Resource Overbooking and Application Profiling in a Shared Internet Hosting Platform

BHUVAN URGAONKAR
The Penn State University
PRASHANT SHENOY
University of Massachusetts
and
TIMOTHY ROSCOE
ETH Zürich

In this article, we present techniques for provisioning CPU and network resources in shared Internet hosting platforms running potentially antagonistic third-party applications. The primary contribution of our work is to demonstrate the feasibility and benefits of overbooking resources in shared Internet platforms. Since an accurate estimate of an application's resource needs is necessary when overbooking resources, we present techniques to profile applications on dedicated nodes, possibly while in service, and use these profiles to guide the placement of application components onto shared nodes. We then propose techniques to overbook cluster resources in a controlled fashion. We outline an empirical appraoch to determine the degree of overbooking that allows a platform to achieve improvements in revenue while providing performance guarantees to Internet applications. We show how our techniques can be combined with commonly used QoS resource allocation mechanisms to provide application isolation and performance guarantees at run-time. We implement our techniques in a Linux cluster and evaluate them using common server applications. We find that the efficiency (and consequently revenue) benefits from controlled overbooking of resources can be dramatic. Specifically, we find that overbooking resources by as little as 1% we can increase the utilization of the cluster by a factor of two, and a 5% overbooking yields a 300–500% improvement, while still providing useful resource guarantees to applications.

Categories and Subject Descriptors: D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems*; D.4.8 [**Operating Systems**]: Performance—*Measurements, modeling and*

A portion of this research appeared in *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI'02)*, USENIX, Berkeley, CA, 2002, 239–254.
Authors' addresses: B. Urgaonkar, Penn State University, Department of Computer Science and Engineering, University Park, PA 16802; email: bhuvan@cse.psu.edu; P. Shenoy, University of Massachusetts, Department of Computer Science, Amherst, MA 01003; email: shenoy@cs.umass.edu; T. Roscoe, ETH Zürich, Department of Computer Science, IFW B 45.2, Haldeneggsteig 4, 8092 Zürich, Switzerland; email: troscoe@inf.ethz.ch.

prediction, stochastic analysis; I.6.4 [**Simulation and Modeling**]: Model Validation and Analysis; I.6.5 [**Simulation and Modeling**]: Model Development—*Modeling methodologies*

General Terms: Design, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Internet application, shared hosting platform, dedicated hosting platform, quality-of-service, yield management, resource overbooking, high percentile, profile, capsule, placement

ACM Reference Format:

Urgaonkar, B., Shenoy, P., and Roscoe, T. 2009. Resource overbooking and application profiling in a shared internet hosting platform. ACM Trans. Intern. Tech. 9, 1, Article 1 (February 2009), 45 pages. DOI = 10.1145/1462159.1462160 http://doi.acm.org/10.1145/1462159.1462160

1. INTRODUCTION AND MOTIVATION

Internet applications of various kinds burst onto the scene in the early to mid 1990s and revolutionized the way we conduct business and education, access news and other information, and seek various forms of entertainment. The explosive growth of the Web, advances in software technologies such as proxy caches, continuous growth in the capacities of servers as well as personal computers, and ever-increasing network availability have helped transform Internet applications into indispensable utilities in our personal, commercial, and educational lives. Consequently, Internet hosting platforms—large clusters of compute and storage servers often built using commodity hardware and software—have emerged as an important business. These hosting platforms have become an increasingly attractive alternative to traditional large multiprocessor servers for housing these applications, in part due to rapid advances in computing technologies and falling hardware prices.

This article addresses challenges in the design of a particular type of Internet hosting platform we call a *shared hosting platform*. This can be contrasted with a *dedicated hosting platform*, where either the entire cluster runs a single application (such as a Web search engine), or each individual processing element in the cluster is dedicated to a single application (as in the "managed hosting" services provided by some data centers). In contrast, shared hosting platforms run a large number of different third-part Internet applications (Web servers, streaming media servers, multi-player game servers, e-commerce applications, etc.), and the number of applications typically *exceeds the number of nodes in the cluster.* More specifically, each application runs on a subset of the nodes and these subsets can overlap with one another. Whereas dedicated hosting platforms are used for many niche applications that warrant their additional cost, the economic reasons of space, power, cooling, and cost make shared hosting platforms an attractive choice for many application hosting environments.

Shared hosting platforms imply a business relationship between the *platform provider* and the *application providers*: the latter pay the former for resources on the platform. In return, the platform provider gives some kind of guarantee of resource availability to applications [Roscoe and Lyles 2000].

Perhaps the central challenge in building such a shared hosting platform is resource management: the ability to reserve resources for individual applications, the ability to isolate applications from other misbehaving or overloaded applications, and the ability to provide performance guarantees to applications. Arguably, the widespread deployment of shared hosting platforms has been *hampered* by the lack of effective resource management mechanisms that meet these requirements. Consequently, most hosting platforms in use today adopt one of two approaches. The first avoids resource sharing altogether by employing a dedicated model. This delivers useful resources to application providers, but is expensive in machine resources. The second approach is to share resources in a best-effort manner among applications, which consequently receive no resource guarantees. While this is cheap in resources, the value delivered to application providers is limited. Consequently, both approaches imply an economic disincentive to deploy viable hosting platforms.

Some recent research efforts have proposed resource management mechanisms for shared hosting platforms [Aron et al. 2000; Chase and Doyle 2001; Appleby et al. 2001; Urgaonkar and Shenoy 2004b; Chen et al. 2005, 2006]. While these efforts take an initial step towards the design of effective shared hosting platforms, many challenges remain to be addressed. In particular, the focus of these bodies of research is on devising algorithms for dynamic resource provisioning and server consolidation in shared hosting platforms. The goal of dynamic resource provisioning is to vary resource allocations for the hosted applications to match their varying workloads. The goal of server consolidation is to minimize the server resources used to house the applications to reduce costs related to electricity, maintenance, etc. Despite this research, resource utilization in most hosting platforms continues to be abysmally low, resulting in wasted operational costs. An important reason for this wastage is the lack of research on revenue maximization techniques such as yield management that are well-studied in several other domains such as the airline industry [Davis 1994; Smith et al. 1992]. To enable these techniques as well as to achieve effective dynamic provisioning and server consolidation, it is crucial to devise mechanisms to accurately identify the resource requirements of the applications. Such mechanisms for inferring the resource requirements of applications have received little attention in the existing research on hosting platforms. In the absence of such mechanisms, a shared platform often has to resort to either over-provisioning resources (i.e., allocating more resources than needed by the applications) or provisioning to meet the worst-case resource needs. Such provisioning can result in significant wastage of resources [Chandra et al. 2003b]. As a result, the hosting platform may end up hosting fewer applications than it could, or allocating more server resources to host a set of applications than needed—this would result in an increase in the operational costs associated with electricity, cooling, and maintenance. Accurately characterizing the resource needs of the applications can allow a shared hosting platform to carefully multiplex its resources among the hosted applications and significantly improve its resource utilization, thereby improving its revenue. This is the context of the present work.

1.1 Research Contributions

The contribution of this article is threefold. First, we show how the resource requirements of an application can be derived using online profiling and modeling. Second, we demonstrate the efficiency benefits to the platform provider of *overbooking* resources on the platform, a form of yield management, and how this can be usefully done without adversely impacting the guarantees offered to application providers. Third, we show how untrusted and/or mutually antagonistic applications in the platform can be isolated from one another.

Automatic Derivation of QoS Requirements. Recent work on resource management mechanisms for clusters (e.g., Aron et al. [2000], Urgaonkar and Shenoy [2004b], Chase and Doyle [2001], Xu et al. [2004], and Appleby et al. [2001]) implicitly assumes that the resource requirements of an application are either known in advance or can be derived, but does not satisfactorily address the problem of how to determine these requirements. Several researchers have developed analytical models for deriving the resource requirements of hosted applications [Pradhan et al. 2002; Benani and Menasce 2005; Urgaonkar et al. 2005a; Cohen et al. 2004; Doyle et al. 2003]. These approaches, however, have proven to be of limited use in the context of shared hosting. Analytical models have been found to become unwieldy when they attempt to capture multiple resources within a server machine. Moreover, these models, often rooted in queuing theory, only capture steady-state resource requirements and limited workload regions (e.g., single bottleneck resource). The effectiveness of a resource management technique for a shared hosting platform is crucially dependent on the ability to reserve appropriate amount of resources for each hosted application—overestimating an application's resource needs can result in idling of resources, while underestimating them can degrade application performance.

A shared hosting platform can significantly enhance its utility to users by automatically deriving the quality-of-service (QoS) requirements of an application. Automatic derivation of QoS requirements involves (i) monitoring an application's resource usage, and (ii) using these statistics to derive QoS requirements that conform to the observed behavior.

In this article, we employ kernel-based profiling mechanisms to empirically monitor an application's resource usage and propose techniques to derive QoS requirements from this observed behavior. We then use these techniques to experimentally profile several server applications such as Web, streaming, game, and database servers. Our results show that the bursty resource usage of server applications makes it feasible to extract statistical multiplexing gains by overbooking resources on the hosting platform.

Revenue Maximization through Overbooking. The goal of the owner of a hosting platform is to maximize revenue, which implies that the cluster should strive to maximize the number of applications that can be housed on a given hardware configuration. We believe that a hosting platform can benefit from existing research in the area of Yield Management (YM) [Davis 1994] in this regard. YM is the process of understanding, anticipating, and reacting to

consumer behavior in order to maximize revenue. American Airlines, which was a pioneer in the innovation of YM systems, estimated that the utilization of YM increased its revenue by $1.4 billion between 1989 and 1991 [Smith et al. 1992]. The use of modern telephone and networking infrastructure allows airlines to monitor how seats are being reserved and use this information to *overbook* flights (i.e., sell more tickets than there are seats) or offer discounts when it appears as if seats will otherwise be vacant. Overbooking seats in an airplane improves profit because past observations of passenger behavior have demonstrated that the likelihood of all passengers arriving for a flight are very low—some passengers do not turn up for their scheduled flights due to delays or last-minute changes of plan. Therefore, by selling a few more tickets than the number of seats, an airline can improve the chances of its flights being full. The number of extra tickets is chosen so as to ensure that a passenger not getting his promised flight is a rare event.

A shared hosting platform can also benefit from such overbooking of resources. Overbooking resources in a shared hosting platform would be achieved by provisioning each application lower than its peak (worst-case) resource requirements. Whereas a passenger's requirement for a seat can be captured by a *binary* random variable (0 if the passenger does not turn up for his scheduled flight and 1 otherwise), an application's requirement for a temporal resource (like CPU or network bandwidth) over a period of time needs to be represented by a *continuous* random variable (ranging from 0 to the maximum available capacity for the purposes of this article).[1] Overbooking of a resource would yield improved utilization but would result in violations of the resource needs of one or more applications whenever the aggregate requirement of the applications hosted on the same machine exceeds the capacity of the machine. Excessive overbooking could cause a significant degradation in the performance of the application's clients. The resulting customer dissatisfaction could result in a loss of revenue for the application provider. The degree of overbooking should, therefore, be chosen such that such degradation occurs rarely. This is analogous to the rarity of the event that all the passengers on a flight turn up (causing some of them to be denied seats on the flight) when the degree of overbooking is properly chosen.

A well-designed hosting platform should be able to provide performance guarantees to applications even when overbooked, with the proviso that this guarantee is now probabilistic instead of deterministic (for instance, an application might be provided a 99% guarantee (0.99 probability) that its resource needs will be met). Since different applications have different tolerance to such overbooking (e.g., the latency requirements of a game server make it less tolerant to violations of performance guarantees than a Web server), an overbooking mechanism should take into account diverse application needs.

The primary contribution of this article is to demonstrate the feasibility and benefits of overbooking resources in shared hosting platforms. We propose

[1]Resources such as memory and disk bandwidth are known to behave differently and are beyond the scope of this article [Berger et al. 2003; Shenoy and Vin 1998; Zhang et al. 2005a, 2005b]. We conduct a discussion on how memory may be overbooked in a shared platform in Section 6.

techniques to overbook resources in a controlled fashion based on application resource needs. Although such overbooking can result in transient overloads where the aggregate resource demand temporarily exceeds capacity, our techniques limit the chances of transient overload of resources to predictably rare occasions, and provide useful performance guarantees to applications in the presence of overbooking.

The techniques we describe are general enough to work with many commonly used OS resource allocation mechanisms. Experimental results demonstrate that overbooking resources by amounts as small as 1% yields a factor of two increase in the number of applications supported by a given platform configuration, while a 5–10% overbooking yields a 300–500% increase in effective platform capacity. In general, we find that the more bursty the application resource needs, the higher are the benefits of resource overbooking. We also find that collocating CPU-bound and network-bound applications as well as bursty and non-bursty applications yields additional multiplexing gains when overbooking resources.

Placement and Isolation of Antagonistic Applications. In a shared hosting platform, it is assumed that third-party applications may be antagonistic to each other and/or the platform itself, either through malice or bugs. A hosting platform should address these issues by isolating applications from one another and preventing malicious, misbehaving, or overloaded applications from affecting the performance of other applications.

A third contribution of our work is to demonstrate how untrusted third-party applications can be isolated from one another in shared hosting platforms. This isolation takes two forms. Local to a machine, each processing node in the platform employs resource management techniques that "sandbox" applications by restricting the resources consumed by an application to its reserved share. Globally, the process of placement, whereby components of an application are assigned to individual processing nodes, can be constrained by externally imposed policies—for instance, by risk assessments made by the provider about each application, which may prevent an application from being collocated with certain other applications. Since a manual placement of applications onto nodes is infeasibly complex in large clusters, the design of automated placement techniques that allow a platform provider to exert sufficient control over the placement process is a key issue.

1.2 System Model and Terminology

The shared hosting platform assumed in our research consists of a cluster of *N nodes*, each of which consists of processor, memory, and storage resources as well as one or more network interfaces. Platform nodes are allowed to be heterogeneous with different amounts of these resources on each node. The nodes in the hosting platform are assumed to be interconnected by a high-speed LAN such as gigabit Ethernet. Each cluster node is assumed to run an operating system kernel that supports some notion of quality of service such as reservations or shares. Such mechanisms have been extensively studied over the past decade and many deployed commercial and open-source operating

systems such as Solaris [Sun98b 1998], IRIX [Sgi99 1999], Linux [Sundaram et al. 2000], and FreeBSD [Blanquer et al. 1999; Banga et al. 1999] already support such features.

In this article, we primarily focus on managing two resources—CPU and network interface bandwidth—in shared hosting platforms. The challenges of managing other resources in hosting environments, such as memory, disk bandwidth, and storage space, are beyond the scope of this article. Nevertheless, we believe the techniques developed here are also applicable to these other resources.

We use the term, *application*, for a complete service running on behalf of an application provider; since an application will frequently consist of multiple distributed components, we use the term *capsule* to refer to that component of an application running on a single node. Each application has at least one capsule, possibly more if the application is distributed. Capsules provide a useful abstraction for logically partitioning an application into subcomponents and for exerting control over the distribution of these components onto different nodes. To illustrate, consider an e-commerce application consisting of a Web server, a Java application server and a database server. If all three components need to be collocated on a single node, then the application will consist of a single capsule with all three components. On the other hand, if each component needs to be placed on a different node, then the application should be partitioned into three capsules. Depending on the number of its capsules, each application runs on a subset of the platform nodes and these subsets can overlap with one another, resulting in resource sharing.

The rest of this article is structured as follows. Section 2 discusses techniques for empirically deriving an application's resource needs, while Section 3 discusses our resource overbooking techniques and capsule placement strategies. We discuss implementation issues in Section 4 and present our experimental results in Section 5. In Section 6, we discuss related work. Finally, Section 7 presents concluding remarks.

2. AUTOMATIC DERIVATION OF APPLICATION QOS REQUIREMENTS

The first step in hosting a new application is to derive its resource requirements. While the problem of QoS-aware resource management has been studied extensively in the literature [Banga et al. 1999; Blanquer et al. 1999; Chandra et al. 2000; Duda and Cheriton 1999; Goyal et al. 1996a, 1996b; Jones et al. 1997; Lin et al. 1998; Leslie et al. 1996; Berger et al. 2003], the problem of *how much* resource to allocate to each application has received relatively little attention. We assume that an application provider wishes to provide some kind of guarantee on the QoS it will offer its clients. For example, a Web server may wish to ensure that the response time experienced by its clients is below 1 second on average for requests of a certain type. Similarly, a streaming media server may wish to ensure that the streaming rate it provides to its clients is high enough so that they do not experience more than one playback discontinuity during a two-hour-long movie. In order to provide such guarantees, the application provider needs to determine the amount of resources that the hosting platform

should provision for it, so its QoS requirements may be met. The problem of translating the QoS requirements of an application to its resource requirements is referred to as *application modeling*. Existing research on application modeling has used analytical techniques based on queuing-theory [Menasce et al. 2004; Benani and Menasce 2005; Urgaonkar et al. 2005a; Doyle et al. 2003; Kamra et al. 2004] or machine learning [Cohen et al. 2004] as well as less formal measurement-based techniques [Pradhan et al. 2002]. To the best of our knowledge, these models only allow the determination of the *average* and the *worst-case* resource requirements of an application. They neither provide a detailed description of the resource usage distribution of an application nor any information on the impact of resource overbooking on application performance. Therefore, while being suitable for use in existing techniques for provisioning based on worst-case needs, they do not provide sufficient information to enable controlled overbooking of resources. Furthermore, all of these models have to be necessarily developed by the application provider, since developing them requires an intimate knowledge of the software architecture of the application. Consequently, the resource requirements inferred using such models leaves little room for a shared hosting platform to make efficient decisions regarding placing the applications in a manner that will allow it to reduce its costs and improve its revenue.

In this section, we present an alternative approach for application modeling. We propose a measurement-based technique to automatically derive the resource requirements of an application. As opposed to existing models, our modeling technique can be used by a hosting platform to infer the resource requirements of an application without explicit knowledge of the application's software architecture. The hosting platform can essentially treat an application as a "black-box". At the same time, it relieves the application provider of the task of modeling its application to provide the hosting platform with its resource requirements. Our approach is similar in spirit to black-box [Kelly et al. 2004] and gray-box [Arpaci-Dusseau and Arpaci-Dusseau 2001; Burnett et al. 2002] modeling techniques that have been used in other systems domains. In the remainder of the article, we will use the terms resource requirements and QoS requirements interchangeably. Deriving the QoS requirements is a two-step process: (i) we first use profiling techniques to monitor application behavior, and (ii) we then use our empirical measurements to derive QoS requirements that conform to the observed behavior.

2.1 Application QoS Requirements: Definitions

The QoS requirements of an application are defined on a per-capsule basis. For each capsule, the QoS requirements specify the intrinsic rate of resource usage, the variability in the resource usage, the time period over which the capsule desires resource guarantees, and the level of overbooking that the application (capsule) is willing to tolerate. As explained earlier, in this article, we are concerned with two key resources, namely CPU and network interface bandwidth. For each of these resources, we define the QoS requirements along the above dimensions in an OS-independent manner. In Section 4.1, we show how to map

these requirements to various OS-specific resource management mechanisms that have been developed.

More formally, we represent the QoS requirements of an application capsule by a quintuple $(\sigma, \rho, \tau, U, O)$:

—*Token Bucket Parameters* (σ, ρ). We capture the basic resource requirements of a capsule by modeling resource usage as a token bucket (σ, ρ) [Tang and Tai 1999]. The parameter σ denotes the intrinsic rate of resource consumption, while ρ denotes the variability in the resource consumption. More specifically, σ denotes the rate at which the capsule consumes CPU cycles or network interface bandwidth, while ρ captures the maximum burst size. By definition, a token bucket bounds the resource usage of the capsule to $\sigma \cdot t + \rho$ over any interval t.

—*Period* τ. The third parameter τ denotes the time period over which the capsule desires guarantees on resource availability. Put another way, the system should strive to meet the QoS requirements of the capsule over each interval of length τ. The smaller the value of τ, the more stringent are the desired guarantees (since the capsule needs to be guaranteed resources over a finer time scale). In particular, for the above token bucket parameters, the capsule requires that it be allocated at least $\sigma \cdot \tau + \rho$ resources every τ time units.

—*Usage Distribution* U. While the token bucket parameters succinctly capture the capsule's resource requirements, they are not sufficiently expressive by themselves to denote the QoS requirements in the presence of overbooking. Consequently, we use two additional parameters—U and O—to specify resource requirements in the presence of overbooking. The first parameter U denotes the probability distribution of resource usage. Note that U is a more detailed specification of resource usage than the token bucket parameters (σ, ρ), and indicates the probability with which the capsule is likely to use a certain fraction of the resource (i.e., $U(x)$ is the probability that the capsule uses a fraction x of the resource, $0 \leq x \leq 1$). A probability distribution of resource usage is necessary so that the hosting platform can provide (quantifiable) probabilistic guarantees even in the presence of overbooking.

—*Overbooking Tolerance* O. The parameter O is the overbooking tolerance of the capsule. It specifies the probability with which the capsule's requirements may be violated due to resource overbooking (by providing it with less resources than the required amount). Thus, the overbooking tolerance indicates the minimum level of service that is acceptable to the capsule. To illustrate, if $O = 0.01$, the capsule's resource requirements should be met 99% of the time (or with a probability of 0.99 in each interval τ).

In our prior work [Urgaonkar et al. 2002], we had assumed that the parameters τ and O are specified by the application provider. This was assumed to be based on a contract between the platform provider and the application provider (e.g., the more the application provider is willing to pay for resources, the stronger are the provided guarantees), or on the particular characteristics of the application (e.g., a streaming media server requires more stringent guarantees and is less tolerant to violations of these guarantees than a Web server).

In this article, we also develop an empirical approach that a hosting platform can use to determine the parameters τ and O that would help an application meet its QoS goals. We describe these mechanisms next.

2.2 Kernel-Based Profiling of Resource Usage

Our techniques for empirically deriving the QoS requirements of an application rely on profiling mechanisms that monitor application behavior. Recently, a number of application profiling mechanisms ranging from OS-kernel-based profiling [Anderson et al. 1997] to run-time profiling using specially linked libraries [Shende et al. 1998] have been proposed.

We use kernel-based profiling mechanisms in the context of shared hosting platforms, for a number of reasons. First, being kernel-based, these mechanisms work with any application and require no changes to the application at the source or binary levels. This is especially important in hosting environments where the platform provider may have little or no access to third-party applications. Second, accurate estimation of an application's resource needs requires detailed information about when and how much resources are used by the application at a fine time-scale. Whereas detailed resource allocation information is difficult to obtain using application-level techniques, kernel-based techniques can provide precise information about various kernel events such as CPU scheduling instances and network packet transmissions times.

The profiling process involves running the application on a set of *isolated* platform nodes (the number of nodes required for profiling depends on the number of capsules). By isolated, we mean that each node runs only the minimum number of system services necessary for executing the application and no other applications are run on these nodes during the profiling process—such isolation is necessary to minimize interference from unrelated tasks when determining the application's resource usage. The application is then subjected to a realistic workload, and the kernel profiling mechanism is used to track its resource usage. It is important to emphasize that the workload used during profiling should be both realistic and representative of real-world workloads. While techniques for generating such realistic workloads are orthogonal to our current research, we note that a number of different workload-generation techniques exist, ranging from trace replay of actual workloads to running the application in a "live" setting, and from the use of synthetic workload generators to the use of well-known benchmarks. Any such technique suffices for our purpose as long as it realistically emulates real-world conditions, although we note that, from a business perspective, running the application "for real" on an isolated machine to obtain a profile may be preferable to other workload generations techniques.

We use the Linux trace toolkit as our kernel profiling mechanism [LTT02]. The toolkit provides flexible, low-overhead mechanisms to trace a variety of kernel events such as system call invocations, process, memory, file system, and network operations. The user can specify the specific kernel events of interest as well as the processes that are being profiled to selectively log events. For our purposes, it is sufficient to monitor CPU and network activity of capsule processes—we monitor CPU scheduling instances (the time instants at which

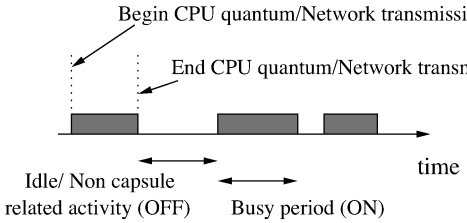


Fig. 1. An example of an On-Off trace.

capsule processes get scheduled and the corresponding quantum durations) as well as network transmission times and packet sizes. Given such a trace of CPU and network activity, we now discuss the derivation of the capsule's QoS requirements.

2.3 Empirical Derivation of the QoS Requirements

We use the trace of kernel events obtained from the profiling process to model CPU and network activity as a simple On-Off process. This is achieved by examining the time at which each event occurs and its duration and deriving a sequence of busy (On) and idle (Off) periods from this information (see Figure 1). This trace of busy and idle periods can then be used to derive both the resource usage distribution U as well as the token bucket parameters (σ, ρ).

2.3.1 *Determining the Usage Distribution U.* Recall that the usage distribution U denotes the probability with which the capsule uses a certain fraction of the resource. To derive U, we simply partition the trace into measurement intervals of length $\mathcal{I}$ and measure the fraction of time for which the capsule was busy in each such interval. This value, which represents the fractional resource usage in that interval, is histogrammed and then each bucket is normalized with respect to the number of measurement intervals $\mathcal{I}$ in the trace to obtain the probability distribution U. Figure 2(a) illustrates this process.

2.3.2 *Determining τ and O.* We take an empirical approach for determining the parameters τ and O. Our reasons for choosing such an approach are similar to those that we described earlier concerning the shortcomings of analytical approaches in accurately capturing the distributions of resource requirements needed for achieving controlled overbooking in a shared hosting platform.

Qualitatively, higher values of τ and O allow the platform to achieve higher utilization while resulting in more instances of QoS violations for the application. The goal of the hosting platform is to determine the parameters τ and O that would maximize its revenue. To capture this tradeoff quantitatively, the platform needs to be able to compare the improvement in its revenue due to higher utilization with the possible loss in its revenue due to QoS violations. We assume that an application provider specifies a bound on an application-specific QoS metric (such as an upper bound on the average response time or a lower bound on the throughput) that it requires the hosting platform to guarantee. We also assume that the application provider and the hosting platform agree upon

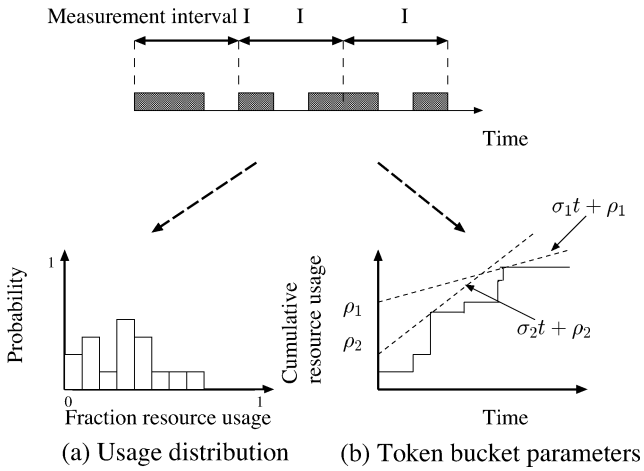


Fig. 2. Derivation of the usage distribution and token bucket parameters.

a penalty that the platform must pay whenever QoS violations occur despite the application's workload conforming to its expected intensity.

We conduct a simple search over reasonable values for τ and O to estimate how the expected revenue for the platform would vary with τ and O. For each chosen pair of τ and O, we provision the concerned resource (CPU or network bandwidth) corresponding to the $(100-O)$th percentile of the distribution U derived earlier—this represents the minimum amount of the resource that would be available to the application if we were to overbook the resource by $O\%$. We then re-subject the application to the workload used during offline profiling and measure the expected revenue (taking into account penalties due to QoS violations). Our search space is manageably small. First, we discretize the search space and only consider integral values of τ (in seconds) and O (in multiples of 1%). Second, the time period over which Internet server applications desire resource guarantees is at most a few seconds. Therefore, we need to search over a small set of values of τ. Finally, we use a binary search over the range [0%, 50%] to determine the value of O which is expected to provide the peak revenue—a smaller degree of overbooking would not fully exploit statistical multiplexing gains whereas a larger degree of overbooking would cause enough QoS violations to offset the improvement in revenue due to higher utilization.

Deriving Token Bucket Parameters (σ, ρ). Recall that a token bucket limits the resource usage of a capsule to $\sigma \cdot t + \rho$ over any interval t. A given On-Off trace can have, in general, many (σ, ρ) pairs that satisfy this bound. To intuitively understand why, let us compute the cumulative resource usage for the capsule over time. The cumulative resource usage is simply the total resource consumption thus far and is computed by incrementing the cumulative usage after each ON period. Thus, the cumulative resource usage is a step function as depicted in Figure 2(b). Our objective is to find a line $\sigma \cdot t + \rho$ that bounds the cumulative resource usage; the slope of this line is the token bucket rate σ and its Y-intercept is the burst size ρ. As shown in Figure 2(b), there are in general many such curves, all of which are valid descriptions of the observed resource usage.

Several algorithms that mechanically compute all valid (σ, ρ) pairs for a given On-Off trace have been proposed recently. We use a variant of one such algorithm [Tang and Tai 1999] in our research—for each On-Off trace, the algorithm produces a range of σ values (i.e., $[\sigma_{min}, \sigma_{max}]$) that constitute valid token bucket rates for observed behavior. For each σ within this range, the algorithm also computes the corresponding burst size ρ. Although any pair within this range conforms to the observed behavior, the choice of a particular (σ, ρ) has important practical implications.

Since the overbooking tolerance O for the capsule is given, we can use O to choose a particular (σ, ρ) pair. To illustrate, if $O = 0.05$, the capsule needs must be met 95% of the time, which can be achieved by reserving resources corresponding to the 95th percentile of the usage distribution. Consequently, a good policy for shared hosting platforms is to *pick a* σ *that corresponds to the* $(1 - O) * 100$th *percentile of the resource usage distribution* U, and to pick the corresponding ρ as computed by the above algorithm. This ensures that we provision resources based on a high percentile of the capsule's needs and that this percentile is chosen based on the specified overbooking tolerance O.

2.4 Profiling Server Applications: Experimental Results

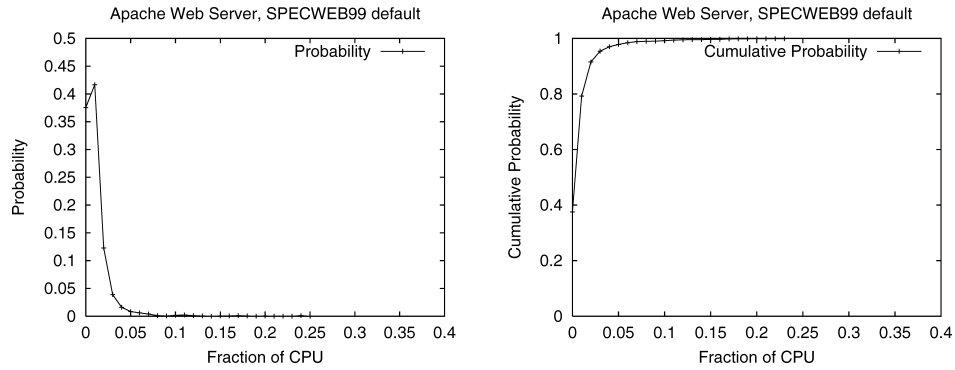
In this section, we profile several commonly-used server applications to illustrate the process of deriving an application's QoS requirements. Our experimentally derived profiles not only illustrate the inherent nature of various server application but also demonstrate the utility and benefits of resource overbooking in shared hosting platforms.

The test bed for our profiling experiments consists of a cluster of five Dell Poweredge 1550 servers, each with a 966 MHz Pentium III processor and 512 MB memory running Red Hat Linux 7.0. All servers runs the 2.2.17 version of the Linux kernel patched with the Linux trace toolkit version 0.9.5, and are connected by 100 Mb/s Ethernet links to a Dell PowerConnect (model no. 5012) Ethernet switch.

To profile an application, we run it on one of our servers and use the remaining servers to generate the workload for profiling. We assume that all machines are lightly loaded and that all nonessential system services (e.g., mail services, X windows server) are turned off to prevent interference during profiling. We profile the following server applications in our experiments:

—*Apache Web Server.* We use the SPECWeb99 benchmark [SPECWeb99] to generate the workload for the Apache Web server (version 1.3.24). The SPECWeb benchmark allows control along two dimensions—the number of concurrent clients and the percentage of dynamic (cgi-bin) HTTP requests. We vary both parameters to study their impact on Apache's resource needs.

—*MPEG Streaming Media Server.* We use a home-grown streaming server to stream MPEG-1 video files to multiple concurrent clients over UDP. Each client in our experiment requests a 15-minute-long variable bit rate MPEG-1 video with a mean bit rate of 1.5 Mb/s. We vary the number of concurrent clients and study its impact on the resource usage at the server.



(a) Probability distribution (PDF) (b) Cumulative distribution function (CDF)

Fig. 3. Profile of the Apache Web server using the default SPECWeb99 configuration.

—*Quake Game Server.* We use the publicly available Linux Quake server to understand the resource usage of a multi-player game server; our experiments use the standard version of Quake I—a popular multi-player game on the Internet. The client workload is generated using a bot—an autonomous software program that emulates a human player. We use the publicly available "terminator" bot to emulate each player; we vary the number of concurrent players connected to the server and study its impact on the resource usage.

—*PostgreSQL Database Server.* We profile the postgreSQL database server (version 7.2.1) using the *pgbench 1.2* benchmark. This benchmark is part of the postgreSQL distribution and emulates the TPC-B transactional benchmark [pgbench 2002]. The benchmark provides control over the number of concurrent clients as well as the number of transactions performed by each client. We vary both parameters and study their impact on the resource usage of the database server.

We now present some results from our profiling study.

Figure 3(a) depicts the CPU usage distribution of the Apache Web server obtained using the default settings of the SPECWeb99 benchmark (50 concurrent clients, 30% dynamic cgi-bin requests). Figure 3(b) plots the corresponding cumulative distribution function (CDF) of the resource usage. As shown in the figure (and summarized in Table I), the worst-case CPU usage (100th profile) is 25% of CPU capacity. Furthermore, the 99th and the 95th percentiles of CPU usage are 10 and 4% of capacity, respectively. These results indicate that CPU usage is bursty in nature and that the worst-case requirements are significantly higher than a high percentile of the usage. Consequently, under provisioning (i.e., overbooking) by a mere 1% reduces the CPU requirements of Apache by a factor of 2.5, while overbooking by 5% yields a factor of 6.25 reduction (implying that 2.5 and 6.25 times as many Web servers can be supported when provisioning based on the 99th and 95th percentiles, respectively, instead of the 100th profile). Thus, even small amounts of overbooking can potentially yield significant increases in platform capacity. Figure 4 depicts the possible valid (σ, ρ) pairs for Apache's CPU usage. Depending on the specified overbooking tolerance

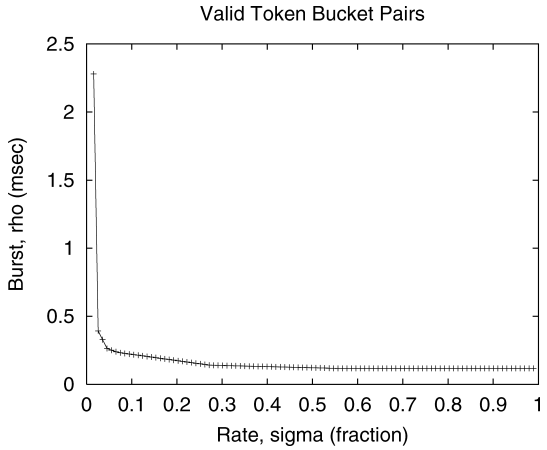


Fig. 4. Token bucket parameters describing the CPU requirements of the Apache Web server using the default SPECWeb99 configuration.

Table I. Summary of Profiles

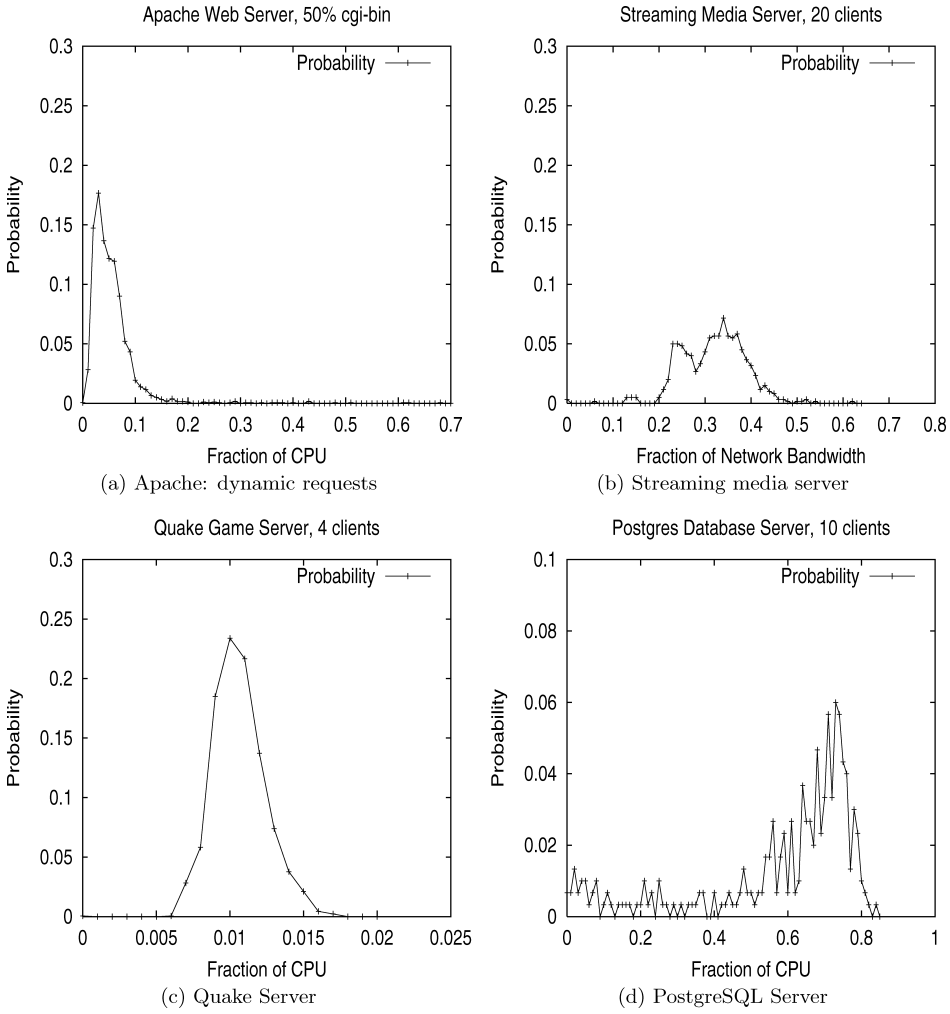
Application	Resource	Res. usage at percentile 100^{th}	99^{th}	95^{th}	(σ, ρ) for $O = 0.01$
WS, default	CPU	0.25	0.10	0.04	(0.10, 0.218)
WS, 50% dynamic	CPU	0.69	0.29	0.12	(0.29, 0.382)
SMS, k = 4	Net	0.19	0.16	0.11	(0.16, 1.89)
SMS, k = 20	Net	0.63	0.49	0.43	(0.49, 6.27)
GS, k = 2	CPU	0.011	0.010	0.009	(0.010, 0.00099)
GS, k = 4	CPU	0.018	0.016	0.014	(0.016, 0.00163)
DBS, k = 1 (def)	CPU	0.33	0.27	0.20	(0.27, 0.184)
DBS, k = 10	CPU	0.85	0.81	0.79	(0.81, 0.130)

Although we profiled both CPU and network usage for each application, we only present results for the more constraining resource. Abbreviations: WS = Apache, SMS = streaming media server, GS = Quake game server, DBS = database server, k = num. clients.

O, we can set σ to an appropriate percentile of the usage distribution U, and the corresponding ρ can then be chosen using this figure.

Figures 5(a)–(d) depict the CPU or network bandwidth distributions, as appropriate, for various server applications. Specifically, the figure shows the usage distribution for the Apache Web server with 50% dynamic SPECWeb requests, the streaming media server with 20 concurrent clients, the Quake game server with 4 clients and the postgreSQL server with 10 clients. Table I summarizes our results and also presents profiles for several additional scenarios (only a small representative subset of the three dozen profiles obtained from our experiments are presented here). Table I also lists the worst-case resource needs as well as the 99th and the 95th percentile of the resource usage.

Together, Figure 5 and Table I demonstrate that all server applications exhibit burstiness in their resource usage, albeit to different degrees. This burstiness causes the worst-case resource needs to be significantly higher than a high percentile of the usage distribution. Consequently, we find that the 99th percentile is smaller by a factor of 1.1-2.5, while the 95th percentile yields a



(a) Apache: dynamic requests
(b) Streaming media server
(c) Quake Server
(d) PostgreSQL Server

Fig. 5. Profiles of Various Server Applications.

factor of 1.3-6.25 reduction when compared to the 100th percentile. Together, these results illustrate the potential gains that can be realized by overbooking resources in shared hosting platforms.

3. RESOURCE OVERBOOKING AND CAPSULE PLACEMENT IN HOSTING PLATFORMS

Having derived the QoS requirements of each capsule, the next step is to determine which platform node will run each capsule. Several considerations arise when making such placement decisions. First, since platform resources are being overbooked, the platform should ensure that the QoS requirements of a capsule will be met even in the presence of overbooking. Second, since multiple nodes may have the resources necessary to house each application capsule,

the platform will need to pick a specific mapping from the set of feasible mappings. This choice will be constrained by issues such as trust among competing applications. In this section, we present techniques for overbooking platform resources in a controlled manner. The aim is to ensure that: (i) the QoS requirements of the application are satisfied and (ii) overbooking tolerances as well as external policy constraints are taken into account while making placement decisions.

3.1 Resource Overbooking Techniques

A platform node can accept a new application capsule so long as the resource requirements of existing capsules are not violated, and sufficient unused resources exist to meet the requirements of the new capsule. However, if the node resources are overbooked, another requirement is added: the overbooking tolerances of individual capsules already placed on the node should not be exceeded as a result of accepting the new capsule. Verifying these conditions involves two tests:

Test 1: Resource requirements of the new and existing capsules can be met. To verify that a node can meet the requirements of all capsules, we simply sum the requirements of individual capsules and ensure that the aggregate requirements do not exceed node capacity. For each capsule i on the node, the QoS parameters (σ_i,ρ_i) and τ_i require that the capsule be allocated $(\sigma_i \cdot \tau_i + \rho_i)$ resources in each interval of duration τ_i. Further, since the capsule has an overbooking tolerance O_i, in the worst case, the node can allocate only $(\sigma_i \cdot \tau_i + \rho_i) * (1 - O_i)$ resources and yet satisfy the capsule needs (thus, the overbooking tolerance represents the fraction by which the allocation may be reduced if the node saturates due to overbooking). Consequently, even in the worst case scenario, the resource requirements of all capsules can be met so long as the total resource requirements do not exceed the capacity:

$$\sum_{i=1}^{k+1}(\sigma_i \cdot \tau_{min} + \rho_i) \cdot (1 - O_i) \leq C \cdot \tau_{min}, \tag{1}$$

where C denotes the CPU or network interface capacity on the node, k denotes the number of existing capsules on the node, $k + 1$ is the new capsule, and $\tau_{min} = \min(\tau_1, \tau_2, \ldots, \tau_{k+1})$ is the period τ for the capsule that desires the most stringent guarantees.

Test 2: Overbooking tolerances of all capsules are met. The overbooking tolerance of a capsule is met only if the total amount of overbooking is smaller than its specified tolerance. To compute the aggregate overbooking on a node, we must first compute the total resource usage on the node. Since the usage distributions U_i of individual capsules are known, the total resource on a node is simply the sum of the individual usages. That is, $Y = \sum_{i=1}^{k+1} U_i$, where Y denotes the of aggregate resource usage distribution on the node. Assuming each U_i is independent, the resulting distribution Y can be computed from

elementary probability theory.[2] Given the total resource usage distribution Y, the probability that the total demand exceeds the node capacity should be less than the overbooking tolerance for every capsule, that is,

$$Pr(Y > C) \leq O_i \quad \forall i, \tag{2}$$

where C denotes the CPU or network capacity on the node. Rather than verifying this condition for each individual capsule, it suffices to do so for the least-tolerance capsule. That is,

$$Pr(Y > C) \leq \min(O_1, O_2, \ldots, O_{k+1}), \tag{3}$$

where $Pr(Y > C) = \sum_{x=C}^{\infty} Y(x)$. Note that Eq. (3) enables a platform to provide a probabilistic guarantee that a capsule's QoS requirements will be met at least $(1 - O_{min}) \times 100\%$ of the time.

Equations (1) and (3) can easily handle heterogeneity in nodes by using appropriate C values for the CPU and network capacities on each node. A new capsule can be placed on a node if Eqs. (1) and (3) are satisfied for both the CPU and network interface. Since multiple nodes may satisfy a capsule's CPU and network requirements, especially at low and moderate utilizations, we need to devise policies to choose a node from the set of all feasible nodes for the capsule. We discuss this issue next.

3.2 Capsule Placement Algorithms

Consider an application with m capsules that needs to be placed on a shared hosting platform with N nodes. For each of the m capsules, we can determine the set of *feasible* platform nodes. A feasible node is one that can satisfy the capsule's resource requirements (i.e., satisfies Eqs. (1) and (3) for both the CPU and network requirements). The platform must then pick a feasible node for each capsule such that all m capsules can be placed on the platform, with the constraint that no two capsules can be placed on the same node (since, by definition, two capsules from the same application are not collocated).

The placement of capsules onto nodes subject to the above constraint can be handled as follows. We model the placement problem using a graph that contains a vertex for each of the m capsules and N nodes. We add an edge from a capsule to a node if that node is a feasible node for the capsule (i.e., has sufficient resources to house the application). The result is a bipartite graph where each edges connects a capsule to a node. Figure 6 illustrates such a graph with three capsules and four nodes. As shown in the figure, determining an appropriate placement is a non-trivial problem since the placement decision for one capsule can impact the placement of other capsules. In the above figure, for instance, placing either capsule 2 or 3 onto node 3 eliminates any possible placement for capsule 1 (which has node 3 as its only feasible node). Multiple

[2]This is done using the z-transform. The z-transform of a random variable U is the polynomial $Z(U) = a_0 + za_1 + z^2a_2 + \cdots$ where the coefficient of the ith term represents the probability that the random variable equals i (i.e., $U(i)$). If $U_1, U_2, \ldots, U_{k+1}$ are $k+1$ independent random variables, and $Y = \sum_{i=1}^{k+1} U_i$, then $Z(Y) = \prod_{i=1}^{k+1} Z(U_i)$. The distribution of Y can then be computed using a polynomial multiplication of the z-transforms of $U_1, U_2, \ldots, U_{k+1}$ [Papoulis and Pillai 2002].

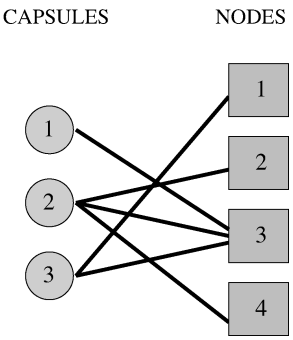


Fig. 6. A bipartite graph indicating which capsules can be placed on which nodes.

such constraints may exist, all of which will need to be taken into account when determining the final placement.

Given such a graph, we use the following algorithm to determine a placement. This algorithm solves the maximum bipartite matching problem [Cormen et al. 1991] on this graph. If it finds a maximum matching of size m, then it places the capsules on the nodes that they are associated with in the matching. If, however, the maximum matching is smaller than m, then there is at least one capsule that cannot be placed. In this case, the algorithm terminates declaring that no placement exists for the application.

LEMMA 1. *An application with m capsules can be placed on a hosting platform if, and only if, there is a matching of size m in the bipartite graph G modeling its placement on the platform.*

PROOF.
($\Rightarrow$) A matching of size m in G specifies a one-to-one correspondence between capsules and nodes. Since edges connote feasibility, this correspondence is a valid placement.
($\Leftarrow$) If G admits no matching of size m, then any placement of capsules on nodes must end up with distinct capsules sharing the same node. This means that the application cannot be placed without violating the capsule placement restriction. □

The run-time of this algorithm is given by the time to solve the maximal bipartite matching problem [Cormen et al. 1991], that is, $O((n + M)^3)$.

When there are multiple nodes that a capsule may be placed on, which node should we select? As we will show in Section 5.2, the choice of a particular feasible node can have important implications of the total number of applications supported by the cluster. As a baseline, we use a policy that picks a node uniformly at random when multiple feasible nodes exist. We consider three other policies, in addition to random, for making this decision. The first policy is best-fit, where we choose the feasible node that has the least unused resources (i.e., constitutes the best fit for the capsule). The second policy is worst-fit, where we place the capsule onto the feasible node with the most unused resources. In general, the unused network and CPU capacities on a node may be different, and

similarly, the capsule may request different amounts of CPU and network resources. Consequently, defining the best and worst fits for the capsule must take into account the unused capacities on *both* resources—we currently do so by simply considering the mean unused capacity across the two resources and compare it to the mean requirements across the two resources to determine the "fit".

A third policy is to place a capsule onto a node that has other capsules with similar overbooking tolerances. Since a node must always meet the requirements of its least tolerant capsule per Eq. (3), collocating capsules with similar overbooking tolerances permits the platform provider to maximize the amount of resource overbooking in the platform. For instance, placing a capsule with a tolerance of 0.01 onto a node that has an existing capsule with $O = 0.05$ reduces the maximum permissible overbooking on that node to 1% (since $O_{min} = \min(0.01, 0.05) = 0.01$). On the other hand, placing this less-tolerant capsule on another node may allow future, more tolerant capsules to be placed onto this node, thereby allowing nodes resources to be overbooked to a greater extent. We experimentally compare the effectiveness of these three policies in Section 5.2.

3.3 Policy Constraints on Capsule Placement

Whereas the strategies outlined in the previous section take QoS requirements into account while making placement decisions, they do not consider externally imposed policies that might constrain placement. For example, a platform provider might refuse to collocate capsules from applications owned by competing providers. Alternatively, the decision as to whether to collocate application capsules might be a quantitative one, involving some model of risk assessment.

To capture these notions, we quantify the "trust" between a newly arriving application and the existing k applications using a trust vector $<T_1, T_2, \ldots, T_k>$. Essentially, the vector specifies trust between applications in a pair-wise fashion; the ith element of the vector, T_i, denotes the trust between the new application and application i. T_i can vary between 0 and 1 depending on the level of trust between the two applications—a value of 0 indicates no trust whatsoever, a 1 indicates complete trust, and intermediate values indicate varying degrees of trust. An application capsule should not be collocated with a capsule with $T_i = 0$. In general, application capsules should be placed on nodes containing capsules with larger trust values.

3.3.1 *Dynamic Determination of Trust Values.* In practice, a platform may have little or no basis for determining the trust values between a newly arriving application and the already placed applications. We envision dynamically varying trust values based on measurements of resource usage interference between co-located applications. A shared platform should allow application providers to choose that their application be hosted on dedicated server(s). Such applications would be guaranteed to be immune to any resource assignment violations that overbooking may result in. Such dedicated hosting would, however, come at the cost of higher price of hosting.

Multiple approaches are possible for associating trust values with applications willing to be co-located with other applications. A simple approach is to

assume any newly arriving application to be completely trustworthy and update its trust values (and those of other applications for it) dynamically, as described below. A more sophisticated approach would consider the burstiness in an application resource needs as an indicator of how much other applications may trust it. Intuitively, already placed applications should have lower trust values for more bursty applications.

Our platform would employ dynamic measurements to keep track of changing resource needs of applications. Details of this mechanism appear in Section 3.4. These measurements would be used to update the mutual trust values of co-located applications. Applications whose resource usage behavior deviates from their expected profiles would decrease in how much other applications should trust them. Once such updated trust values have been determined, the platform can employ the enhanced placement algorithm described below for determining where to host various capsules.

3.3.2 *Trust-Aware Placement Mechanisms.* The policies outlined in the previous section can be modified to account for this notion of potentially antagonistic applications (as determined by an external policy). To do so, we enhance the bipartite graph with a weight on each edge. The weight of an edge is the trust value of the least trust-worthy application capsule on that node and the current application. Edges with a weight 0 are deleted. Given the resulting bipartite graph, we need to pick a placement that attempts to maximize the sum of the weights of the chosen edges (which ensures that capsules get placed onto nodes running applications that are trusted to a greater extent). The resulting placement decisions are driven by two considerations: (i) metrics such as best-fit, worst-fit or the overbooking tolerance (which impacts the effective platform capacity), and (ii) the weight of the edges (which determines the level of trust between collocated capsules). Such decisions can be made by computing a weighted sum of the two metrics—namely the nature of the "fit" and the weight of the edge—and picking a feasible node with the maximum weighted sum. Thus, we can ensure that external policy constraints are taken into account when making placement decisions.

We now outline an algorithm that a hosting platform can use to determine the "most preferred" placement. We reduce our placement problem to a well-known graph-theoretic problem. The first step in our reduction consists of modifying the weights on the edges of our feasibility graph to ensure that *lower weights mean higher preference*. This os done simply by replacing the weight of each edge by its reciprocal. As an example, see the graph on the left of Figure 7 wherein capsule $C1$ prefers node $N2$ more than node $N1$, etc.

The placement problem in such scenarios is to find a maximum matching of minimum weight in this weighted graph. This placement problem reduces to the *Minimum-Weight Perfect Matching Problem*.

Minimum-Weight Perfect Matching (MWPM). A *perfect matching* in a graph G is a subset of edges that "touch" each vertex exactly once. Given a real weight c_e for each edge e, the **MWPM** requires one to find a perfect matching M whose weight, $\sum_{e \in M} c_e$, is minimum.

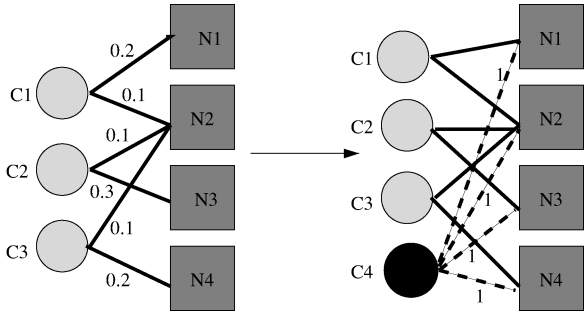


Fig. 7. Reducing Minimum-Weight Maximum Matching to Minimum-Weight Perfect Matching.

The reduction works as follows: By normalization, we may assume that each weight ω in the feasibility graph G lies in the range $0 \leq \omega \leq 1$ and that the weights sum to 1. Let there be m capsules and $N \geq m$ nodes.[3] Create an augmented graph $\widetilde{G}$ by adding $N - m$ new, *dummy* capsules and unit-weight edges connecting each with *all* nodes. Figure 7 exemplifies this reduction. The left-hand graph G shows the normalized preferences of the capsules $C1, C2, C3$ for their feasible nodes. The right-hand graph $\widetilde{G}$ adds a new capsule, $C4$, to equalize the numbers of capsules and nodes, and adds unit-weight edges between $C4$ and all nodes. (Since the weights of the original edges do not change, they are not shown in $\widetilde{G}$.)

LEMMA 2. *A matching of size m and cost c exists in the feasibility graph G of an application with m capsules and a cluster with $N \geq m$ nodes if, and only if, a perfect matching of cost $(c + N - m)$ exists in the augmented graph $\widetilde{G}$.*

PROOF.
($\Rightarrow$) Given a matching M of size m and cost c in G, we construct a perfect matching $\widetilde{M}$ in $\widetilde{G}$ as follows. $\widetilde{M}$ contains all edges in M, in addition to edges that "touch" each dummy capsule. To choose these latter edges, we consider dummy capsules one by one (in any order), and, for each, we add to $\widetilde{M}$ an edge connecting it to any as-yet "untouched" node. Since G admits a matching of size m, and since each dummy capsule connects to all N nodes, $\widetilde{G}$ is certain to have a size-N (hence, perfect) matching. Further, since each edge that "touches" a dummy capsule has unit weight, and there are $(N - m)$ such edges, the cost of $\widetilde{M}$ is $c + (N - m)$.
($\Leftarrow$) Let $\widetilde{G}$ admit a perfect matching $\widetilde{M}$ of cost $c + N - m$. Consider the set M comprising all m edges in $\widetilde{M}$ that do not "touch" a dummy capsule. Since $\widetilde{M}$ is a matching in $\widetilde{G}$, M is a matching in G. Moreover, the cost of M is just $N - m$ less than the cost of $\widetilde{M}$, namely, c. □

Thus, the reduction preceding Lemma 2 yields the desired placement algorithm. We construct the feasibility graph G_A for application A and augment it to $\widetilde{G}_A$. If $\widetilde{G}_A$ contains a perfect matching, then we remove the edges that "touch" dummy capsules and obtain the desired placement. If $\widetilde{G}_A$ does not contain a

[3]If $m \geq N$, then this application cannot be placed on this hosting platform.

perfect matching, then we know that A cannot be placed. One finds polynomial-time algorithms for MWPM in Edmonds [1965] and Cook and Rohe [1999].

3.4 Handling Dynamically Changing Resource Requirements

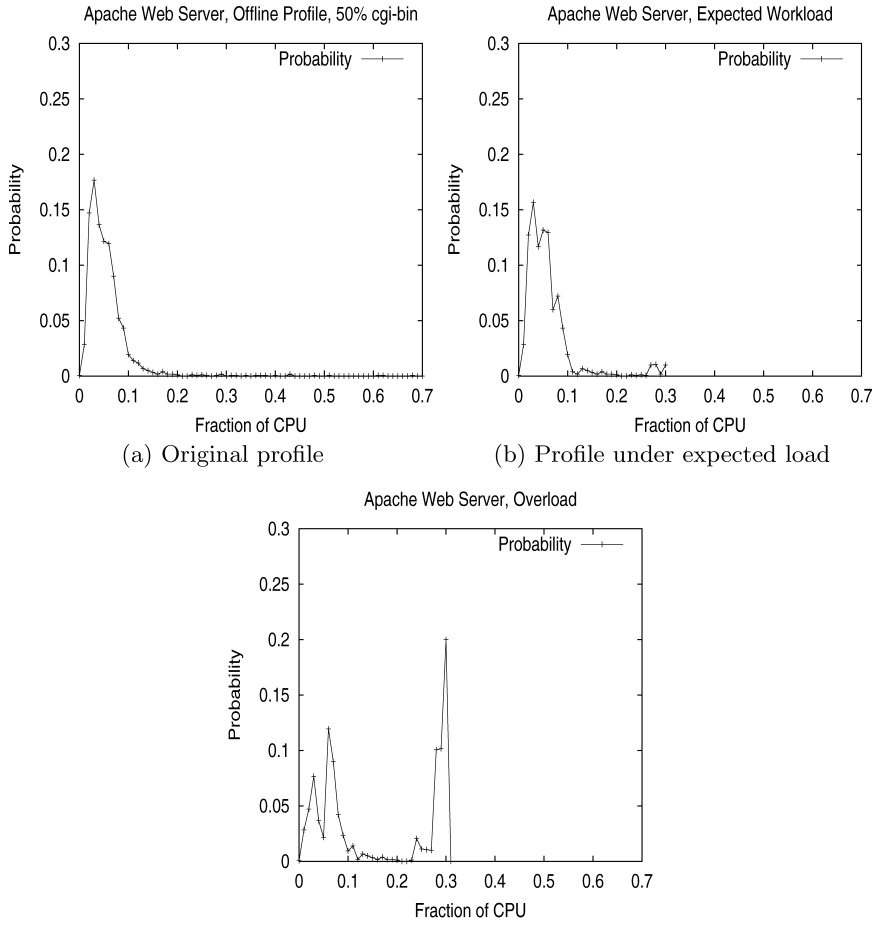
Our discussion thus far has assumed that the resource requirements of an application at run-time do not change after the initial profiling phase. In reality though, resource requirements change dynamically over time, in tandem with the workload seen by the application. Hosting platforms need to employ dynamic capacity provisioning to match the resources assigned to hosted applications to their varying workloads [Benani and Menasce 2005; Ranjan et al. 2002; Urgaonkar et al. 2005a; Appleby et al. 2001; Chase and Doyle 2001; Chen et al. 2005; Aron et al. 2000]. *How should a hosting platform that employs overbooking of resources implement dynamic capacity provisioning?* In this section, we outline our approach to address this question.

First, recall that we provision resources based on a high percentile of the application's resource usage distribution. Consequently, variations in the application workload that affect only the average resource requirements of the capsules, *but not the tail of the resource usage distribution*, will not result in violations of the probabilistic guarantees provided by the hosting platform. In contrast, workload changes that cause an increase in the tail of the resource usage distribution will certainly affect application QoS guarantees.

How a platform should deal with such changes in resource requirements depends on several factors. Since we are interested in yield management, the platform should increase the resources allocated to an overload application only if it increases revenues for the platform provider. Thus, if an application provider only pays for a fixed amount of resources, there is no economic incentive for the platform provider to increase the resource allocation beyond this limit even if the application is overloaded. In contrast, if the contract between the application and platform provider permits usage-based charging (i.e., charging for resources based on the actual usage, or a high percentile of the usage[4]), then allocating additional resources in response to increased demand is desirable for maximizing revenue. In such a scenario, handling dynamically changing requirements involves two steps: (i) detecting changes in the tail of the resource usage distribution, and (ii) reacting to these changes by varying the actual resources allocated to the application.

To detect such changes in the tail of an application's resource usage distribution, we propose to conduct continuous, on-line profiling of the resource usage of all capsules using low-overhead profiling tools. This would be done by recording the CPU scheduling instants, network transmission times and packet sizes for all processes over intervals of a suitable length. At the end of each interval, this data would be processed to construct the latest resource usage distributions for all capsules. An application overload would manifest itself through an increased concentration in the high percentile buckets of the resource usage distributions of its capsules.

[4]ISPs charge for network bandwidth in this fashion—the customer pays for the 95th percentile of it s bandwidth usage over a certain period.



(a) Original profile (b) Profile under expected load

(c) Profile under overload

Fig. 8. Demonstration of how an application overload may be detected by comparing the latest resource usage profile with the original offline profile.

We present the results of a simple experiment to illustrate this. Figure 8(a) shows the CPU usage distribution of the Apache Web server obtained via offline profiling. The workload for the Web server was generated by using the SPECWeb99 benchmark emulating 50 concurrent clients with 50% dynamic cgi-bin requests. The offline profiling was done over a period of 30 minutes. Next, we assumed an overbooking tolerance of 1% for this Web server capsule. As described in Section 2.3, it was assigned a CPU rate of 0.29 (corresponding to the 99th percentile of its CPU usage distribution). The remaining capacity was assigned to a greedy *dhrystone* application (this application performs compute-intensive integer computations and greedily consumes all resources allocated to it). The Web server was then subjected to exactly the same workload (50 clients with 50% cgi-bin requests) for 25 minutes, followed by a heavier workload consisting of 70 concurrent clients with 70% dynamic cgi-bin requests for 5 minutes. The heavier workload during the last 5 minutes was

to simulate an unexpected flash crowd. The Web server's CPU usage distribution was recorded over periods of length 10 minute each. Figure 8(b) shows the CPU usage distribution observed for the Web server during a period of expected workload. We find that this profile is very similar to the profile obtained using offline measurements, except being upper-bounded by the CPU rate assigned to the capsule. Figure 8(c) plots the CPU usage distribution during the period when the Web server was overloaded. We find an increased concentration in the high percentile regions of this distribution compared to the original distribution.

The detection of application overload would trigger remedial actions that would proceed in two stages. First, new resource requirements would be computed for the affected capsules. Next, actions would be taken to provide the capsules the newly computed resource shares—this may involve increasing the resource allocations of the capsules, or moving the capsules to nodes with sufficient resources. Implementing and evaluating these techniques for handling application overloads are part of our ongoing research on shared hosting platforms. We have designed, implemented, and evaluated such remedial actions in a recent paper on dynamic capacity provisioning in hosting platforms [Urgaonkar et al. 2005b]. These techniques can be used by a shared hosting platform employing resource overbooking to dynamically adjust the placement of hosted applications in response to varying workloads.

4. IMPLEMENTATION CONSIDERATIONS

In this section, we first discuss implementation issues in integrating our resource overbooking techniques with OS resource allocation mechanisms. We then present an overview of our prototype implementation.

4.1 Providing Application Isolation at Run Time

The techniques described in the previous section allow a platform provider to overbook platform resources and yet provide guarantees that the QoS requirements of applications will be met. The task of enforcing these guarantees at run-time is the responsibility of the OS kernel. To meet these guarantees, we assume that the kernel employs resources allocation mechanisms that support some notion of quality of service. Numerous such mechanisms—such as reservations, shares and token bucket regulators [Banga et al. 1999; Duda and Cheriton 1999; Jones et al. 1997; Leslie et al. 1996]—have been proposed recently. All of these mechanisms allow a certain fraction of each resource (CPU cycles, network interface bandwidth) to be reserved for each application and enforce these allocations on a fine time scale.

In addition to enforcing the QoS requirements of each application, these mechanisms also isolate applications from one another. By limiting the resources consumed by each application to its reserved amount, the mechanisms prevent a malicious or overloaded application from grabbing more than its allocated share of resources, thereby providing application isolation at run-time—an important requirement in shared hosting environments running untrusted applications.

Our overbooking techniques can exploit many commonly used QoS-aware resource allocation mechanisms. Since the QoS requirements of an application are defined in a OS- and mechanism-independent manner, we need to map these OS-independent QoS requirements to mechanism-specific parameter values. We outline these mappings for three commonly-used QoS-aware mechanisms.

CPU Reservations. A reservation-based CPU scheduler [Jones et al. 1997; Leslie et al. 1996] requires the CPU requirements to be specified as a pair (x, y) where the capsule desires x units of CPU time every y time units (effectively, the capsule requests $\frac{x}{y}$ fraction of the CPU). For reasons of feasibility, the sum of the requests allocations should not exceed 1 (i.e., $\sum_j \frac{x_j}{y_j} \leq 1$). In such a scenario, the QoS requirements of a capsule with token bucket parameters (σ_i, ρ_i) and an overbooking tolerance O_i can be translated to CPU reservation by setting $(1-O_i)\cdot\sigma_i = \frac{x_i}{y_i}$ and $(1-O_i)\cdot\rho_i = x_i$. To see why, recall that $(1-O_i)\cdot\sigma_i$ denotes the rate of resource consumption of the capsule in the presence of overbooking, which is same as $\frac{x_i}{y_i}$. Further, since the capsule can request x_i units of the CPU every y_i time units, and in the worst case, the entire x_i units may be requested continuously, we set the burst size to be $(1 - O_i) \cdot \rho_i = x_i$. These equations simplify to $x_i = (1 - O_i) \cdot \rho_i$ and $y_i = \rho_i/\sigma_i$.

Proportional-Share and Lottery Schedulers. Proportional-share and lottery schedulers [Duda and Cheriton 1999; Goyal et al. 1996a, 1996b; Waldspurger and Weihl 1994] enable resources to be allocated in relative terms—in either case, a capsule is assigned a weight w_i (or w_i lottery tickets) causing the scheduler to allocate $w_i \sum_j w_j$ fraction of the resource. Further, two capsules with weights w_i and w_j are allocated resources in proportion to their weights $(w_i : w_j)$. For such schedulers, the QoS requirements of a capsule can be translated to a weight by setting $w_i = (1 - O_i) \cdot \sigma_i$. By virtue of using a single parameter w_i to specify the resource requirements, such schedulers ignore the burstiness ρ in the resource requirements. Consequently, the underlying scheduler will only approximate the desired QoS requirements. The nature of approximation depends on the exact scheduling algorithm—the finer the timescale of the allocation supported by the scheduler, the better will the actual allocation approximate the desired requirements.

Rate Regulators. Rate regulators are commonly used to police the network interface bandwidth used by an application. Such regulators limit the sending rate of the application based on a specified profile. A commonly used regulator is the token bucket regulator that limits the amount of bytes transmitted by an application to $\sigma \cdot t + \rho$ over any interval t. Since we model resource usage of a capsule as a token bucket, the QoS requirements of a capsule trivially map to an actual token bucket regulator and no special translation is necessary.

4.2 Prototype Implementation

We have implemented a Linux-based shared hosting platform that incorporates the techniques discussed in the previous sections. Our implementation consists of three key components: (i) a profiling module that allows us to profile

applications and empirically derive their QoS requirements, (ii) a control plane that is responsible for resource overbooking and capsule placement, and (iii) a QoS-enhanced Linux kernel that is responsible for enforcing application QoS requirements.

The profiling module runs on a set of dedicated (and therefore isolated) platform nodes and consists of a vanilla Linux kernel enhanced with the Linux trace toolkit. As explained in Section 2, the profiling module gathers a kernel trace of CPU and network activities of each capsule. It then post-processes this information to derive an On-Off trace of resource usage and then derives the usage distribution U and the token bucket parameters for this usage.

The control plane is responsible for placing capsules of newly arriving applications onto nodes while overbooking node resources. The control plane also keeps state consisting of a list of all capsules residing on each node and their QoS requirements. It also maintains information about the hardware characteristics of each node. The requirements of a newly arriving application are specified to the control plane using a resource specification language. This specification includes the CPU and network bandwidth requirements of each capsule and the trust vector. The control plane uses this specification to derive a placement for each capsule as discussed in Section 3.2. In addition to assigning each capsule to a node, the control plane also translates the QoS parameters of the capsules to parameters of commonly used resource allocation mechanisms (discussed in the previous section).

The third component, namely the QoS-enhanced Linux kernel, runs on each platform node and is responsible for enforcing the QoS requirements of capsules at run time. We choose Linux over other operating system kernels since a number of QoS-aware resource allocation mechanisms have already been implemented in Linux, allowing us to experiment with these mechanisms. For the purposes of this article, we implement the H-SFQ proportional-share CPU scheduler [Goyal et al. 1996a, 1996b]. H-SFQ is a *hierarchical* proportional-share scheduler that allows us to group resource principals (processes, lightweight processes) and assign an aggregate CPU share to the entire group. This functionality is essential since a capsule contains all processes of an application that are collocated on a node and the QoS requirements are specified for the capsule as a whole rather than for individual resource principals. To implement such an abstraction, we create a separate node in the H-SFQ scheduling hierarchy for each capsule, and attach all resource principals belonging to a capsule to this node. The node is then assigned a weight (determined using the capsule's QoS requirements) and the CPU allocation of the capsule is shared by all resource principals of the capsule.[5] We implement a token bucket regulator to provide QoS guarantees at the network interface card. Our rate regulator allows us to associate all network sockets belonging to a group of processes to a single token bucket. We instantiate a token bucket regulator for each capsule and regulate the network bandwidth usage of all resource principals contained in this capsule using the (σ, ρ) parameters of the

[5]The use of the scheduling hierarchy to further multiplex capsule resources among resource principals in a controlled way is clearly feasible but beyond the scope of this article.

capsule's network bandwidth usage. In Section 5.3, we experimentally demonstrate the efficacy of these mechanisms in enforcing the QoS requirements of capsules even in the presence of overbooking. While we have experimented with other resource allocation mechanisms such as reservations [Leslie et al. 1996] and have found that overbooking techniques indeed work well with other commonly used mechanisms, we omit here the results obtained using these other mechanisms due to space constraints.

5. EXPERIMENTAL EVALUATION

In this section, we present the results of our experimental evaluation. The setup used in our experiments is identical to that described in Section 2.4—we employ a cluster of Linux-based servers as our shared hosting platform. Each server runs a QoS-enhanced Linux kernel consisting of the H-SFQ CPU scheduler and a leaky bucket regulator for the network interface. The control plane for the shared platform implements the resource overbooking and capsule placement strategies discussed earlier in this article. For ease of comparison, we use the same set of applications discussed in Section 2.4 and their derived profiles (see Table I) for our experimental study.

Since our resource overbooking techniques apply only to CPU and network bandwidth, in all the experiments presented in this section, we ensure that the working sets of co-located applications can be accommodated within the RAM present in the server. We similarly ensure that the sum of the disk bandwidth needs of co-located applications can be comfortably accommodated within the aggregate available disk bandwidth. All the conclusions drawn from our empirical evaluation, therefore, hold subject to the condition that memory and disk bandwidth were available in plenty in any shared server.

5.1 Efficacy of Resource Overbooking

Our first experiment examines the efficacy of overbooking resources in shared *Web hosting* platforms—a type of shared hosting platform that runs only Web servers. Each Web server running on the platform is assumed to conform to one of the four Web server profiles gathered from our profiling study (two of these profiles are shown in Table I; the other two employed varying mixes of static and dynamic SPECWeb99 requests). The objective of our experiment is to examine how many such Web servers can be supported by a given platform configuration for various overbooking tolerances. We vary the overbooking tolerance from 0% to 10%, and for each tolerance value, attempt to place as many Web servers as possible until the platform resources are exhausted. We first perform the experiment for a cluster of 5 nodes (identical to our hardware configuration) and then repeat it for cluster sizes ranging from 16 to 128-nodes (since we lack clusters of these sizes, for these experiments, we only examine how many applications can be accommodated on the platform and do not actually run these applications). Figure 9 depicts our results with 95% confidence intervals. The figure shows that, the larger the amount of overbooking, the larger is the number of Web servers that can be run on a given platform. Specifically, for a 128-node platform, the number of Web servers that can be supported increases

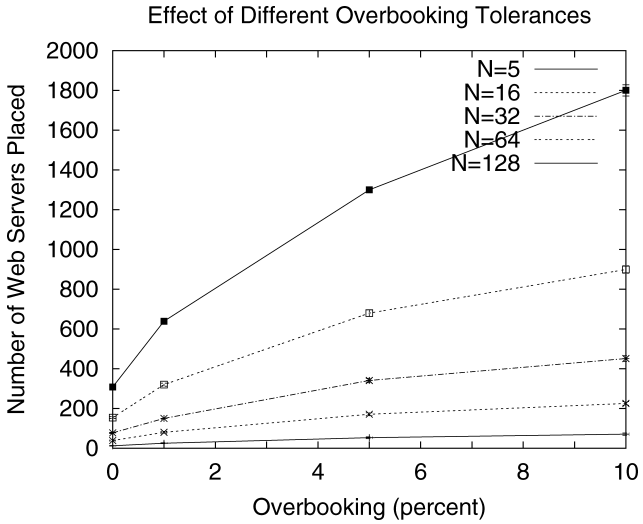
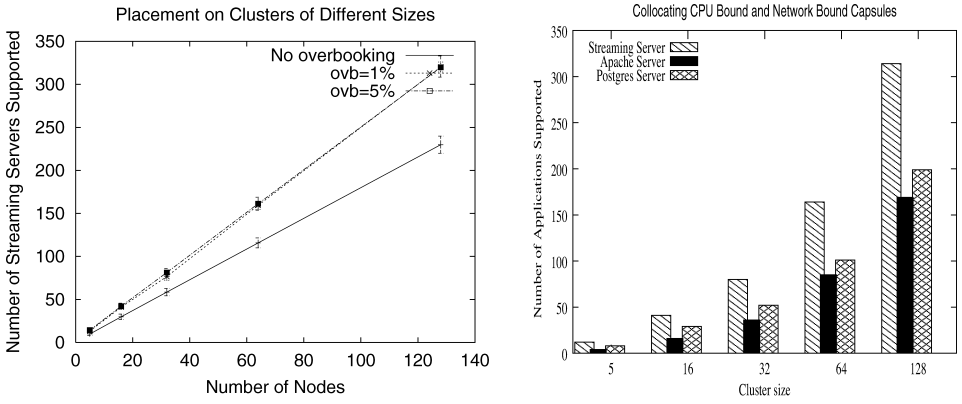


Fig. 9. Benefits of resource overbooking in a Web hosting platform.

from 307 when no overbooking is employed to over 1800 for 10% overbooking (a factor of 5.9 increase). Even for a modest 1% overbooking, we see a factor of 2 increase in the number of Web servers that can be supported on platforms of various sizes. Thus, even modest amounts of overbooking can significantly enhance revenues for the platform provider.

Next, we examine the benefits of overbooking resources in a shared hosting platform that runs a mix of streaming servers, database servers and Web servers. To demonstrate the impact of burstiness on overbooking, we first focus only on the streaming media server. As shown in Table I, the streaming server (with 20 clients) exhibits less burstiness than a typical Web server, and consequently, we expect smaller gains due to resource overbooking. To quantify these gains, we vary the platform size from 5 to 128 nodes and determine the number of streaming servers that can be supported with 0%, 1% and 5% overbooking. Figure 10(a) plots our results with 95% confidence intervals. As shown, the number of servers that can be supported increases by 30–40% with 1% overbooking when compared to the no overbooking case. Increasing the amount of overbooking from 1% to 5% yields only a marginal additional gain, consistent with the profile for this streaming server shown in Table I (and also indicative of the less-tolerant nature of this soft real-time application). Thus, less bursty applications yield smaller gains when overbooking resources.

Although the streaming server does not exhibit significant burstiness, large statistical multiplexing gains can still accrue by co-locating bursty and non-bursty applications. Further, since streaming server is heavily network-bound and uses a minimal amount of CPU, additional gains are possible by co-locating applications with different bottleneck resources (e.g., CPU-bound and network-bound applications). To examine the validity of this assertion, we conduct an experiment where we attempt to place a mix of streaming, Web and database servers—a mix of CPU-bound and network-bound as well as bursty



(a) Streaming media servers (b) Mix of applications

Fig. 10. Benefits of resource overbooking for a less bursty streaming server application and for application mixes.

and non-bursty applications. Figure 10(b) plots the number of applications supported by platforms of different sizes with 1% overbooking. As shown, an identical platform configuration is able to support a large number of applications than the scenario where only streaming servers are placed on the platform. Specifically, for a 32-node cluster, the platform supports 36 and 52 additional Web and database servers in addition to the approximately 80 streaming servers that were supported earlier. We note that our capsule placement algorithms are automatically able to extract these gains without any specific "tweaking" on our part. Thus, collocating applications with different bottleneck resources and different amounts of burstiness enhance additional statistical multiplexing benefits when overbooking resources.

It must be noted that these gains offered by overbooking are compared to schemes that provision resources for applications based on the peak demands. Whereas our techniques explicitly characterize the tail of resource usage and provision based on it, certain applications with rate regulators embedded in them are implicitly provisioned based on tails. The peak demands observed for such applications by our offline profiling technique would in fact correspond to some high percentile of their "actual" resource usage (i.e., if there were no rate regulation.) Clearly, the benefits of overbooking would turn out to be less impressive for such applications since their a significant part of their burstiness is likely to have been removed by the rate regulators.

5.2 Capsule Placement Algorithms

Our next experiment compares the effectiveness of the best-fit, worst-fit and random placement algorithms discussed in Section 3.2. Using our profiles, we construct two types of applications: a replicated Web server and an e-commerce application consisting of a front-end Web server and a back-end database server. Each arriving application belongs to one of these two categories and is assumed to consist of 2–10 capsules, depending on the degree of replication. The overbooking tolerance is set to 5%. We then determine the number of applications

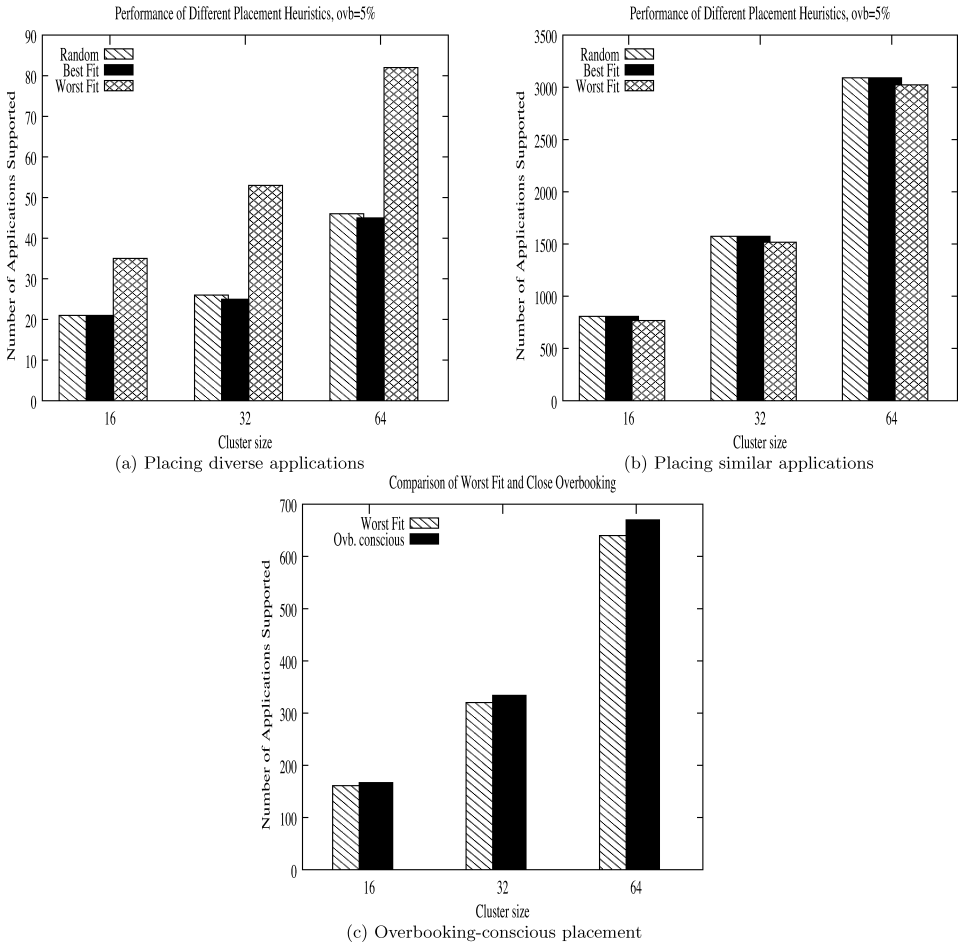


Fig. 11. Performance of various capsule placement strategies.

that can be placed on a given platform by different placement strategies. Figure 11(a) depicts our results. As shown, best-fit and random placement yield similar performance, while worst-fit outperforms these two policies across a range of platform sizes. This is because best-fit places capsules onto nodes with smaller unused capacity, resulting in "fragmentation" of unused capacity on a node; the leftover capacity may be wasted if no additional applications can be accommodated. Worst fit, on the other hand, reduces the chances of such fragmentation by placing capsules onto nodes with the larger unused capacity. While such effects become prominent when application capsules have widely varying requirements (as observed in this experiment), they become less noticeable when the application have similar resource requirements. To demonstrate this behavior, we attempted to place Quake game servers onto platforms of various sizes. Observe from Table I that the game server profiles exhibit less diversity than a mix of Web and database servers. Figure 11(b) shows that, due

Table II. Effectiveness of Kernel Resource Allocation Mechanisms

Application	Metric	Isolated Node	100th	99th	95th	Average
Apache	Throughput (req/s)	67.93 ± 2.08	67.51 ± 2.12	66.91 ± 2.76	64.81 ± 2.54	39.82 ± 5.26
PostgreSQL	Throughput (transactions/s)	22.84 ± 0.54	22.46 ± 0.46	22.21 ± 0.63	21.78 ± 0.51	9.04 ± 85
Streaming	Number of violations	0	0	2	40	98

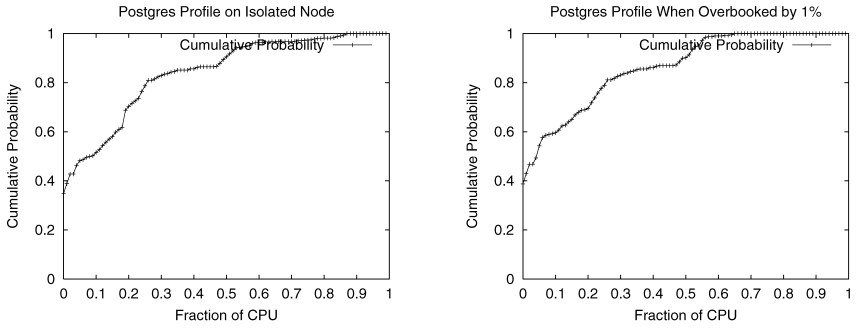
All results are shown with 95% confidence intervals.

to the similarity in the application resource requirements, all policies are able to place a comparable number of game servers.

Finally, we examine the effectiveness of taking the overbooking tolerance into account when making placement decisions. We compare the worst-fit policy to an overbooking-conscious worst-fit policy. The latter policy chooses the three worst-fits among all feasible nodes and picks the node that best matches the overbooking tolerance of the capsule. Our experiment assumes a Web hosting platform with two types of applications: less-tolerant Web servers that permit 1% overbooking and more tolerant Web servers that permit 10% overbooking. We vary the platform size and examine the total number of applications placed by the two policies. As shown in Figure 11(c), taking overbooking tolerances into account when making placement decisions can help increase the number of applications placed on the cluster. However, we find that the additional gains are small ($< 6\%$ in all cases), indicating that a simple worst-fit policy may suffice for most scenarios.

5.3 Effectiveness of Kernel Resource Allocation Mechanisms

While our experiments thus far have focused on the impact of overbooking on platform capacity, in our next experiment, we examine the impact of overbooking on application performance. We show that combining our overbooking techniques with kernel-based QoS resource allocation mechanisms can indeed provide application isolation and quantitative performance guarantees to applications (even in the presence of overbooking). We begin by running the Apache Web server on a dedicated (isolated) node and examine its performance (by measuring throughput in requests/s) for the default SPECWeb99 workload. We then run the Web server on a node running our QoS-enhanced Linux kernel. We first allocate resources based on the 100th percentile of its usage (no overbooking) and assign the remaining capacity to a greedy *dhrystone* application (this application performs compute-intensive integer computations and greedily consumes all resources allocated to it). We measure the throughput of the Web server in presence of this background dhrystone application. Next, we reserve resources for the Web server based on the 99th and the 95th percentiles, allocate the remaining capacity to the dhrystone application, and measure the server throughput. Table II depicts our results. As shown, provisioning based on the 100th percentile yields performance that is comparable to running the application on an dedicated node. Provisioning based on the 99th and 95th percentiles results in a small degradation in throughput, but well within the



(a) PostgreSQL CDF on isolated node (b) PostgreSQL CDF on an overbooked node

Fig. 12. Effect of overbooking on the PostgreSQL server CPU profile.

permissible limits of 1% and 5% degradation, respectively, due to overbooking. Table II also shows that provisioning based on the *average* resource requirements results in a substantial fall in throughput, indicating that reserving resources based on mean usage is not advisable for shared hosting platforms.

We repeat the above experiment for the streaming server and the database server. The background load for the streaming server experiment is generated using a greedy UDP sender that transmits network packets as fast as possible, while that in case of the database server is generated using the dhrystone applications. In both cases, we first run the application on an isolated node and then on our QoS-enhanced kernel with provisioning based on the 100th, 99th and the 95th percentiles. We measure the throughput in transaction/s for the database server and the mean number of violations (playback discontinuities) experienced by a client of the streaming media server. Table II plots our results. Like in the Web server, provisioning based on the 100th percentile yields performance comparable to running the application on an isolated node, while small amounts of overbooking result in correspondingly small amounts of degradation in application performance.

For each of the above scenarios, we also computed the application profile in the presence of background load and overbooking and compared it to the profile gathered on the isolated node. Figure 12 shows one such pair and compares the profile of the database server on the isolated node with that obtained with background load and 1% overbooking. As can be seen, the two profiles look similar, indicating that the presence of background load does not interfere with the application behavior, and hence, the profiles obtained by running the application on an isolated node are representative of the behavior on an overbooked node (for a given workload).

Together, these results demonstrate that our kernel resource allocation mechanisms are able to successfully isolate applications from one another and are able to provide quantitative performance guarantees even when resources are overbooked.

5.4 Determining the Degree of Overbooking

In this section, we present an experimental illustration of the empirical approach for determining the degree of overbooking that we presented

in Section 2.3.2. We experiment with the streaming media server in this section.

The network bandwidth profile for the streaming media server with 20 clients was presented in Figure 5 and selected percentiles of the distribution were presented in Table I. We will illustrate our approach by comparing the expected revenue from the streaming media server under degrees of overbooking corresponding to these selected percentiles. For illustrative purposes, we assume that the streaming media server pays the hosting platform \$5 for every successfully streamed movie. However, it imposes a penalty for every playback discontinuity experienced by its clients. We consider a wide range of penalty values from the set {\$0.01, \$0.1, \$0.25, \$1}. For the purposes of this article, we assume that when we provision the network bandwidth based on a certain percentile lower than the peak, the residual bandwidth (which would be made available to another co-located application) yields the same revenue per unit bandwidth as yielded by the streaming media server.[6] We subject the streaming media server to the workload imposed by 20 clients under different degrees of network bandwidth overbooking as described in Section 2.3.2. For our four choices of the penalty paid for playback discontinuities and with the identical revenue per unit bandwidth assumed above, we obtain Figures 13(a)–(d). Each figure plots the variation of expected revenue (what the hosted applications pay the platform *minus* the penalty due to QoS violations). Note that we have also assumed that the residual bandwidth is provisioned in a way that does not cause any penalties, another simplifying assumption in our current approach.

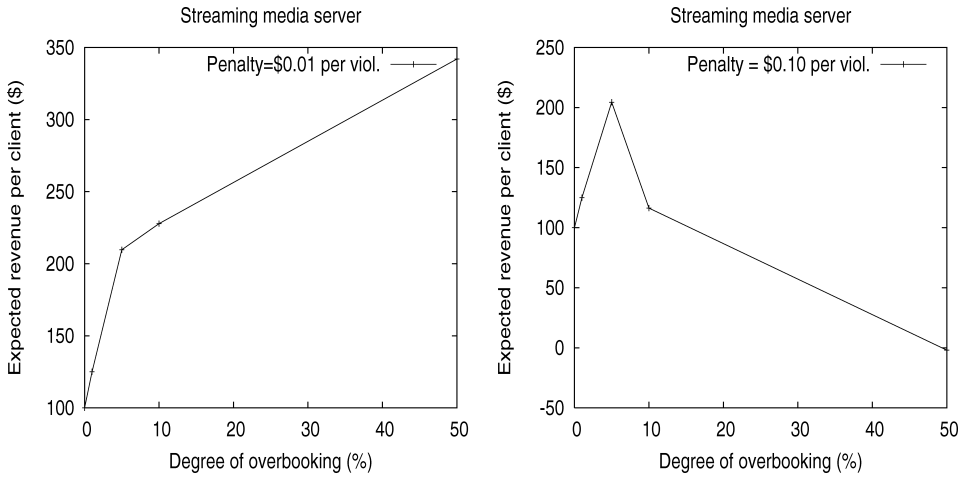
The monotonically increasing revenue in Figure 13(a) simply indicates that the first penalty function of \$0.01 per playback discontinuity is an impractical and injudicious choice under the assumed charging model. For the remaining three penalty functions, an overbooking of 5% is expected to yield the maximum revenue (Figures 13(b)–(d).)

6. RELATED WORK

6.1 Prior Research on Resource Management in Clustered Environments

Research on clustered environments over the past decade has spanned a number of issues. Systems such as Condor have investigated techniques for harvesting idle CPU cycles on a cluster of workstations to run batch jobs [Litzkow et al. 1988]. The design of scalable, fault-tolerant network services running on server clusters has been studied in Fox et al. [1997]. Use of virtual clusters to manage resources and contain faults in large multiprocessor systems has been studied in Govil et al. [1999]. Scalability, availability, and performance issues in dedicated clusters have been studied in the context of clustered mail servers [Saito et al. 1999] and replicated Web servers [Aron et al. 2000]. Numerous middleware-based approaches for clustered environments have also been

[6]Note that, in general, different applications would yield different amounts of revenue per unit resource. A general optimization problem capturing this is part of future work and is beyond the scope of the current work. However, we believe our simple example is sufficiently illustrative of the key ideas.



(a) $0.01 per discontinuity (b) $0.1 per discontinuity

(c) $0.25 per discontinuity (d) $1.0 per discontinuity

Fig. 13. Expected revenue from the streaming media server with different penalties per playback discontinuity.

proposed [Corba 2006]. Ongoing efforts in the grid computing community have focused on developing standard interfaces for resource reservations in clustered environments [GridForum02b 2002].

Finally, efforts such as gang scheduling and co-scheduling have investigated the issue of coordinating the scheduling of tasks in distributed systems [Arpaci-Dusseau 2001; Hori et al. 1996]; however, neither technique incorporates the issue of quality-of-service while making scheduling decisions.

6.2 Operating Systems Support for QoS and Dynamic Resource Management

In the context of QoS-aware resource allocation, numerous efforts over the past decade have developed predictable resource allocation mechanisms for single

machine environments [Banga et al. 1999; Blanquer et al. 1999; Berger et al. 2003; Duda and Cheriton 1999; Goyal et al. 1996a, 1996b; Jones et al. 1997; Lin et al. 1998; Leslie et al. 1996; Sundaram et al. 2000; Verghese et al. 1998]. Such techniques form the building blocks for resource allocation in clustered environments. Recent research on designing efficient virtual machine monitors has made available systems infrastructure that can provide improved consolidation of applications in a shared hosting platform (by allowing heterogeneous operating systems to be co-located) as well as more flexible migration of application/capsules in response to changing workloads [Barham et al. 2003; Govindan et al. 2007; Clark et al. 2005; Menon et al. 2006, 2005; Sapuntzakis et al. 2002; M. Nelson and Hutchins 2005; Waldspurger 2002; Whitaker et al. 2002].

6.3 Resource Management in Hosting Platforms

Research on resource management in hosting platforms spans the topics of specification of service-level agreements (SLAs), techniques to realize them such as static capacity planning (similar to resource requirement inference which we discuss separately below), dynamic capacity provisioning, and admission control.

The WSLA project at IBM [WSLA] addresses service level management issues and challenges in designing an unambiguous and clear specification of SLAs that can be monitored by the service provider, customer, and even by a third-party.

The work on dynamic provisioning of a platform's resources may be classified into two categories. Some papers have addressed the problem of provisioning resources at the granularity of individual servers as in our work. Aron et al. [2000], Benani and Menasce [2005], Ranjan et al. [2002], and Urgaonkar et al. [2005a] consider the problem of dynamically varying the number of servers assigned to a single service hosted on a data center. The Oceano project at IBM [Appleby et al. 2001] has developed a server farm in which servers can be moved dynamically across hosted applications depending on their changing needs.

Other papers have considered the provisioning of resources at finer granularity of resources. Muse [Chase and Doyle 2001; Chen et al. 2005] presents an architecture for resource management in a hosting center. The system computes resource allocations by attempting to maximize the overall profit. *Cluster Reserves* [Aron et al. 2000] has also investigated resource allocation in server clusters. The work assumes a large application running on a cluster, where the aim is to provide differentiated service to clients based on some notion of service *class*. This is achieved by making the OS schedulers provide fixed resource shares to applications spanning multiple nodes. The *Cluster-On Demand (COD)* [Chase et al. 2003] work presents an automated framework to manage resources in a shared hosting platform. COD introduces the notion of a *virtual cluster*, which is a functionally isolated group of hosts within a shared hardware base. A key element of COD is a protocol to re-size virtual clusters dynamically in cooperation with pluggable middleware components. Chandra et al. [2003a] model a server resource that services multiple applications as a GPS system and presents online workload prediction and optimization-based techniques for

dynamic resource allocation. Urgaonkar and Shenoy [Urgaonkar and Shenoy 2004b] address the problem of providing resource guarantees to distributed applications running on a shared hosting platform.

Statistical admission control techniques that overbook resources have been studied in the context of video-on-demand servers [Vin et al. 1994] and ATM networks [Boorstyn et al. 2000]. To the best of our knowledge, ours is the first work to consider resource overbooking in context of shared hosting platforms (i.e., clustered environments).

Numerous papers address the problem of maintaining SLAs by using admission control [Voigt et al. 2001; Welsh and Culler 2003; Iyer et al. 2000; Cherkasova and Phaal 1999; Kanodia and Knightly 2000; Li and Jamin 2000; Knightly and Shroff 1999; Verma and Ghosal 2003; Elnikety et al. 2004; Jamjoom et al. 2000].

6.4 Resource Requirement Inference

Numerous papers have developed analytical models for various kinds of applications. We take an empirical approach in this paper as these analytical approaches are only suitable for describing average resource requirements of applications and are therefore of limited use in overbooking of resources. Modeling of single-tier Internet applications, of which HTTP servers are the most common example, has been studied extensively. A queuing model of a Web server serving static content was proposed in Slothouber [1996]. The model employs a network of four queues—two modeling the Web server itself, and the other two modeling the Internet communication network. A queuing model for performance prediction of single-tier Web servers with static content was proposed in Doyle et al. [2003]. This approach (i) explicitly models CPU, memory, and disk bandwidth in the Web server, (ii) utilizes knowledge of file size and popularity distributions, and (iii) relates average response time to available resources. A GPS-based queuing model of a single resource, such as the CPU, at a Web server was proposed in Chandra et al. [2003a]. The model is parameterized by online measurements and is used to determine the resource allocation needed to meet desired average response time targets. A G/G/1 queuing model for replicated single-tier applications (e.g., clustered Web servers) has been proposed in Urgaonkar and Shenoy [2004a]. The architecture and prototype implementation of a performance management system for cluster-based Web services was proposed in Levy et al. [2003]. The work employs an M/M/1 queuing model to compute responses times of Web requests. A model of a Web server for the purpose of performance control using classical feedback control theory was studied in Abdelzaher et al. [2002]; an implementation and evaluation using the Apache Web server was also presented in the work. A combination of a Markov chain model and a queuing network model to capture the operation of a Web server was presented in Menasce [2003]—the former model represents the software architecture employed by the Web server (e.g., process-based versus thread-based) while the latter computes the Web server's throughput.

A few recent efforts have focused on the modeling of multi-tier applications. However, many of these efforts either make simplifying assumptions or are

based on simple extensions of single-tier models. A number of papers have taken the approach of modeling only the *most constrained* or the *most bottlenecked* tier of the application. For instance, Villela et al. [2004] considers the problem of provisioning servers for only the Java application tier; it uses an M/G/1/PS model for each server in this tier. Similarly, the Java application tier of an e-commerce application with N servers is modeled as a G/G/N queuing system in Ranjan et al. [2002]. Other efforts have modeled the entire multi-tier application using a single queue—an example is Kamra et al. [2004], that uses a M/GI/1/PS model for an e-commerce application. While these approaches are useful for specific scenarios, they have many limitations. For instance, modeling only a single bottlenecked tier of a multi-tier application will fail to capture caching effects at other tiers. Such a model can not be used for capacity provisioning of other tiers. Finally, as we show in our experiments, system bottlenecks can shift from one tier to another with changes in workload characteristics. Under these scenarios, there is no single tier that is the "most constrained".

Some researchers have developed sophisticated queuing models capable of capturing the simultaneous resource demands and parallel subpaths that occur within a tier of a multi-tier application. An important example of such models are Layered Queuing Networks (LQN). LQNs are an adaptation of the Extended Queuing Network defined specifically to represent the fact that software servers are executed on top of other layers of servers and processors, giving complex combinations of simultaneous requests for resources [Liu et al. 2001; Franks 1999; Xu et al. 2006; Rolia and Sevcik 1995; Woodside and Raghunath 1995]. Kounev and Buchmann [2003] use a model based on a network of queues for performance prediction of a 2-tier SPECjAppServer2002 application and solve this model numerically using publicly available analysis software. Bennani and Menasce [2005] model a multi-tier Internet service serving multiple types of transactions as a network of queues with customers belonging to multiple classes. The authors employ an approximate mean-value analysis algorithm to develop an online provisioning technique using this model. Two of the coauthors on this submission developed a queueing model for a multi-tier Internet application that also employs mean-value analysis [Urgaonkar et al. 2005a].

Cohen et al. [2004] uses a probabilistic modeling approach called Tree-Augmented Bayesian Networks (TANs) to identify combinations of system-level metrics and threshold values that correlate with high-level performance states—compliance with service-level agreements for average response time—in a three-tier Web service under a variety of conditions. Experiments based on real applications and workloads indicate that this model is a suitable candidate for use in offline fault diagnosis and online performance prediction. Whereas it would be a useful exercise to compare such a learning-based modeling approach with our queuing-theory based model, it is beyond the scope of this article. In the absence of such a comparative study and given the widely different nature of these two modeling approaches, we do not make any assertions about the pros and cons of our model over the TAN-based model.

Finally, following the appearance of our conference paper, Stewart and Shen [2005] have employed an empirical characterization of the resource requirements of multi-tier Internet applications similar to that developed by us.

6.5 Yield Management

Yield management was first explored in the airline industry, particularly by American Airlines [Davis 1994; Smith et al. 1992]. Yield management practices have subsequently been also explored in certain IT areas such as telephony and networking [Gupta et al. 1999; Vin et al. 1994; Boorstyn et al. 2000]. To the best of our knowledge, our conference paper was the first to study YM in the domain of Internet hosting.

There have recently been questions about how effective YM is in the big picture. A firm that wants to satisfy its customers and have them come back are putting their customer relations in jeopardy by using YM practices. While this statistic is impressive and shows how YM can be effective, it is also misleading. Many economists argue that the benefits of YM are only felt up-front and are short-lived. The costs of lower customer satisfaction and the loss of relationship marketing can have longer more serious effects and in the end make it detrimental to the firm. Because American Airlines was a pioneer with YM, it is obvious that the innovation will result in a large increase in revenue. However, as the rest of the industry catches up with the technology if the YM systems, the competitive advantage can be lost, while the long lasting costs are not. Also, customers feel that the personal connection with the company is lost as the company sees them only as revenue generators. This can result in the loss of relationship marketing and goodwill between the parties. So it comes down to a cost-benefit situation for the firm. The extra revenue now is the benefit and loss of goodwill and possibly a drop in revenue in the future is the cost. It is of course up to the firm to forecast if the benefits outweigh the costs. In this article, we take an initial step towards studying such tradeoffs when using YM in shared hosting platforms. Our approach is empirical in nature and is presented in Section 5.4.

6.6 Management and Overbooking of Nontemporal Resources

Extensive existing research on determining working set sizes of applications is relevant to the design of a shared hosting platform that aims at maximizing yield by packing applications on its server pool. A recent paper by Waldspurger investigates ways to overbook memory in consolidated servers using the VMWare ESX virtualization engine [Waldspurger 2002]. Overbooking of memory needs to be carried out more conservatively than that of CPU and network bandwidth since the performance degradation due to even slight overbooking can be significant due to the high penalty of page faults. In a recent research project, two of the authors (Urgaonkar and Shenoy) investigated the problem of dynamically partitioning available memory among co-located applications with the goal of minimizing the unfairness in CPU allocation that can result under memory pressure [Berger et al. 2003]. Extensive research also exists on partitioning disk bandwidth in predictable ways among applications accessing shared storage [Shenoy and Vin 1998; Zhang et al. 2005a, 2005b]. The focus of our work was on CPU and network bandwidth and these pieces of research on managing memory and disk bandwidth are complementary to our work.

6.7 Related Research on Placement/Packing

The placement problem that arises in our research is a variation of well-known knapsack problems. Several similar packing problems have been studied, among others, in the context of placing files in parallel file stores [Lee et al. 2000]; task assignment in a distributed server system [Harchol-Balter 2000]; and store assignment for response time minimization [Verma and Anand 2006].

7. CONCLUDING REMARKS

In this article, we presented techniques for provisioning CPU and network resources in shared hosting platforms running potentially antagonistic third-party applications. We argued that provisioning resources solely based on the worst-case needs of applications results in low average utilization, while provisioning based on a high percentile of the application needs can yield statistical multiplexing gains that significantly increase the utilization of the cluster. Since an accurate estimate of an application's resource needs is necessary when provisioning resources, we presented techniques to profile applications on dedicated nodes, while in service, and used these profiles to guide the placement of application capsules onto shared nodes. We then proposed techniques to overbook cluster resources in a controlled fashion. We conducted an empirical evaluation of the degradation in QoS that such overbooking can result in. Our evaluation enabled us to suggest rules-of-thumb for determining the degree of overbooking that allows a hosting platform to achieve improvements in revenue without compromising the QoS requirements of the hosted applications. Our techniques, in conjunction with commonly used OS resource allocation mechanisms, can provide application isolation and performance guarantees at run-time in the presence of overbooking. We implemented our techniques in a Linux cluster and evaluated them using common server applications. We found that the efficiency benefits from controlled overbooking of resources can be dramatic. Specifically, overbooking resources by as little as 1% increases the utilization of the hosting platform by a factor of 2, while overbooking by 5–10% results in gains of up to 500%. The more bursty the application resources needs, the higher are the benefits of resource overbooking. More generally, our results demonstrate the benefits and feasibility of overbooking resources for the platform provider.

REFERENCES

ABDELZAHER, T., SHIN, K. G., AND BHATTI, N. 2002. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Trans. Parall. Distrib. Syst. 13*, 1 (Jan.).

ANDERSON, J., BERC, L., DEAN, J., GHEMAWAT, S., HENZINGER, M., LUENG, S., VANDERVOORDE, M., WALDSPURGER, C., AND WEIHL, W. 1997. Continuous profiling: Where have all the cycles gone? In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*. ACM, New York, 1–14.

APPLEBY, K., FAKHOURI, S., FONG, L., GOLDSZMIDT, M. K. G., KRISHNAKUMAR, S., PAZEL, D., PERSHING, J., AND ROCHWERGER, B. 2001. Oceano—SLA-based management of a computing utility. In *Proceedings of the IFIP/IEEE Symposium on Integrated Network Management*. IEEE Computer Society Press, Los Alamitos, CA.

ARON, M., DRUSCHEL, P., AND ZWAENEPOEL, W. 2000. Cluster reserves: A mechanism for resource management in cluster-based network servers. In *Proceedings of the ACM SIGMETRICS Conference*. ACM, New York.

ARPACI-DUSSEAU, A. AND ARPACI-DUSSEAU, R. 2001. Information and control in gray-box systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP 2001)*. ACM, New York, 43–56.

ARPACI-DUSSEAU, A. C. 2001. Implicit coscheduling: Coordinated scheduling with implicit information in distributed systems. *ACM Trans. Comput. Syst. 19*, 3, 283–331.

BANGA, G., DRUSCHEL, P., AND MOGUL, J. 1999. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd Symposium on Operating System Design and Implementation (OSDI'99)*. 45–58.

BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBUER, R., PRATT, I., AND WARFIELD, A. 2003. Xen and the art of virtulization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, New York.

BENNANI, M. AND MENASCE, D. 2005. Resource allocation for autonomic data centers using analytic performance models. In *Proceedings of IEEE International Conference on Autonomic Computing (ICAC-05)*. IEEE Computer Society Press, Los Alamitos, CA.

BERGER, E., KAPLAN, S., URGAONKAR, B., SHARMA, P., CHANDRA, A., AND SHENOY, P. 2003. Scheduler-aware virtual memory management. In *Poster at the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*. ACM, New York.

BLANQUER, J., BRUNO, J., MCSHEA, M., OZDEN, B., SILBERSCHATZ, A., AND SINGH, A. 1999. Resource management for QoS in Eclipse/BSD. In *Proceedings of the FreeBSD'99 Conference*.

BOORSTYN, R., BURCHARD, A., LIEBEHERR, J., AND OOTTAMAKORN, C. 2000. Statistical service assurances for traffic scheduling algorithms. *IEEE J. Select. Areas Commun. 18*, 12, 2651–2664.

BURNETT, N., BENT, J., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. 2002. Exploiting gray-box knowledge of buffer-cache management. In *Proceedings of the USENIX Annual Technical Conference*.

CHANDRA, A., ADLER, M., GOYAL, P., AND SHENOY, P. 2000. Surplus fair scheduling: A proportional-share CPU scheduling algorithm for symmetric multiprocessors. In *Proceedings of the 4th Symposium on Operating System Design and Implementation (OSDI 2000)*.

CHANDRA, A., GONG, W., AND SHENOY, P. 2003a. Dynamic resource allocation for shared data centers using online measurements. In *Proceedings of the 11th International Workshop on Quality of Service (IWQoS 2003)*.

CHANDRA, A., GOYAL, P., AND SHENOY, P. 2003b. Quantifying the benefits of resource multiplexing in on-demand data centers. In *Proceedings of the 1st Workshop on Algorithms and Architectures for Self-Managing Systems*.

CHASE, J. AND DOYLE, R. 2001. Balance of power: Energy management for server clusters. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*.

CHASE, J., GRIT, L., IRWIN, D., MOORE, J., AND SPRENKLE, S. 2003. Dynamic virtual clusters in a grid site manager. In *Proceedings of the 12th International Symposium on High Performance Distributed Computing (HPDC-12)*.

CHEN, Y., DAS, A., QIN, W., SIVASUBRAMANIAM, A., WANG, Q., AND NATARAJAN, G. 2005. Managing server energy and operational costs in hosting centers. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2005)*, ACM, New York.

CHEN, Y., DAS, A., WANG, Q., SIVASUBRAMANIAM, A., HARPER, R., AND BLAND, M. 2006. Consolidating clients on back-end servers with co-location and frequency control. In *Poster at the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2006)*. ACM, New York.

CHERKASOVA, L. AND PHAAL, P. 1999. Session based admission control: A mechanism for improving performance of commercial web sites. In *Proceedings of the 7th International Workshop on Quality of Service*. IEEE Computer Society Press, Los Alamitos, CA.

CLARK, C., FRASER, K., HAND, S., HANSEN, J., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. 2005. Live migration of virtual machines. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI'05)*.

COHEN, I., CHASE, J., GOLDSZMIDT, M., KELLY, T., AND SYMONS, J. 2004. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proceedings of the 6th USENIX Symposium in Operating Systems Design and Implementation (OSDI 2004)*.

COOK, W. AND ROHE, A. 1999. Computing minimum-weight perfect matchings. *INFORMS J. Comput.* 138–148.

Corba 2006. Corba documentation. http://www.omg.org/.

CORMEN, T., LEISERSON, C., AND RIVEST, R. 1991. *Introduction to Algorithms*. The MIT Press, Cambridge, MA.

DAVIS, P. 1994. Airline ties profitability to yield management. *SIAM News*.

DOYLE, R., CHASE, J., ASAD, O., JIN, W., AND VAHDAT, A. 2003. Model-based resource provisioning in a web service utility. In *Proceedings of the 4th USITS*.

DUDA, K. J. AND CHERITON, D. R. 1999. Borrowed-virtual-time (BVT) scheduling: Supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*. ACM, New York, 261–276.

EDMONDS, J. 1965. Maximum matching and a polyhedron with 0,1 - Vertices. *J. Rese. NBS 69B*.

ELNIKETY, S., NAHUM, E., TRACEY, J., AND ZWAENEPOEL, W. 2004. A method for transparent admission control and request scheduling in e-commerce web sites. In *Proceedings of the 13th International Conference on World Wide Web*. 276–286.

FOX, A., GRIBBLE, S., CHAWATHE, Y., BREWER, E., AND GAUTHIER, P. 1997. Cluster-based scalable network services. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP'97)*. ACM, New York.

FRANKS, R. 1999. Performance Analysis of Distributed Server Systems. Ph.D. dissertation, Carleton University.

GOVIL, K., TEODOSIU, D., HUANG, Y., AND ROSENBLUM, M. 1999. Cellular disco: Resource management using virtual clusters on Shared-memory Multiprocessors. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'99)*. ACM, New York, 154–169.

GOVINDAN, S., NATH, A., DAS, A., URGAONKAR, B., AND SIVASUBRAMANIAM, A. 2007. Xen and co.: Communication-aware CPU scheduling for consolidated xen-based hosting platforms. In *Proceedings of the 3rd International ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE)*. ACM, New York.

GOYAL, P., GUO, X., AND VIN, H. M. 1996a. A hierarchical CPU scheduler for multimedia operating systems. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation (OSDI'96)*. 107–122.

GOYAL, P., VIN, H. M., AND CHENG, H. 1996b. Start-time fair queuing: A scheduling algorithm for integrated services packet switching networks. In *Proceedings of ACM SIGCOMM '96*. ACM, New York.

GridForum02b 2002. Global grid forum: Scheduling and resource management working group. http://www-unix.mcs.anl.gov/ schopf/ggf-sched/.

GUPTA, A., STAHL, D., AND WHINSTON, A. 1999. The economics of network management. *Commun. ACM 42*, 5, 57–63.

HARCHOL-BALTER, M. 2000. Task assignment with unknown duration. In *Proceedings of the International Conference on Distributed Computing Systems*. 214–224.

HORI, A., TEZUKA, H., ISHIKAWA, Y., SODA, N., KONAKA, H., AND MAEDA, M. 1996. Implementation of gang scheduling on a workstation cluster. In *Proceedings of the IPPS'96 Workshop on Job Scheduling Strategies for Parallel Processing*. 27–40.

IYER, R., TEWARI, V., AND KANT, K. 2000. Overload control mechanisms for web servers. In *Proceedings of the Workshop on Performance and QoS of Next Generation Networks*.

JAMJOOM, H., REUMANN, J., AND SHIN, K. 2000. QGuard: Protecting internet servers from overload. Tech. Rep. CSE-TR-427-00, Department of Computer Science, University of Michigan.

JONES, M. B., ROSU, D., AND ROSU, M. 1997. CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'97)*. ACM, New York, 198–211.

KAMRA, A., MISRA, V., AND NAHUM, E. 2004. Yaksha: A controller for managing the performance of 3-tiered websites. In *Proceedings of the 12th International Workshop on Quality of Service (IWQoS)*. IEEE Computer Society Press, Los Alamitos, CA.

KANODIA, V. AND KNIGHTLY, E. 2000. Multi-class latency-bounded web servers. In *Proceedings of International Workshop on Quality of Service (IWQoS'00)*. IEEE Computer Society Press, Los Alamitos, CA.

KELLY, T., COHEN, I., GOLDSZMIDT, M., AND KEETON, K. 2004. Inducing models of black-box storage arrays. Tech. Rep. HPL-2004, HP Labs.

KNIGHTLY, E. AND SHROFF, N. 1999. Admission control for statistical QoS: Theory and practice. *IEEE Network 13*, 2. 20–29.

KOURNEV, S. AND BUCHMANN, A. 2003. Performance modeling and evaluation of large-scale J2EE applications. In *Proceedings of the International Conference of the Computer Measurement Group*.

LEE, L.-W., SCHEUERMANN, P., AND VINGRALEK, R. 2000. File assignment in parallel I/O systems with minimal variance of service time. *IEEE Trans. Comput. 49*, 2, 127–140.

LESLIE, I., MCAULEY, D., BLACK, R., ROSCOE, T., BARHAM, P., EVERS, D., FAIRBAIRNS, R., AND HYDEN, E. 1996. The design and implementation of an operating system to support distributed multimedia applications. *IEEE J. Selected Areas in Communication, 14*, 7, 1280–1297.

LEVY, R., NAGARAJARAO, J., PACIFICI, G., SPREITZER, M., TANTAWI, A., AND YOUSSEF, A. 2003. Performance management for cluster based web services. In *Proceedings of the IFIP/IEEE 8th International Symposium on Integrated Network Management*. IEEE Computer Society Press, Los Alamitos, CA, Vol. 246. 247–261.

LI, S. AND JAMIN, S. 2000. A measurement-based admission-controlled web server. In *Proceedings of the 9th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2000)*. IEEE Computer Society Press, Los Alamitos, CA.

LIN, C., CHU, H., AND NAHRSTEDT, K. 1998. A soft-real-time scheduling server on the windows NT. In *Proceedings of the 2nd USENIX Windows NT Symposium*.

LITZKOW, M., LIVNY, M., AND MUTKA, M. 1988. Condor—A hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*. 104–111.

LIU, T.-K., KUMARAN, S., AND LUO, Z. 2001. Layered queueing models for Enterprise Java Beans Applications. Tech. rep., IBM. June.

LTT02. The linux toolkit project page. http://www.opensys.com/LTT.

MENASCE, D. 2003. Web Server Software Architectures. *IEEE Internet Comput. 7*.

MENASCE, D., ALMEIDA, V., AND DOWDY, L. 2004. *Performance by Design: Computer Capacity Planning by Example*. Prentice-Hall, Englewood Cliffs, NJ.

MENON, A., COX, A., AND ZWAENEPOEL, W. 2006. Optimizing network virtualization in xen. In *Proceedings of the USENIX Annual Technical Conference (USENIX'06)*.

MENON, A., SANTOS, J., TURNER, Y., JANAKIRAMAN, G., AND ZWAENEPOEL, W. 2005. Diagnosing performance overheads in the xen virtual machine environment. In *Proceedings of the International Conference on Virtual Execution Environments*.

NELSON, M., LIM, B.-H., AND HUTCHINS, G. 2005. Fast transparent migration for virtual machines. In *Proceedings of the 2005 USENIX Annual Technical Conference.* 391–394

PAPOULIS, A. AND PILLAI, S. 2002. *Probability, Random Variables and Stochastic Processes*. McGraw-Hill, Englewood Cliffs, NJ.

pgbench 2002. The pgbench man page, postgresql software distribution.

PRADHAN, P., TEWARI, R., SAHU, S., CHANDRA, A., AND SHENOY, P. 2002. An observation-based approach towards self-managing web servers. In *Proceedings of the 10th International Workshop on Quality of Service (IWQoS 2002)*. IEEE Computer Society Press, Los Alamitos, CA.

RANJAN, S., ROLIA, J., FU, H., AND KNIGHTLY, E. 2002. QoS-driven server migration for internet data centers. In *Proceedings of the 10th International Workshop on Quality of Service (IWQoS)*. IEEE Computer Society Press, Los Alamitos, CA.

ROLIA, J. AND SEVCIK, K. 1995. The method of layers. *IEEE Trans. Softw. Eng. 21*, 8, 689–700.

ROSCOE, T. AND LYLES, B. 2000. Distributing computing without DPEs: Design considerations for public computing platforms. In *Proceedings of the 9th ACM SIGOPS European Workshop*. ACM, New York.

SAITO, Y., BERSHAD, B., AND LEVY, H. 1999. Manageability, availability and performance in porcupine: A highly scalable, cluster-based mail service. In *Proceedings of the 17th Symposium on Operating Systems Principles (SOSP'99)*. ACM, New York.

SAPUNTZAKIS, C., CHANDRA, R., PFAFF, B., CHOW, J., LAM, M. S., AND ROSENBLUM, M. 2002. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*.

Sgi99 1999. React: Irix real-time extensions. http://www.sgi.com/software/react.

SHENDE, S., MALONY, A., CUNY, J., LINDLAN, K., BECKMAN, P., AND KARMESIN, S. 1998. Portable profiling and tracing for parallel scientific applications using C++. In *Proceedings of ACM SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT)*. ACM, New York. 134–145.

SHENOY, P. AND VIN, H. 1998. Cello: A disk scheduling framework for next generation operating systems. In *Proceedings of ACM SIGMETRICS Conference*. ACM, New York. 44–55.

SLOTHOUBER, L. 1996. A model of web server performance. In *Proceedings of the 5th International World Wide Web Conference*.

SMITH, B. C., LEIMKUHLER, J. F., AND DARROW, R. M. 1992. Yield management at American Airlines. *Interfaces, 22*, 1, 8–31.

SPECWeb99. The Standard Performance Evaluation Corporation (SPEC). http://www.spec.org/.

STEWART, C. AND SHEN, K. 2005. Performance modeling and system management for multi-component online services. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation*.

Sun98b 1998. Solaris resource manager 1.0: Controlling system resources effectively. http://www.sun.com/software/white-papers/wp-srm.

SUNDARAM, V., CHANDRA, A., GOYAL, P., SHENOY, P., SAHNI, J., AND VIN, H. 2000. Application performance in the QLinux multimedia operating system. In *Proceedings of the 8th ACM Conference on Multimedia*. ACM, New York.

TANG, P. AND TAI, T. 1999. Network traffic characterization using token bucket model. In *Proceedings of IEEE Infocom'99*. IEEE Computer Society Press, Los Alamitos, CA.

URGAONKAR, B., PACIFICI, G., SHENOY, P., SPREITZER, M., AND TANTAWI, A. 2005a. An analytical model for multi-tier internet services and its applications. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2005)*. ACM, New York.

URGAONKAR, B. AND SHENOY, P. 2004a. Cataclysm: Handling extreme overloads in internet services. In *Proceedings of the 23rd Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 2004)*. ACM, New York.

URGAONKAR, B. AND SHENOY, P. 2004b. Sharc: Managing CPU and network bandwidth in shared Clusters. *IEEE Transactions on Parallel and Distributed Systems, 15*, 1, 2–17.

URGAONKAR, B. SHENOY, P., CHANDRA, A., AND GOYAL, P. 2005b. Dynamic provisioning of multi-tier internet applications. In *Proceedings of the 2nd IEEE International Conference on Autonomic Computing (ICAC-05)*. IEEE Computer Society Press, Los Alamitos, CA.

URGAONKAR, B., SHENOY, P., AND ROSCOE, T. 2002. Resource overbooking and application profiling in shared hosting platforms. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2002)*.

VERGHESE, B., GUPTA, A., AND ROSENBLUM, M. 1998. Performance isolation: Sharing and isolation in shared-memory multiprocessors. In *Proceedings of ASPLOS-VIII*. 181–192.

VERMA, A. AND ANAND, A. 2006. On store placement for response time minimization in parallel disks. In *Proceedings of ICDCS'0*, 31.

VERMA, A. AND GHOSAL, S. 2003. On admission control for profit maximization of networked service providers. In *Proceedings of the 12th International World Wide Web Conference (WWW2003)*.

VILLELA, D., PRADHAN, P., AND RUBENSTEIN, D. 2004. Provisioning servers in the application tier for e-commerce systems. In *Proceedings of the 12th International Workshop on Quality of Service (IWQoS)*. IEEE Computer Society Press, Los Alamitos, CA.

VIN, H. M., GOYAL, P., GOYAL, A., AND GOYAL, A. 1994. A statistical admission control algorithm for multimedia servers. In *Proceedings of the ACM Multimedia'94*. ACM, New York, 33–40.

VOIGT, T., TEWARI, R., FREIMUTH, D., AND MEHRA, A. 2001. Kernel mechanisms for service differentiation in overloaded web servers. In *Proceedings of USENIX Annual Technical Conference*.

WALDSPURGER, C. 2002. Memory resource management in VMWare ESX server. In *Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI'02)*.

WALDSPURGER, C. A. AND WEIHL, W. E. 1994. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation (OSDI'94)*.

WELSH, M. AND CULLER, D. 2003. Adaptive overload control for busy internet servers. In *Proceedings of the 4th USENIX Conference on Internet Technologies and Systems (USITS'03)*.

WHITAKER, A., SHAW, M., AND GRIBBLE, S. D. 2002. Scale and performance in the denali isolation kernel. In *Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI'02)*.

WOODSIDE, C. AND RAGHUNATH, G. 1995. General bypass architecture for high-performance distributed algorithms. In *Proceedings of the 6th IFIP Conference on Performance of Computer Networks*.

WSLA. Web service level agreements (wsla) project. http://www.research.ibm.com/wsla.

XU, J., OUFIMTSEV, A., WOODSIDE, M., AND MURPHY, L. 2006. Performance modeling and prediction of enterprise JavaBeans with layered queuing network templates. *SIGSOFT Softw. Eng. Notes 31*, 2.

XU, W., BODIK, P., AND PATTERSON, D. 2004. A flexible architecture for statistical learning and data mining from system log streams. In *Proceedings of Workshop on Temporal Data Mining: Algorithms, Theory and Applications at the 4th IEEE International Conference on Data Mining (ICDM'04)*.

ZHANG, J., SIVASUBRAMANIAM, A., RISKA, A., WANG, Q., AND RIEDEL, E. 2005a. An interposed 2-level I/O scheduling framework for performance virtualization. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2005)*. ACM, New York.

ZHANG, J., SIVASUBRAMANIAM, A., WANG, Q., RISKA, A., AND RIEDEL, E. 2005b. Storage performance virtualization via throughput and latency control. In *Proceedings of MASCOTS*.

Received October 2006; revised May 2007; accepted January 2008