mechanism was proposed in [1], but it requires prior knowledge of both the underlying workload and pricing models. A fixed schedule is usually derived by observing typical daily demand fluctuations from the past utilization log. A constant number of servers are activated to provide services regardless of current utilization or service quality. The time granularity of a schedule depends on how fast the workload changes and is usually in the order of an hour. Although this scheme is simple, it does not provide any service-quality guarantee, and the cluster is usually underutilized.

Another commonly used practice is to benchmark each server's capacity and then use controllers to adjust the number of active servers accordingly. We call controllers based on static capacity estimation *static controllers*. The static controller turns on a new server when the current aggregated arrival rate is higher than the current aggregated capacity and turns off if a server can be spared. After a newly turned-on server starts accepting requests, we also enforce a 10 second warm-up time before the next capacity adjustment. Safety margins are also frequently used to avoid capacity overestimation. However, as good capacity estimation is available for our static controller, we did not apply such margins.

With online server capacity estimation, a cluster-sized controller as simple as a static controller can still be utilized and will be more effective. We call these controllers *dynamic controllers*. The only difference is that it uses the online capacity estimate instead of the fixed capacity setting. With more accurate capacity estimation, the timing to adjust cluster size can be more precise and, thus, more energy can be saved without sacrificing service quality.

We measured the power consumption of each server under boot, idle, and fully loaded conditions and of the load balancer. Since our platform does not support sophisticated energy-management approaches (for example, PowerNow), the power usage does not vary widely from idle ($\sim$ 60 W) to fully loaded ($\sim$ 65 W). According to the actual turn-on/turn-off time that we measured, we estimated the total power consumption for the entire cluster. In what follows, percentage energy savings are compared with always-on server clusters. Although much energy is wasted for very little productivity (if any), "always-on" is still the most common practice today.

7.5.1 "Standard" Workload

We first compare static controllers with dynamic controllers by using the same workload model that was used to determine server capacity statically. To predict real-world energy savings, we construct an average daily server load curve using the ClarkNet-HTTP Web server log from the Internet Traffic Archive [44] and scale it to fit our testbed. Its sinusoid-like appearance is merely a coincidence. Each emulated user still follows the Surge workload model. We also compress the time scale by a factor of six to accelerate experiments.

Since the workload is the same as the one that the static controller used to benchmark server capacity, and the dynamic controller has an accurate estimate as well, both controllers perform almost the same in executing this workload. Both controllers achieve 38 percent of energy savings and merely 0.001 percent of total requests experienced accept-queue overflows and penalties of TCP timeouts. Although the performance is similar, the dynamic controller does not require any prior knowledge of the workload and is therefore easier to deploy as it eliminates the need for benchmarking. In addition, since a benchmark workload usually differs from the actual workload and the workload itself also changes during the course of a day, static controllers cannot achieve the highest utilization in reality. For comparison with fixed-schedule controllers, we have also built a capacity schedule based on the average daily load curve. If the schedule has a time slice of 30 minutes, 31 percent of energy can be saved. If the size of the time slice increases to 2 hours, energy savings are reduced to 25 percent.

7.5.2 Decreased Resource Demand

When per-request resource demand is reduced, we expect that each server can handle more requests in a fixed amount of time and fewer servers are needed to handle the same workload. To lower the resource demand, we half the object size created by Surge and keep the other parameters unchanged. The throughput and the number of servers activated by both controllers are plotted in Fig. 11. The static controller still turns on new servers when the throughput reaches 595 reqs/second for each server. However, active servers did not reach full utilization at that moment and, hence, turning on new servers would consume extra energy. On the other hand, the dynamic controller estimated each server to be able to handle about 750 reqs/second, so it turned on servers later and turned off servers earlier. At the end of the experiment, the server cluster with

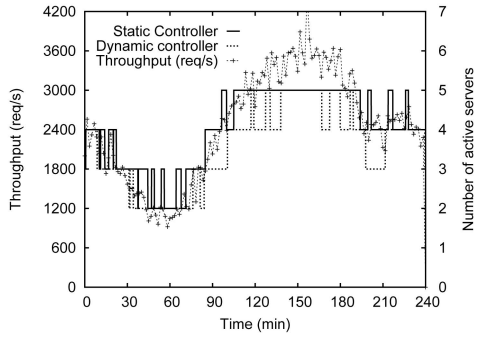

Fig. 11. When the object size is halved, server capacity increases. The dynamic controller activates fewer servers, saving more energy than the static controller.

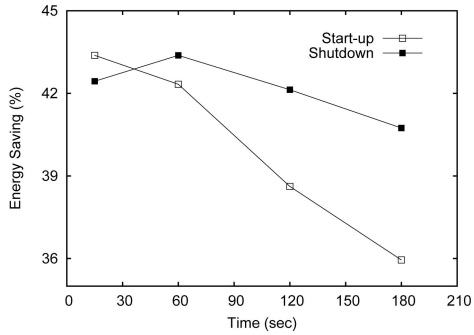

Fig. 12. The percentage of energy conservation depends more on start-up delays than shutdown delays.

the static controller is found to save 35 percent of energy, whereas the one using the dynamic controller saves 44 percent of energy. The service quality is well maintained as less than 0.001 percent of the total requests experienced accept-queue overflows. This demonstrates the ability of capacity estimation to correctly recognize the decreased resource demand in each request, thus saving more energy.

7.5.3 Increased Resource Demand

As per-request resource demand increases, more servers would be needed to handle the increased workload. To simulate the demand increase, we replace each static document with a Server-Side Include (SSI) dynamic document. Since Web servers are required to parse the content of the document before sending it out, dynamic content will increase the resource demand of processing each request.

When the static controller is applied to this scenario, the result becomes totally unacceptable as the server saturated before reaching its outdated estimated capacity. Therefore, no new server is turned on to shed the workload from the already overloaded server. A large number of requests failed to go through. On the other hand, the dynamic controller does not suffer from the same problem. As the active server gets saturated, new servers are activated, and the quality of service is maintained. The result shows that 31 percent of energy is saved in this experiment, and only 0.002 percent of total requests experienced queue overflows.

7.5.4 Start-Up and Shutdown Delay

Since current server systems need a nonnegligible amount of time to boot up and start serving clients, we use T_c in our queue-monitoring scheme to compensate for such a delay. A long start-up delay essentially requires a larger prediction window and, therefore, lowers the capacity of each server since we have to spare some queue space for the predicted increase of workload. This effect is similar to the qualitative analysis given in [33]. Long shutdown delays also decrease the agility of server cluster resizing since a server that is currently being powered down cannot be turned on until it is completely shut down.

To study the effect of start-up and shutdown delays, we keep all servers up and simulate the delays in the controller. The controller includes the powering-on server in the load balancer's configuration only after the simulated start-up delay has elapsed. Similarly, when a server is turned off, it is removed from the configuration and changed into a soft shutdown state, which is maintained by the controller. The effects of start-up and shutdown delays are plotted in Fig. 12.

While keeping the shutdown delay at 60 seconds, we simulate various start-up delays between 15 and 180 seconds. As the result shows, a delay longer than 60 seconds starts affecting the energy conservation. The percentage of energy conservation drops from 43 to 36 percent. This result suggests that a fast-reacting server platform can save considerably more energy. Similarly, while keeping the start-up delay at 60 seconds, long shutdown delays also reduce the energy-savings benefit, but to a lesser extent. We expect that a fast-oscillating workload can benefit short shutdown-delay configurations, although such a workload is not commonly seen.

8 Conclusions

In this paper, with a symmetric server cluster model and empirical evidences showing the impact of dropping TCP handshake packets, we defined the capacity of Internet servers as sustainable throughput with a low (for example, 1 percent) SYN request drop ratio. This definition ensures that the client-perceived performance remains acceptable as long as enough capacity is allocated. To estimate capacity, we proposed a new mechanism based on listen-queue monitoring and demonstrated that it is superior to utilization-based estimation. Also, several different means of collecting system performance measurements are all implemented, tested, and evaluated. We found pitfalls in different measurement methods, such as the priority of measurement-collection processes and the actual meaning of each counter value. More importantly, our queue-monitoring mechanism can obtain a good estimate of server capacity irrespective of whether workload is CPU bound or I/O bound.

We applied the proposed capacity estimation to adapt the size of a Web server cluster. By turning on and off server machines to match the actual workload, energy consumption can be reduced significantly, and the utilization (and, therefore, the profit) of a data center can be maximized. Without requiring prior knowledge of workload or cost, we achieved 31 to 44 percent of energy savings under various workloads, thus confirming the value of a good capacity-estimation mechanism.

Acknowledgments

The work reported in this paper was supported in part by the US National Science Foundation under Grant CCR 0216977 and by the Hewlett Packard Laboratories. An earlier version of this paper was presented in slide format at Integrated Network Management 2005 (IM05).

REFERENCES

[1] J.S. Chase, D.C. Anderson, P.N. Thakar, A.M. Vahdat, and R.P. Doyle, "Managing Energy and Server Resources in Hosting Centers," *Proc. 18th ACM Symp. Operating Systems Principles (SOSP '01),* pp. 103-116, 2001.

[2] Y. Chen, A. Das, W. Qin, A. Sivasubramaniam, Q. Wang, and N. Gautam, "Managing Server Energy and Operational Costs in Hosting Centers," *Proc. Int'l Conf. Measurement and Modeling of Computer Systems (SIGMETRICS '05),* pp. 303-314, 2005.

[3] A. Fox, S.D. Gribble, Y. Chawathe, E.A. Brewer, and P. Gauthier, "Cluster-Based Scalable Network Services," *Proc. 16th ACM Symp. Operating Systems Principles (SOSP '97),* pp. 78-91, 1997.

[4] A. Iyengar, J. Challenger, D. Dias, and P. Dantzig, "High-Performance Web Site Design Techniques," *IEEE Internet Computing,* vol. 4, no. 2, pp. 17-26, 2000.

[5] T.P. Brisco, *DNS Support for Load Balancing,* IETF RFC 1794, Apr. 1995.

[6] D.M. Dias, W. Kish, R. Mukherjee, and R. Tewari, "A Scalable and Highly Available Web Server," *Proc. IEEE CS Int'l Conf. (COMPCON '96),* pp. 85-92, 1996.

[7] W. Zhang, "Linux Virtual Server for Scalable Network Services," *Proc. Ottawa Linux Symp.,* 2000.

[8] X. Zhang, M. Barrientos, J.B. Chen, and M. Seltzer, "HACC: An Architecture for Cluster-Based Web Servers," *Proc. Third USENIX Windows NT Symp.,* pp. 155-164, 1999.

[9] L. Cherkasova and M. Karlsson, "Scalable Web Server Cluster Design with Workload-Aware Request Distribution Strategy," *Proc. Third Int'l Workshop Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS '01),* pp. 212-221, 2001.

[10] V.S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E.M. Nahum, "Locality-Aware Request Distribution in Cluster-Based Network Servers," *Proc. Eighth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII),* pp. 205-216, 1998.

[11] Z. Ge, P. Ji, and P. Shenoy, "A Demand Adaptive and Locality Aware (DALA) Streaming Media Server Cluster Architecture," *Proc. 12th Int'l Workshop Network and Operating Systems Support for Digital Audio and Video,* pp. 139-146, 2002.

[12] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger, "Océano—SLA Based Management of a Computing Utility," *Proc. Int'l Federation for Information Processing (IFIP)/IEEE Int'l Symp. Integrated Network Management,* pp. 855-868, 2001.

[13] M.E. Crovella and A. Bestavros, "Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes," *IEEE/ACM Trans. Networking,* vol. 5, no. 6, pp. 835-846, 1997.

[14] M. Arlitt and T. Jin, "Workload Characterization of the 1998 World Cup Web Site," Technical Report HPL-1999-35(R.1), Hewlett-Packard Laboratories, 1999.

[15] L. Breslau, P. Cao, L. Fan, G. Philips, and S. Shenker, "Web Caching and Zipf-Like Distributions: Evidence and Implications," *Proc. IEEE INFOCOM '99,* pp. 126-134, 1999.

[16] P. Barford and M.E. Crovella, "Generating Representative Web Workloads for Network and Server Performance Evaluation," *Proc. Int'l Conf. Measurement and Modeling of Computer Systems (SIGMETRICS '98),* pp. 151-160, 1998.

[17] D. Mosberger and T. Jin, "httperf: A Tool for Measuring Web Server Performance," *Proc. First Workshop Internet Server Performance,* pp. 59-67, June 1998.

[18] Standard Performance Evaluation Corporation (SPEC), "SPECweb99 Benchmark," 1999, http://www.spec.org/osg/web99/.

[19] J. Judge, H.W.P. Beadle, and J. Chicharo, "Sampling HTTP Response Packets for Prediction of Web Traffic Volume Statistics," *Proc. IEEE Globecom,* 1998.

[20] W. Shi, R. Wright, E. Collins, and V. Karamcheti, "Workload Characterization of a Personalized Web Site and Its Implications for Dynamic Content Caching," *Proc. Seventh Int'l Workshop Web Caching and Content Distribution (WCW '02),* pp. 1-16, 2002.

[21] M. Aron, P. Druschel, and W. Zwaenepoel, "Cluster Reserves: A Mechanism for Resource Management in Cluster-Based Network Servers," *Proc. Int'l Conf. Measurement and Modeling of Computer Systems (SIGMETRICS '00),* pp. 90-101, 2000.

[22] G. Banga, P. Druschel, and J.C. Mogul, "Resource Containers: A New Facility for Resource Management in Server Systems," *Proc. Third Symp. Operating Systems Design and Implementation (OSDI '99),* pp. 45-58, 1999.

[23] Y. Diao, N. Gandhi, J.L. Hellerstein, S. Parekh, and D.M. Tilbury, "Using Mimo Feedback Control to Enforce Policies for Interrelated Metrics with Application to the Apache Web Server," *Proc. Eighth IEEE/IFIP Network Operations and Management Symp. (NOMS '02),* pp. 219-234, 2002.

[24] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus, "Using Control Theory to Achieve Service Level Objectives in Performance Management," *Proc. IFIP/IEEE Int'l Symp. Integrated Network Management,* 2001.

[25] M. Welsh and D. Culler, "Adaptive Overload Control for Busy Internet Servers," *Proc. Fourth USENIX Symp. Internet Technologies and Systems (USITS '03),* 2003.

[26] A. Bouch, A. Kuchinsky, and N. Bhatti, "Quality is in the Eye of the Beholder: Meeting Users' Requirements for Internet Quality of Service," Technical Report HPL-2000-4, Hewlett-Packard Laboratories, 2000.

[27] J. Nielson, *Usability Engineering.* Academic Press, 1993.

[28] R. Rajamony and M. Elnozahy, "Measuring Client-Perceived Response Times on the WWW," *Proc. Third USENIX Symp. Internet Technologies and Systems (USITS '01),* 2001.

[29] D.P. Olshefski, J. Nieh, and D. Agrawal, "Inferring Client Response Time at the Web Server," *Proc. Int'l Conf. Measurement and Modeling of Computer Systems (SIGMETRICS '02),* pp. 160-171, 2002.

[30] M. Weiser, B. Welch, A.J. Demers, and S. Shenker, "Scheduling for Reduced CPU Energy," *Proc. First Symp. Operating Systems Design and Implementation (OSDI '94),* pp. 13-23, 1994.

[31] P. Pillai and K.G. Shin, "Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems," *Proc. 18th ACM Symp. Operating Systems Principles (SOSP '01),* pp. 89-102, 2001.

[32] P. Bohrer, E.N. Elnozahy, T. Keller, M. Kistler, C. Lefurgy, and R. Rajamony, "The Case for Power Management in Web Servers," *Power-Aware Computing,* R. Graybill and R. Melhem, eds., Kluwer/Plenum Series in Computer Science, Jan. 2002.

[33] K. Rajamani and C. Lefurgy, "On Evaluating Request-Distribution Schemes for Saving Energy in Server Clusters," *Proc. 2003 IEEE Int'l Symp. Performance Analysis of Systems and Software (ISPASS '03),* pp. 111-122, 2003.

[34] E. Pinheiro, "Load Balancing and Unbalancing for Power and Performance in Cluster-Based Systems," Technical Report DCS-TR-440, Dept. of Computer Science, Rutgers Univ., 2001.

[35] M. Elnozahy, M. Kistler, and R. Rajamony, "Energy Conservation Policies for Web Servers," *Proc. Fourth USENIX Symp. Internet Technologies and Systems (USITS '03),* 2003.

[36] P. Nuutinen, G. Hope, H. Arhippainen, L. Cuypers, W. Milanowski, R. Reesen, and G. Shannon, "Capacity Planning for Logical Partitioning," 2001, http://www.redbooks.ibm.com/redbooks/pdfs/sg246209.pdf.

[37] Sun Microsystems, "Sun Dynamic System Domains," http://www.sun.com/servers/whitepapers/domains.htm, 2005.

[38] Hewlett-Packard, "The HP Partitioning Continuun for HP Integrity and HP 9000," Aug. 2005, http://h71028.www7.hp.com/ERC/downloads/4AA0-1469ENW.pdf.

[39] T.F. Abdelzaher and N. Bhatti, "Web Server QoS Management by Adaptive Content Delivery," *Proc. Seventh Int'l Workshop Quality of Service (IWQoS '99),* 1999.

[40] J. Martin and A. Nilsson, "On Service Level Agreements for IP Networks," *Proc. IEEE INFOCOM '02,* pp. 855-863, 2002.

[41] IBM, "IBM Tiovli Monitoring," http://www.tivoli.com/, 2005.

[42] Hewlett-Packard, "HP OpenView," http://www.openview.hp.com/, 2005.

[43] K. McCloghrie and M.T. Rose, *Management Information Base for Network Management of TCP/IP-Based Internets: MIB-II,* IETF RFC 1213, Mar. 1991.

[44] "The Internet Traffic Archive," http://ita.ee.lbl.gov/, 2005.

Chang-Hao Tsai received the MS degree from the University of Michigan in 2003, where he is currently a PhD student in the Real-Time Computing Laboratory (RTCL) at the Electrical Engineering and Computer Science (EECS) Department. His current research interests include Internet services, cluster computing, distributed systems, virtual execution environments, and computer networks. He is a student member of the IEEE Computer Society.

Kang G. Shin received the BS degree in electronics engineering from Seoul National University in 1970 and both the MS and PhD degrees in electrical engineering from Cornell University, Ithaca, New York, in 1976 and 1978, respectively. He is the Kevin and Nancy O'Connor professor of computer science and the founding director of the Real-Time Computing Laboratory in the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, Michigan. He has held visiting positions at the US Airforce Flight Dynamics Laboratory; AT&T Bell Laboratories; Computer Science Division within the Department of Electrical Engineering and Computer Science at the University of California, Berkeley; International Computer Science Institute, Berkeley, California; IBM T.J. Watson Research Center; Software Engineering Institute at Carnegie Mellon University; and Hewlett-Packard Research Laboratories. He was general chair of the 2000 IEEE Real-Time Technology and Applications Symposium; and general chair of the Third ACM/USENIX International Conference on Mobile Systems Applications and Services (MobiSys '05). He also served in numerous technical program committees. He was also a distinguished visitor of the IEEE Computer Society, an editor of *IEEE Transactions on Parallel and Distributed Computing*, and an area editor of *International Journal of Time-Critical Computing Systems*, *Computer Networks*, and *ACM Transactions on Embedded Systems*. His current research focuses on QoS-sensitive networking and computing, as well as on embedded real-time OS, middleware, and applications, all with emphasis on timeliness and dependability. He has supervised the completion of 56 PhD theses and authored/coauthored more than 650 technical papers (more than 230 of which are in archival journals) and numerous book chapters in the areas of distributed real-time computing and control, computer networking, fault-tolerant computing, and intelligent manufacturing. He has coauthored (jointly with C.M. Krishna) the textbook *Real-Time Systems* (McGraw Hill, 1997). He has received a number of best paper awards, including the IEEE Communications Society William R. Bennett Prize Paper Award in 2003, and the Best Paper Award from the IWQoS in 2003. He has also received several institutional awards, including the Outstanding Achievement Award in 1999, Service Excellence Award in 2000, Distinguished Faculty Achievement Award in 2001, Stephen Attwood Award in 2004 from the University of Michigan, a Distinguished Alumni Award from the College of Engineering, Seoul National University, in 2002, 2003 IEEE RTC Technical Achievement Award, and 2006 Ho-Am Prize in Engineering. He is a fellow of the IEEE, the IEEE Computer Society, and the ACM, and a member of the Korean Academy of Engineering.

John Reumann received the PhD degree in computer science from the University of Michigan at Ann Arbor in 2003. Since then, he has worked for the IBM T.J. Watson Research Center and for Google's New York engineering group. He continues his work in distributed system software infrastructure at Google and serves in the Operating Systems and Distributed Systems Research Community. He has published several papers on distributed server control systems. He is a member of the IEEE.

Sharad Singhal received the BTech degree from the Indian Institute of Technology, Kanpur, India, and the MS and PhD degrees from Yale University. He is currently a distinguished technologist at Hewlett-Packard Laboratories. He has been involved in industrial research for more than 22 years, including two years at Bell Laboratories where he worked on speech coding and 12 years at Bellcore (now Telecordia) where he conducted and managed research in a number of areas including speech and video processing, neural networks, and middleware and personal communications services. Since 1997, he has been at HP Laboratories, where he has led teams that have developed techniques for monitoring and managing service level agreements; methods for controlling service quality in multitier applications, resource allocation, and assignment algorithms; and architectures for management of large-scale data centers. His current research interests include the application of control theory to systems management, policy-based system management, and large-scale management architectures. He holds 10 patents and has published more than 50 papers in a variety of refereed journals and conferences. He is a member of the IEEE and the Acoustical Society of America.