Resource Overbooking and Application Profiling in Shared Hosting Platforms [*]

Bhuvan Urgaonkar, **Prashant Shenoy** and **Timothy Roscoe**†

Department of Computer Science,
University of Massachusetts
Amherst MA 01003
{bhuvan, shenoy}@cs.umass.edu

†Intel Research at Berkeley
2150 Shattuck Avenue Suite 1300
Berkeley CA 94704
troscoe@intel-research.net

Abstract

In this paper, we present techniques for provisioning CPU and network resources in shared 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 platforms, to maximize the platform yield*: the revenue generated by the available resources. We do this by first deriving an accurate estimate of application resource needs by profiling applications on dedicated nodes, and then using these profiles to guide the placement of application components onto shared nodes. By overbooking cluster resources in a controlled fashion, our platform can provide performance guarantees to applications even when overbooked, and combine these techniques with commonly used QoS resource allocation mechanisms to provide application isolation and performance guarantees at run-time. When compared to provisioning based on the worst-case, the efficiency (and consequently revenue) benefits from controlled overbooking of resources can be dramatic. Specifically, experiments on our Linux cluster implementation indicate that overbooking resources by as little as 1% 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.*

1 Introduction and Motivation

Server clusters built using commodity hardware and software are an increasingly attractive alternative to traditional large multiprocessor servers for many applications, in part due to rapid advances in computing technologies and falling hardware prices.

This paper addresses challenges in the design of a type of server cluster 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-party 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 may overlap. Whereas dedicated hosting platforms are used for many niche applications that warrant their additional cost, 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 *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 [17]. The central challenge to the platform provider is thus one of 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. 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 re-

[*]Portions of this research were done when Timothy Roscoe was a researcher at Sprint ATL and Bhuvan Urgaonkar was a summer intern at Sprint ATL. This research was supported in part by NSF grants CCR-9984030, EIA-0080119 and a gift from Sprint Corporation.

sources. The second approach shares 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.

Recently, several resource management mechanisms for shared hosting platforms have been proposed [1, 3, 7, 23]. This paper reports work performed in this context, but with two significant differences in goals.

Firstly, we seek from the outset to support a diverse set of potentially antagonistic network services simultaneously on a platform. The services will therefore have heterogeneous resource requirements; web servers, continuous media processors, and multiplayer game engines all make different demands on the platform in terms of resource bandwidth and latency. We evaluate our system with such a diverse application mix.

Secondly, we aim to support resource management policies based on *yield management* techniques such as those employed in the airline industry [19]. Yield management is driven by the business relationship between a platform provider and many application providers, and results in different short-term goals. In traditional approaches the most important aim is to satisfy all resource contracts while making efficient use of the platform. Yield management by contrast is concerned with ensuring that as much of the available resource as possible is used to generate revenue, rather than being utilized "for free" by a service (since it would otherwise be idle).

An analogy with air travel may clarify the point: instead of trying to ensure that every ticketed passenger gets to board their chosen flight, we try to ensure that no plane takes off with an empty seat (which is achieved by overbooking seats).

An immediate consequence of this goal is our treatment of "flash crowds"—a shared hosting platform should react to an unexpected high demand on an application only if there is an economic incentive for doing so. That is, the platform should allocate additional resources to an application only if it enhances revenue. Further, any increase in resource allocation of an application to handle unexpected high demands should not be at the expense of contract violations for other applications, since this is economically undesirable.

1.1 Research Contributions

The contribution of this paper 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, and how this can be usefully done without adversely impacting the guarantees offered to application providers. Thirdly, we show how untrusted and/or mutually antagonistic applications in the platform can be isolated from one another. The rest of this section presents these contributions in detail.

Automatic derivation of QoS requirements: We discuss techniques for empirically deriving an application's resource needs. The effectiveness of a resource management technique is crucially dependent on the ability to reserve appropriate resources for each application—overestimating an application's resource needs can result in idling of resources, while underestimating them can degrade application performance. Consequently a shared hosting platform can significantly enhance its utility to users by automatically deriving the 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.

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: We discuss resource overbooking techniques and application placement strategies for shared hosting platforms. Provisioning cluster resources solely based on the worst-case needs of an application results in low average utilization, since the average resource requirements of an application are typically smaller than its worst case (peak) requirements, and resources tend to idle when the application does not utilize its peak reserved share. In contrast, provisioning a cluster based on a high *percentile* of the application needs yields statistical multiplexing gains that significantly increase the average utilization of the cluster at the expense of a small amount of overbooking, and increases the number of applications that can be supported on a given hardware configuration.

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 (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.

We demonstrate the feasibility and benefits of overbooking resources in shared hosting platforms, and propose techniques to overbook (i.e. under-provision) 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.

Placement and isolation of antagonistic applications: We describe an additional aspect of the resource management problem: placement and isolation of antagonistic applications. We assume that third-party applications may be antagonistic cannot be trusted by the platform, due either to malice or bugs. Our work demonstrates how untrusted third-party applications can be isolated from one another in shared hosting platforms in two ways. 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, we present automated placement techniques that allow a platform provider to exert sufficient control over the placement of application components onto nodes in the cluster, since manual placement of applications is unfeasibly complex and error-prone in large clusters.

1.2 System Model and Terminology

The shared hosting platform assumed in our research consists of a cluster of *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 (see Figure 1). Each cluster node is assumed to run an operating system kernel that supports some notion of quality of service such as reservations or shares. In this paper, we focus on managing CPU and network interface bandwidth in shared hosting platforms. As [1] points out, management of other resources which are inherently temporal in nature, such as disk bandwidth, can be performed by similar mechanisms. Spatial resources, in particular physical memory, present a different challenge. A straightforward approach is to use static partitioning as in [1], although recently more sophisticated approaches have been implemented [5].

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 the 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 colocated 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 (see Figure 1).

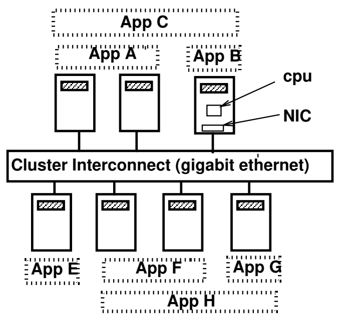


Figure 1: Architecture of a shared hosting platform. Each application runs on one or more nodes and shares resources with other applications.

The rest of this paper 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. Section 6 discusses related work, and 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 [4, 12, 13], the problem of *how much* resource to allocate to each application has received relatively little attention. In this section, we address this issue by proposing techniques to automatically derive the QoS requirements of an application (the terms resource requirements and QoS requirements are used interchangeably in this paper.) 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 paper, 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) [22]. 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).

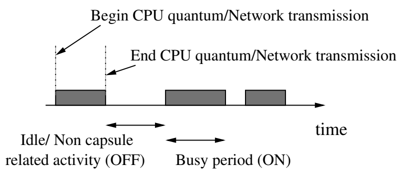
In general, we assume that parameters τ and O are specified by the application provider. This may be based on a contract between the platform provider and the application provider (e.g., the more the application provider is willing to pay for resources, the stronger are the provided guarantees), or on 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). In the rest of this section, we show how to derive the remaining three parameters σ, ρ and U using profiling, given values of τ and O.

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 to run-time profiling using specially linked libraries have been proposed.

We use kernel-based profiling mechanisms in the context of shared hosting platforms, for a number of reasons. Firstly, 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. Secondly, 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).



Begin CPU quantum/Network transmission
End CPU quantum/Network transmission
time
Idle/ Non capsule related activity (OFF)
Busy period (ON)

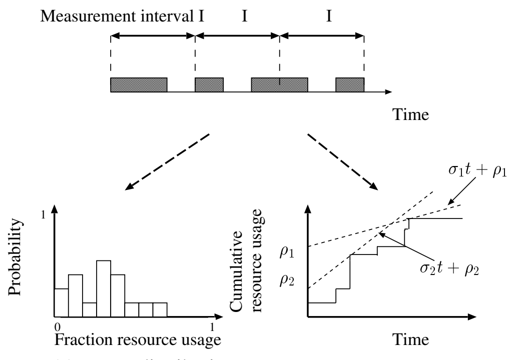
Figure 2: An example of an On-Off trace.

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 generation techniques.

We use the Linux trace toolkit as our kernel profiling mechanism [14]. 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 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



(a) Usage distribution (b) Token bucket parameters

Figure 3: Derivation of the usage distribution and token bucket parameters.

from this information (see Figure 2). 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).

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 3(a) illustrates this process.

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 3(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 3(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 [22] 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 95^{th} 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 100Mbps 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 non-essential system services (e.g., mail services, X windows server) are turned off to prevent interference during profiling. The parameters τ and $\mathcal{I}$ were both set to 1 sec in all our experimentation. We profile the following server applications in our experiments:

- *Apache web server:* We use the SPECWeb99 benchmark [20] 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.

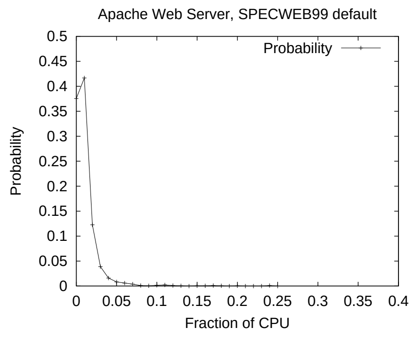
- *MPEG streaming media server:* We use a home-grown streaming server to stream MPEG-1 video files to multiple concurrent clients over UDP. Each client in our experiment requests a 15 minute long variable bit rate MPEG-1 video with a mean bit rate of 1.5 Mb/s. We vary the number of concurrent clients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

- *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.

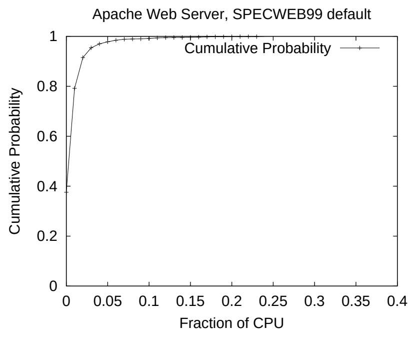
- *PostgreSQL database server:* We profile the postgreSQL database server (version 7.2.1) using the *pgbench 1.2* benchmark. This benchmark is part of the postgreSQL distribution and emulates the TPC-B transactional benchmark [16]. The benchmark provides control over the number of concurrent clients 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.

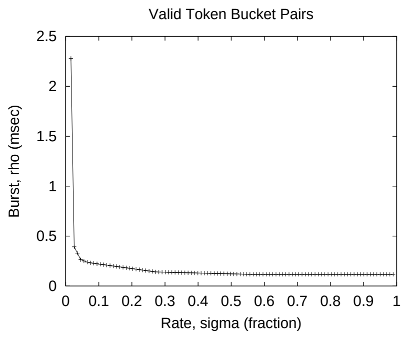
Figure 4(a) depicts the CPU usage distribution of the Apache web server obtained using the default settings of the SPECWeb99 benchmark (50 concurrent clients, 30% dynamic cgi-bin requests). Figure 4(b) plots the corresponding cumulative distribution function (CDF) of the resource usage. As shown in the figure (and summarized in Table 1), the worst case CPU usage (100^{th} percentile) is 25% of CPU capacity. Further, the 99^{th} and the 95^{th} 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 99^{th} and 95^{th} percentiles, respectively, instead of the 100^{th} profile). Thus, even small amounts of overbooking can potentially yield significant increases in platform capacity. Figure 4(c) depicts the possible valid (σ, ρ) pairs for Apache's CPU usage. Depending on the specified overbooking tolerance O, we can set σ to an appropriate



(a) Probability distribution (PDF)



(b) Cumulative distribution function (CDF)



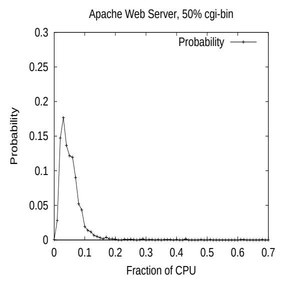
(c) Token bucket parameters

Figure 4: Profile of the Apache web server using the default SPECWeb99 configuration.

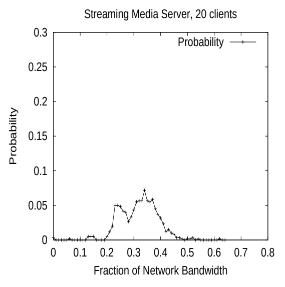
percentile of the usage distribution U, and the corresponding ρ can then be chosen using this figure.

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 1 summarizes our results and also presents profiles for several additional scenarios (only a small subset of the three dozen profiles obtained from our experiments are presented due to space constraints). Table 1 also lists the worst-case resource needs as well as the 99^{th} and the 95^{th} percentile of the resource usage.

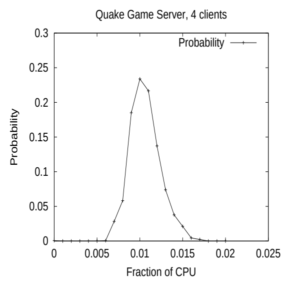
Together, Figure 5 and Table 1 demonstrate that all server applications exhibit burstiness in their resource usage, albeit to different degrees. This burstiness causes the worst-case resource needs to be significantly higher than a high percentile of the usage distribution. Consequently, we find that the 99^{th} percentile is smaller by a factor of 1.1-2.5, while the 95^{th} percentile yields a factor of 1.3-6.25 reduction when compared to the 100^{th} percentile. Together, these results illustrate the potential gains that can be realized by overbooking resources in shared hosting platforms.



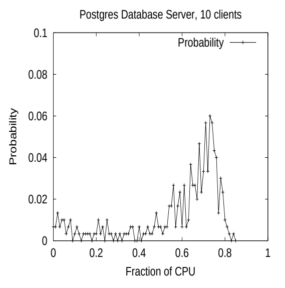
(a) Apache: dynamic requests



(b) Streaming media server



(c) Quake Server



(d) PostgreSQL Server

Figure 5: Profiles of Various Server Applications

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. 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 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:

Application	Res.	Res. usage at percentile			(σ, ρ)
		100^{th}	99^{th}	95^{th}	for $O = 0.01$
WS,default	CPU	0.25	0.10	0.04	(0.10, 0.218)
WS, 50% dyn.	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)

Table 1: Summary of profiles. Although we profiled both CPU and network usage for each application, we only present results for the more constraining resource due to space constraints. Abbreviations: WS=Apache, SMS=streaming media server, GS=Quake game server, DBS=database server, k=number of clients, dyn.=dynamic, Res.=Resource.

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:

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

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. [1]

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,

$$Pr(Y > C) \leq O_i \ \ \forall i \qquad (2)$$

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,

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

where $Pr(Y > C) = \sum_{x=C}^{\infty} Y(x)$. Note that Equation 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 Equations 1 and 3 are satisfied for both the CPU and network interface.

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 Equations 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 colocated).

The placement of capsules onto nodes subject to the above constraint can be handled as follows. We model the

[1]Note that since the σ_i for capsule i was chosen based on the $(1 - O_i) * 100^{th}$ percentile of the capsule's resource usage distribution, this multiplication with $(1 - O_i)$ may seem like penalizing the capsule twice. However, this is not so because σ_i in combination with the burst ρ_i is an upper envelop of the requirements of capsule i. The multiplication with $(1 - O_i)$ allows us to overbook the resources on a node in a controlled manner.

[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 i^{th} term represents the probability that the random variable equals i (i.e., $U(i)$). If $U_1, U_2, ..., U_{k+1}$ 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 $U_1, U_2, \cdots, U_{k+1}$.

placement problem using a graph that contains a vertex for each of the m capsules and N nodes. We add an edge between a capsule and 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 edge connects a capsule to a node.

Given such a graph, we use the following algorithm to determine a placement. The algorithm starts with the capsule that is most constrained (i.e., has the least number of edges/feasible nodes) and places it on any one of its feasible nodes. The node and all of its edges are deleted (since no other capsule can be placed on it). The algorithm then picks the next most constrained capsule and repeats the above process until all m capsules are placed onto nodes. It can be shown that such a greedy algorithm will *always find a placement if one exists* (see an extended version of this paper [24] for a formal proof). This property is used as follows — if no node is found to place a capsule, the algorithm terminates declaring that no placement exists for the application. Further, the algorithm is efficient, since capsules can be placed in a single linear scan once they are sorted in the increasing order of out-degree, resulting in an overall complexity of $O(m \cdot \log m)$.

In our description of the placement algorithm above, we left unspecified how a node is chosen for a capsule out of the many possible feasible nodes. The choice of a particular feasible node can have important implications on the total number of applications supported by the platform. Consequently, we consider four policies for making this decision. The first policy is random, where we pick one feasible node randomly. The second policy is best-fit, where we choose the feasible node which has the least unused resources (i.e., constitutes the best fit for the capsule). The third 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 fourth 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 Equation 3, colocating 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.

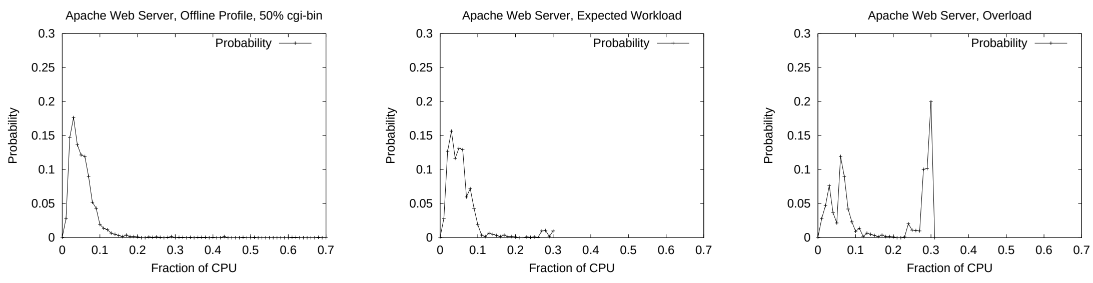
3.3 Handling Dynamically Changing Resource Requirements

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. In this section we outline our approach for dealing with dynamically changing application workloads.

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), 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



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

Figure 6: Demonstration of how an application overload may be detected by comparing the latest resource usage profile with the original offline profile.

overload would manifest itself through an increased concentration in the high percentile buckets of the resource usage distributions of its capsules.

We present the results of a simple experiment to illustrate this. Figure 6(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 99^{th} 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 6(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 6(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.

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 [4, 12, 13]—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 consider three commonly-used QoS-aware mechanisms — reservations, proportional-share schedulers and rate regulators. Due to space constraints, we present only the mapping for reservations here and point the reader to an extended version of this paper [24] for the remaining mappings.

A reservation-based scheduler [12, 13] requires the resource requirements to be specified as a pair (x, y) where the capsule desires x units of the resource every y time units (effectively, the capsule requests $\frac{x}{y}$ fraction of the resource). 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 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 resource 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 $y_i = \rho_i/\sigma_i$.

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. 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).

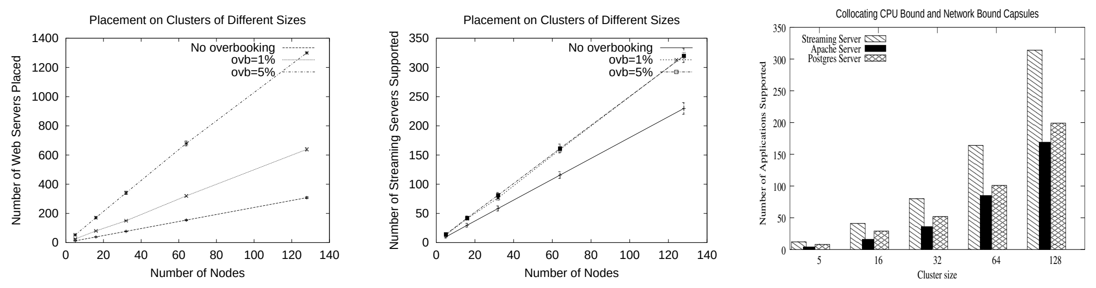
The third component, namely the QoS-enhanced Linux kernel, runs on each platform node and is responsible for enforcing the QoS requirements of capsules at run time. For the purposes of this paper, we implement the H-SFQ proportional-share CPU scheduler [10]. H-SFQ is a *hierarchical* proportional-share scheduler that allows us to group resource principals (processes, lightweight processes) and assign an aggregate CPU share to the entire group. 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 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.

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 paper. For ease of comparison, we use the same set of applications discussed in 2.4 and their derived profiles (see Table 1) for our experimental study.

5.1 Efficacy of Resource Overbooking

Our first set of experiments examine the efficacy of overbooking resources in shared hosting platforms. We first consider 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 1; the other two employed varying mixes of static and dynamic



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

Figure 7: Benefits of resource overbooking for a bursty web server application, a less bursty streaming server application and for application mixes.

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 7(a) 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 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 1, 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 7(b) 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 1 (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 colocating 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 colocating 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 and non-bursty applications. Figure 7(c) 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, colocating applications with different bottleneck resources and different amounts of burstiness enhance additional statistical multiplexing benefits when overbooking resources.

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 that can be placed on a given platform by different placement strategies. Figure 8(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 1 that the game server profiles exhibit less diversity than a mix of web and database servers. Figure 8(b) shows that, due to the similarity in the application resource requirements, all policies are able to place a comparable number of game servers.

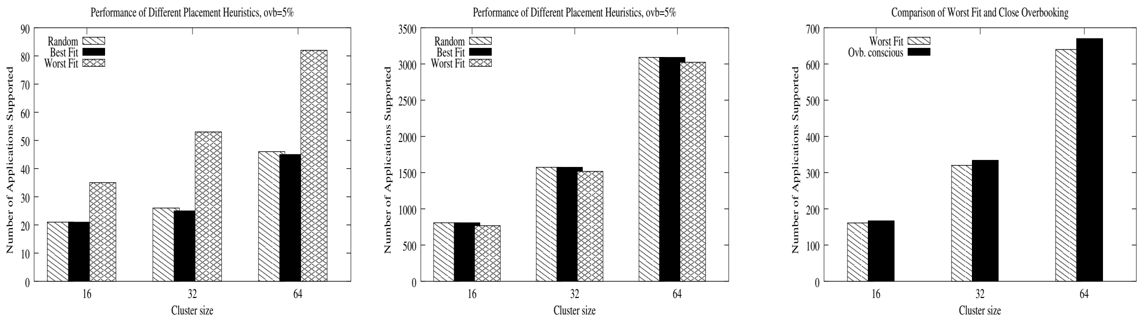
Finally, we examine the effectiveness of taking the overbooking tolerance into account when making placement decisions. We compare the worst-fit policy to an overbooking-conscious worst-fit policy. The latter policy chooses the three worst-fits among all feasible nodes and picks the node that best matches 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 8(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 100^{th} percentile of its usage (no overbooking) and assign the remaining capacity to a greedy *dhrystone* application. 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 99^{th} and the 95^{th} percentiles, allocate the remaining capacity to the dhrystone application, and measure the server throughput. Table 2 depicts our results. As shown, provisioning based on the 100^{th} percentile yields performance that is comparable to running the application on an dedicated node. Provisioning based on the 99^{th} and 95^{th} percentiles results in a small degradation in throughput, but well within the permissible limits of 1% and 5% degradation, respectively, due to overbooking. Table 2 also shows that provisioning based on the average resource requirements results in a substantial fall in throughout, 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 application. In both cases, we first run the application on an isolated node and then on our QoS-enhanced kernel with provisioning based on the 100^{th}, 99^{th} and the 95^{th} percentiles. We also run the application with provisioning based on the average of its resource usage distribution obtained via offline profiling. We measure the throughput in transaction/s for the database server and the mean length of a playback violation (in seconds) for the streaming media server. Table 2 plots our results. Like with the web server, provisioning based on the 100^{th} percentile yields performance comparable to running the application on an isolated node, while a small amount of overbooking results in a corresponding small amount of degradation in application performance.

For each of the above scenarios, we also computed the application profiles in the presence of background load and overbooking and compared these to the profiles gathered on the isolated node. Figure 9 shows one such set of profiles. It should be seen in combination with the second row in Table 2 that corresponds to the PostgreSQL application. Together, they depict the performance of the



(a) Placing diverse applications (b) Placing similar applications (c) Overbooking-conscious placement

Figure 8: Performance of various capsule placement strategies.

Application	Metric	Isolated Node	100^{th}	99^{th}	95^{th}	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 ± 0.85
Streaming	Length of violations (sec)	0	0	0.31 ± 0.04	0.59 ± 0.05	5.23 ± 0.22

Table 2: Effectiveness of kernel resource allocation mechanisms. All results are shown with 95% confidence intervals.

database server for different levels of CPU provisioning. Figures 9(b) and (c) show the CPU profiles of the database server when it is provisioned based on the 99^{th} and the 95^{th} percentiles respectively. As can be seen, the two profiles look similar to the original profile shown in Figure 9(a). Correspondingly, Table 2 shows that for these levels of CPU provisioning, the throughput received by the database server is only slightly inferior to that on an isolated node. This indicates that upon provisioning resources based on a high percentile, the presence of background load interferes minimally with the application behavior. In Figure 9(d), we show the CPU profile when the database server was provisioned based on its average CPU requirement. This profile is drastically different from the original profile. We also present the corresponding low throughput in Table 2. This reinforces our earlier observation that provisioning resources based on the average requirements can result in significantly degraded performance.

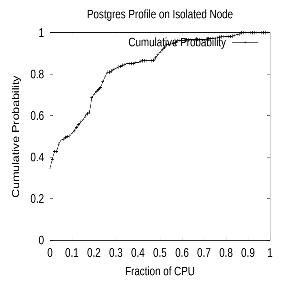
Together, these results demonstrate that our kernel resource allocation mechanisms are able to provide quantitative performance guarantees even when resources are overbooked.

6 Related Work

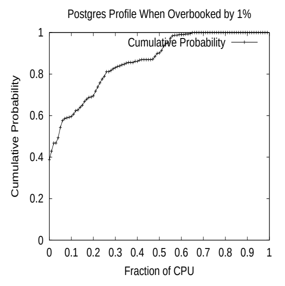
Research on clustered environments over the past decade has spanned a number of issues. Systems such as Condor have investigated techniques for harvesting idle CPU cycles on a cluster of workstations to run batch jobs [15]. The design of scalable, fault-tolerant network services running on server clusters has been studied in [8]. Use of virtual clusters to manage resources and contain faults in large multiprocessor systems has been studied in [9]. Scalability, availability and performance issues in dedicated clusters have been studied in the context of clustered mail servers [18] and replicated web servers [3]. Ongoing efforts in the grid computing community have focused on developing standard interfaces for resource reservations in clustered environments [11]. In the context of QoS-aware resource allocation, numerous efforts over the past decade have developed predictable resource allocation mechanisms for single machine environments [4, 12, 13].

Statistical admission control techniques that overbook resources have been studied in the context of video-on-demand servers [25] and ATM networks [6], but little work as been published to date in the context of shared, cluster-based hosting platforms.

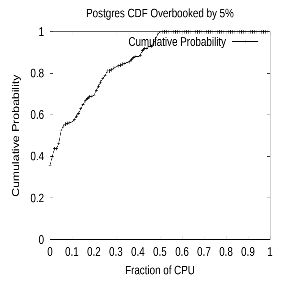
Most closely related to our work is Aron [1, 2], who presents a comprehensive framework for resource management in web servers, with the aim of delivering predictable QoS and differentiated services. New services are profiled by running on lightly-loaded machines, and contracts subsequently negotiated in terms of application level performance (connections per second), reported by the application to the system. CPU and disk bandwidth are scheduled by lottery scheduling [26] and SFQ [10] respectively, while physical memory is statically partitioned between services with free pages allocated temporarily to services that can make use of them. A *resource monitor* running over a longer timescale examines performance



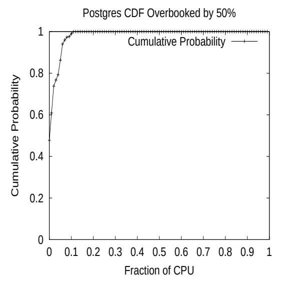
(a) Profile on isolated node



(b) Provision using 99^{th} %-tile



(c) Provision using 95^{th} %-tile



(d) Provision using average

Figure 9: Effect of different levels of provisioning on the PostgreSQL server CPU profile.

reported by the application and system performance information and flags conditions which might violate contracts, to allow extra resources to be provided by external means.

In Aron's system, resource allocation is primarily driven by application feedback and the primary concern is allowing a principal to meet its contract. It is instructive to compare this with our own goal of maximizing the *yield* in the system, which amounts to maximizing the proportion of system resources used to satisfy contracts. Over a long time period (over which new machines can be provisioned, for instance) these goals coincide, but as [19] makes clear, over short periods of time they do not. This difference corresponds to a different kind of relationship between service provider and platform provider. Consequently, Aron's system is able to take advantage of uniform application-reported performance metrics; ours in contrast is oriented toward a heterogeneous mixture of services which are untrusted by the platform and potentially antagonistic.

This subtle but important difference also motivates other differences in design choices between the two systems. In Aron's work, application overload is detected when resource usage exceeds some predetermined threshold. We, on the other hand, detect overload by observing the tail of the recent resource usage distributions. On the other hand, several features of Aron's work, such as the use of multiple random variables to capture the behavior of services over different time scales, are directly applicable to our system.

The specific problem of QoS-aware resource management for clustered environments has been investigated in [3]. This effort builds upon single node QoS-aware resource allocation mechanisms and propose techniques to extend their benefits to clustered environments. [7] proposes a system called Muse for provisioning resources in hosting centers based on energy considerations. Muse is based on an economic approach to managing shared server resources in which services "bid" for resources as a function of delivered performance. It also provides mechanisms to continuously monitor load and compute new resources by estimating the value of their effects on service performance. economic approach for sharing resources in such driven by energy considerations. A salient difference between Muse and our approach is that Muse provisions resources based on the *average* resource requirements whereas we provision based on the *tail* of the resource requirements.

In [27], the authors consider a model of hosting platforms different from that considered in our work. They visualize future applications executing on platforms constructed by clustering multiple, autonomous distributed servers, with resource access governed by agreements between the owners and the users of these servers. They present an architecture for distributed, coordinated enforcement of resource sharing agreements based on an application-independent way to represent resources and agreements. In this work we have looked at hosting platforms consisting of servers in one location and connected by a fast network. However we also believe that distributed hosting platforms will become more popular and resource management in such systems will pose several challenging research problems.

7 Concluding Remarks

In this paper, 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, possibly while in service, and used these profiles to guide the placement of application components onto shared nodes. We then proposed techniques to overbook cluster resources in a controlled fashion such that the platform can provide performance guarantees to applications even when overbooked. 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 when compared to provisioning resources based on the worst-case requirements of applications. 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.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd Peter Druschel for their comments.

References

[1] M. Aron. Differentiated and Predictable Quality of Service in Web Server Systems. *PhD Thesis, Computer Science, Rice University*, October 2000.

[2] M. Aron, S. Iyer, and P. Druschel. A Resource Management Framework for Predictable Quality of Service in Web Servers. Submitted for publication

[3] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster Reserves: A Mechanism for Resource Management in Cluster-based Network Servers. In *Proceedings of the ACM SIGMETRICS Conference, Santa Clara, CA*, June 2000.

[4] G. Banga, P. Druschel, and J. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the third Symposium on Operating System Design and Implementation (OSDI'99), New Orleans*, pages 45–58, February 1999.

[5] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI'02), Boston, MA*, December 2002.

[6] R. Boorstyn, A. Burchard, J. Liebeherr, and C. Oottamakorn. Statistical Service Assurances for Traffic Scheduling Algorithms. *IEEE Journal on Selected Areas in Communications*, 18(12):2651–2664, December 2000.

[7] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle. Managing Energy and Server Resources in Hosting Centers. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 103–116, October 2001.

[8] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based Scalable Network Services. In *Proceedings of the sixteenth ACM symposium on Operating systems principles (SOSP'97), Saint-Malo, France*, pages 78–91, December 1997.

[9] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular Disco: Resource Management using Virtual Clusters on Shared-memory Multiprocessors. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'99), Kiawah Island Resort, SC*, pages 154–169, December 1999.

[10] P. Goyal, X. Guo, and H.M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Proceedings of Operating System Design and Implementation (OSDI'96), Seattle*, pages 107–122, October 1996.

[11] Global Grid Forum: Scheduling and Resource Management Working Group. http://www-unix.mcs.anl.gov/ schopf/ggf-sched, 2002.

[12] M B. Jones, D Rosu, and M Rosu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proceedings of the sixteenth ACM symposium on Operating Systems Principles (SOSP'97), Saint-Malo, France*, pages 198–211, December 1997.

[13] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communication*, 14(7):1280–1297, September 1996.

[14] Linux Trace Toolkit Project Page. http://www.opersys.com/LTT/, 2002.

[15] M. Litzkow, M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, June 1988.

[16] *The pgbench man page, PostgreSQL software distribution*, 2002.

[17] T. Roscoe and B. Lyles. Distributing Computing without DPEs: Design Considerations for Public Computing Platforms. In *Proceedings of the 9th ACM SIGOPS European Workshop, Kolding, Denmark*, September 2000.

[18] Y. Saito, B. Bershad, and H. Levy. Manageability, Availability and Performance in Porcupine: A Highly Available, Scalable Cluster-based Mail Service. In *Proceedings of the 17th SOSP, Kiawah Island Resort, SC*, pages 1–15, December 1999.

[19] B C. Smith, J F. Leimkuhler, and R M. Darrow. Yield Management at American Airlines. *Interfaces*, 22(1):8–31, January-February 1992.

[20] The Standard Performance Evaluation Corporation (SPEC), http://www.spec.org. *SPECWeb99 Benchmark Documentation.*

[21] V Sundaram, A. Chandra, P. Goyal, P. Shenoy, J Sahni, and H Vin. Application Performance in the QLinux Multimedia Operating System. In *Proceedings of the Eighth ACM Conference on Multimedia, Los Angeles, CA*, November 2000.

[22] P. Tang and T. Tai. Network Traffic Characterization Using Token Bucket Model. In *Proceedings of IEEE Infocom'99, New York, NY*, March 1999.

[23] B. Urgaonkar and P. Shenoy. Sharc: Managing CPU and Network Bandwidth in Shared Clusters. Technical Report TR01-08, Department of Computer Science, University of Massachusetts, October 2001.

[24] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource Overbooking and Application Profiling in Shared Hosting Platforms. Technical report TR02-21, Department of Computer Science, University of Massachusetts, May 2002.

[25] H. M. Vin, P. Goyal, A. Goyal, and A. Goyal. A Statistical Admission Control Algorithm for Multimedia Servers. In *Proceedings of the ACM Multimedia'94, San Francisco*, pages 33–40, October 1994.

[26] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-share Resource Management. In *Proceedings of symposim on Operating System Design and Implementation*, November 1994.

[27] T. Zhao and V. Karmacheti. Enforcing Resource Sharing Agreements among Distributed Server Clusters. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS)*, April 2002.