AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers

Anshul Gandhi[*] **Mor Harchol-Balter**[*]
Ram Raghunathan[*] **Michael Kozuch**[†]

April 2012
CMU-CS-12-109

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[*]Carnegie Mellon University, Pittsburgh, PA, USA
[†]Intel Labs, Pittsburgh, PA, USA

This research was sponsored by the National Science Foundation under grant number CNS-1116282 and Intel Science and Technology Center on Cloud Computing. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Data centers; power management; resource provisioning

Abstract

Energy costs for data centers continue to rise, already exceeding $15 billion yearly. Sadly much of this power is *wasted*. Servers are only busy 10-30% of the time on average, but they are often left on, while idle, utilizing 60% or more of peak power when in the idle state.

We introduce a dynamic capacity management policy, *AutoScale*, that greatly reduces the number of servers needed in data centers driven by unpredictable, time-varying load, while meeting response time SLAs. *AutoScale* scales the data center capacity, adding or removing servers as needed. *AutoScale* has two key features: (i) it autonomically maintains just the right amount of spare capacity to handle bursts in the request rate; and (ii) it is *robust* not just to changes in the request rate of real-world traces, but also *request size* and *server efficiency*.

We evaluate our dynamic capacity management approach via implementation on a 38-server multi-tier data center, serving a web site of the type seen in Facebook or Amazon, with a key-value store workload. We demonstrate that *AutoScale* vastly improves upon existing dynamic capacity management policies with respect to meeting SLAs and robustness.

1 Introduction

Many networked services, such as Facebook and Amazon, are provided by multi-tier data center infrastructures. A primary goal for these applications is to provide good response time to users; these response time targets typically translate to some response time Service Level Agreements (SLAs). In an effort to meet these SLAs, data center operators typically over-provision the number of servers to meet their estimate of peak load. These servers are left "always on," leading to only 10-30% server utilization [2, 3]. In fact, [35] reports that the average data center server utilization is only 18% despite years of deploying virtualization aimed at improving server utilization. Low utilization is problematic because servers that are on, while idle, still utilize 60% or more of peak power.

To reduce wasted power, we consider intelligent *dynamic capacity management*, which aims to match the number of active servers with the current load, in situations where future load is unpredictable. Servers which become idle when load is low could be either *turned off*, saving power, or *loaned out* to some other application, or simply *released* to a cloud computing platform, thus saving money. Fortunately, the bulk of the servers in a multi-tier data center are application servers, which are *stateless*, and are thus easy to turn off or give away – for example, one reported ratio of application servers to data servers is 5:1 [12]. We therefore focus our attention on dynamic capacity management of these front-end application servers.

Part of what makes dynamic capacity management difficult is the *setup cost* of getting servers back on/ready. For example, in our lab the setup time for turning on an application server is 260 seconds, during which time power is consumed at the peak rate of 200W. Sadly, little has been done to reduce the setup overhead for servers. In particular, sleep states, which are prevalent in mobile devices, have been very slow to enter the server market. Even if future hardware reduces the setup time, there may still be software imposed setup times due to software updates which occurred when the server was unavailable [12]. Likewise, the setup cost needed to create virtual machines (VMs) can range anywhere from 30s – 1 minute if the VMs are locally created (based on our measurements using kvm [21]) or 10 – 15 minutes if the VMs are obtained from a cloud computing platform (see, for example, [1]). All these numbers are extremely high, when compared with the typical SLA of half a second.

The goal of dynamic capacity management is to scale capacity with unpredictably changing load in the face of high setup costs. While there has been much prior work on this problem, all of it has only focussed on one aspect of changes in load, namely, fluctuations in request rate. This is already a difficult problem, given high setup costs, and has resulted in many policies, including reactive approaches [24, 29, 13, 39, 40, 11] that aim to react to the current request rate, predictive approaches [23, 33, 6, 17] that aim to predict the future request rate, and mixed reactive-predictive approaches [8, 9, 4, 37, 15, 36, 14]. However, in reality there are many other ways in which load can change. For example, *request size* (work associated with each request) can change, if new features or security checks are added to the application. As a second example, *server efficiency* can change, if any abnormalities occur in the system, such as internal service disruptions, slow networks, or maintenance cycles. These other types of load fluctuations are all too common in data centers, and have not been addressed by prior work in dynamic capacity management.

We propose a *new approach to dynamic capacity management*, which we call *AutoScale*. To describe *AutoScale*, we decompose it into two parts: *AutoScale*−− (see Section 3.5), which is a precursor to *AutoScale* and handles only the narrower case of unpredictable changes in request rate, and the full *AutoScale* policy (see Section 4.3), which builds upon *AutoScale*−− to handle all forms of changes in load.

While *AutoScale*−− addresses a problem that many others have looked at, it does so in a very different way. While prior approaches aim at predicting the future request rate and scaling *up* the number of servers to meet this predicted rate, which is clearly difficult to do when request rate is, by definition, unpredictable, *AutoScale*−− does not attempt to predict future request rate. Instead, *AutoScale*−− demonstrates that it is possible to achieve SLAs for real-world workloads by simply being conservative in scaling *down* the number of servers: not turning servers off recklessly. One might think that this same effect could be achieved by leaving a fixed buffer of, say, 20% extra servers on at all times. However, the extra capacity (20% in the above example) should *change* depending on the current load. *AutoScale*−− does just this – it maintains just the right number of servers in the on state at every point in time. This results in much lower power/resource consumption. In Section 3.5, we evaluate *AutoScale*−− on a suite of six different real-world traces, comparing it against five different capacity management policies commonly used in the literature. We demonstrate that in all cases, *AutoScale*−− significantly outperforms other policies, meeting response time SLAs while greatly reducing the number of servers needed, as shown in Table 3.

To fully investigate the applicability of *AutoScale*−−, we experiment with multiple setup times ranging from 260 seconds all the way down to 20 seconds in Section 3.7. Our results indicate that *AutoScale*−− can provides significant benefits across the entire spectrum of setup times, as shown in Figure 9.

To handle a broader spectrum of possible changes in load, including unpredictable changes in the request size and server efficiency, we introduce the *AutoScale* policy in Section 4.3. While prior approaches react only to changes in the request rate, *AutoScale* uses a novel capacity inference algorithm, which allows it to determine the appropriate capacity regardless of the source of the change in load. Importantly, *AutoScale* achieves this *without* requiring any knowledge of the request rate or the request size or the server efficiency, as shown in Tables 4, 5 and 6.

To evaluate the effectiveness of *AutoScale*, we build a three-tier testbed consisting of 38 servers that uses a key-value based workload, involving multiple interleavings of CPU and I/O within each request. While our implementation involves physically turning servers on and off, one could instead imagine that any idle server that is turned off is instead "given away", and there is a setup time to get the server back. To understand the benefits of *AutoScale*, we evaluate all policies on three metrics: $\mathbf{T_{95}}$, the 95th percentile of response time, which represents our SLA; $\mathbf{P_{avg}}$, the average power usage; and $\mathbf{N_{avg}}$, the average capacity, or number of servers in use (including those idle and in setup). Our goal is to meet the response time SLA, while keeping $\mathbf{P_{avg}}$ and $\mathbf{N_{avg}}$ as low as possible. The drop in $\mathbf{P_{avg}}$ shows the possible savings in power by turning off servers, while the drop in $\mathbf{N_{avg}}$ represents the potential capacity/servers available to be given away to other applications or to be released back to the cloud so as to save on rental costs.

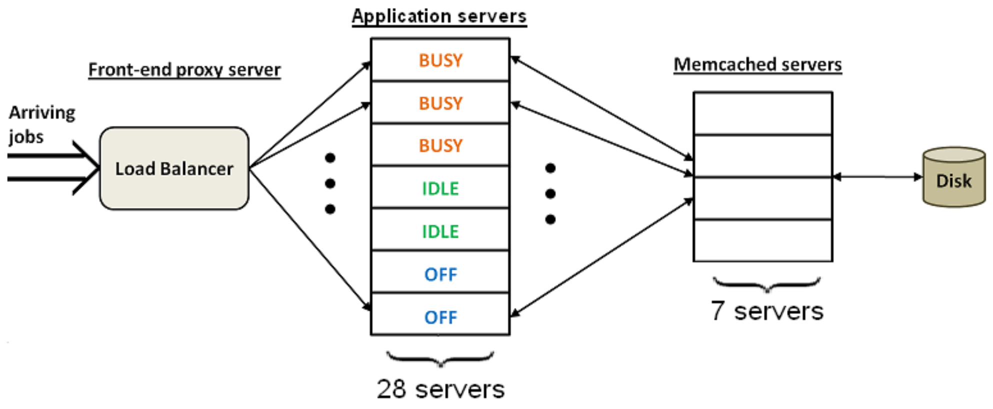


Figure 1: Our experimental testbed.

This paper makes the following contributions:

- We overturn the common wisdom that says that capacity provisioning requires "knowing the future load and planning for it," which is at the heart of existing predictive capacity management policies. Such predictions are simply not possible when workloads are unpredictable, and, we furthermore show they are unnecessary, at least for the range of variability in our workloads. We demonstrate that simply provisioning carefully and not turning servers off recklessly achieves better performance than existing policies that are based on predicting current load or over-provisioning to account for possible future load.
- We introduce our capacity inference algorithm which allows us to determine the appropriate capacity at any point of time in response to changes in request rate, request size and/or server efficiency, without any knowledge of these quantities (see Section 4.3). We demonstrate that *AutoScale*, via the capacity inference algorithm, is robust to all forms of changes in load, including unpredictable changes in request size and unpredictable degradations in server speeds, within the range of our traces. In fact, for our traces, *AutoScale* is robust to even a 4-fold increase in request size. To the best of our knowledge, *AutoScale* is the first policy to exhibit these forms of robustness. As shown in Tables 4, 5 and 6, other policies are simply not comparable on this front.

2 Experimental setup

2.1 Our experimental testbed

Figure 1 illustrates our data center testbed, consisting of 38 Intel Xeon servers, each equipped with two quad-core 2.26 GHz processors. We employ one of these servers as the front-end load generator running httperf [28] and another server as the front-end load balancer running Apache, which distributes requests from the load generator to the application servers. We modify Apache on the load balancer to also act as the capacity manager, which is responsible for turning servers on and off. Another server is used to store the entire data set, a billion key-value pairs, on a database.

Seven servers are used as memcached servers, each with 4GB of memory for caching. The remaining 28 servers are employed as application servers, which parse the incoming php requests and collect the required data from the back-end memcached servers. Our ratio of application servers to memcached servers is consistent with the typical ratio of 5:1 [12].

We employ capacity management on the application servers only, as they maintain no volatile state. We use the SNMP communication protocol to remotely turn application servers on and off via the power distribution unit (PDU). We monitor the power consumption of individual servers by reading the power values off of the PDU. The idle power consumption for our servers is about 140W (with C-states enabled) and the average power consumption for our servers when they are busy or in setup is about 200W.

In our experiments, we observed the setup time for the servers to be about 260 seconds. However, we also examine the effects of lower setup times that could either be a result of using sleep states (which are prevalent in laptops and desktop machines, but are not well supported for server architectures yet), or using virtualization to quickly bring up virtual machines. We replicate this effect by not routing requests to a server if it is marked for sleep, and by replacing its power consumption values with 0W. When the server is marked for setup, we wait for the setup time before sending requests to the server, and replace its power consumption values during the setup time with 200W.

2.2 Workload

We design a key-value workload to model realistic multi-tier applications such as the social networking site, Facebook, or e-commerce companies like Amazon [10]. Each generated request (or job) is a php script that runs on the application server. A request begins when the application server requests a value for a key from the memcached servers. The memcached servers provide the value, which itself is a collection of new keys. The application server then again requests values for these new keys from the memcached servers. This process can continue iteratively. In our experiments, we set the number of iterations to correspond to an average of roughly 3,000 key requests per job, which translates to a mean request size of approximately 120 ms, assuming no resource contention. The request size distribution is highly variable, with the largest request being roughly 20 times the size of the smallest request.

We can also vary the distribution of key requests by the application server. In this paper we use the Zipf [30] distribution, whereby the probability of generating a particular key varies inversely as a power of that key. To minimize the effects of cache misses in the memcached layer (which could result in an unpredictable fraction of the requests violating the $\mathbf{T_{95}}$ SLA), we tune the parameters of the Zipf distribution so that only a negligible fraction of requests miss in the memcached layer.

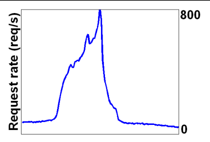
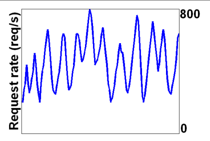
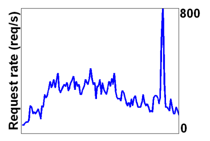
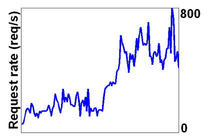
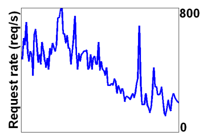
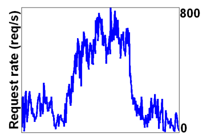
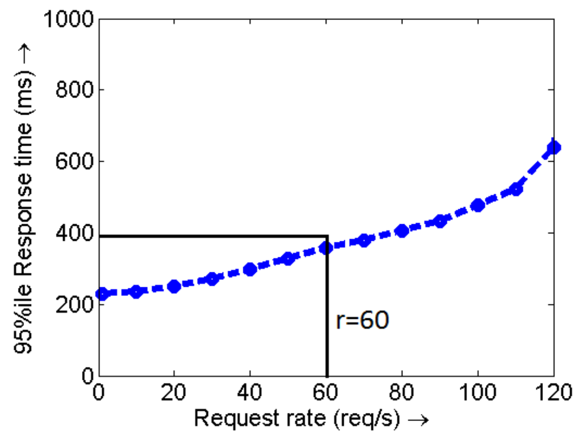
Name	Trace	Plot
Slowly varying	ITA [18]	
Quickly varying	Synthetic	
Big spike	NLANR [31]	
Dual phase	NLANR [31]	
Large variations	NLANR [31]	
Steep tri phase	SAP [34]	

Table 1: Description of the traces we use for experiments.

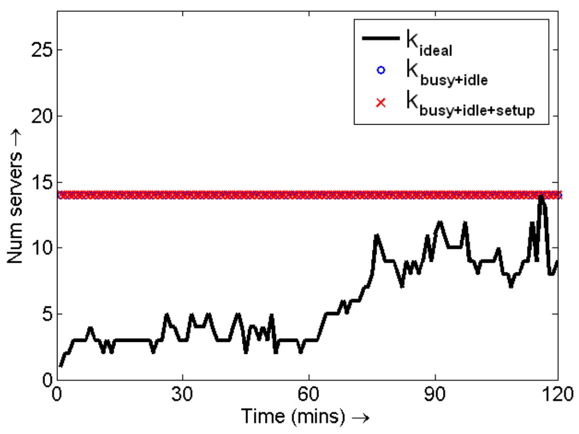
2.3 Trace-based arrivals

We use a variety of arrival traces to generate the request rate of jobs in our experiments, most of which are drawn from real-world traces. Table 1 describes these traces. In our experiments, the seven memcached servers can together handle at most 800 job requests per second, which corresponds to roughly 300,000 key requests per second at each memcached server. Thus, we scale the arrival traces such that the maximum request rate into the system is 800 req/s. Further, we scale the duration of the traces to 2 hours. We evaluate our policies against the full set of traces (see Table 3 for results).



95%ile response time vs. request rate

Figure 2: A single server can handle 60 req/s.



$\mathbf{T_{95}}$=291ms, $\mathbf{P_{avg}}$=2,323W, $\mathbf{N_{avg}}$=14

Figure 3: *AlwaysOn*.

3 Results: Changing request rates

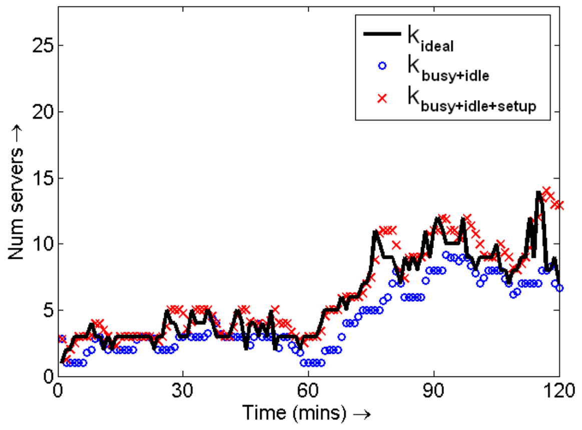
This section and the next both involve implementation and performance evaluation of a range of capacity management policies. Each policy will be evaluated against the six traces described in Table 1. We will present detailed results for the Dual phase trace and show summary results for all traces in Table 3. The Dual phase trace is chosen because it is quite bursty and also represents the diurnal nature of typical data center traffic, whereby the request rate is low for a part of the day (usually the night time) and is high for the rest (day time). The goal throughout will be to meet 95%ile guarantees of $\mathbf{T_{95}} = 400 - 500$ ms[1], while minimizing the average power consumed by the application servers, $\mathbf{P_{avg}}$, or the average number of application servers used, $\mathbf{N_{avg}}$. Note that $\mathbf{P_{avg}}$ largely scales with $\mathbf{N_{avg}}$.

For capacity management, we want to choose the number of servers at time t, $k(t)$, such that we meet a 95%ile response time goal of $400 - 500$ ms. Figure 2 shows measured 95%ile response time at a *single server* versus request rate. According to this figure, for example, to meet a 95%ile goal of 400 ms, we require the request rate to a single server to be no more than $r = 60$ req/s. Hence, if the total request rate into the data center at some time t is say, $R(t) = 300$ req/s, we know that we need at least $k = \lceil 300/r \rceil = 5$ servers to ensure our 95%ile SLA.

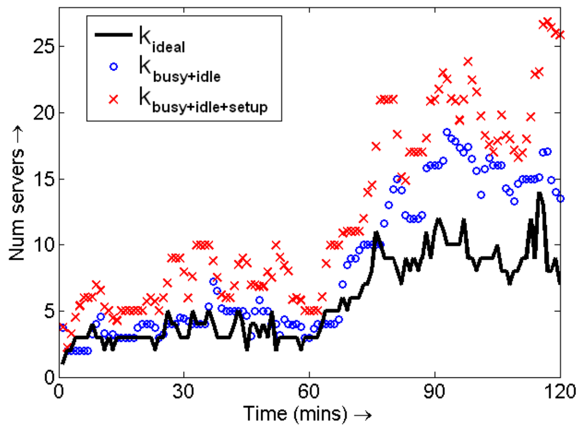
3.1 AlwaysOn

The *AlwaysOn* policy [38, 8, 17] is important because this is what is currently deployed by most of the industry. The policy selects a fixed number of servers, k, to handle the peak request rate

[1]It would be equally easy to use 90%ile guarantees or 99%ile guarantees. Likewise, we could easily have aimed for 300ms or 1 second response times rather than 500ms. Our choice of SLA is motivated by recent studies [36, 23, 27, 10] which indicate that 95%ile guarantees of hundreds of milliseconds are typical.



(a) $\mathbf{T_{95}}$=11,003ms, $\mathbf{P_{avg}}$=1,281W, $\mathbf{N_{avg}}$=6.2



(b) $\mathbf{T_{95}}$=487ms, $\mathbf{P_{avg}}$=2,218W, $\mathbf{N_{avg}}$=12.1

Figure 4: (a) *Reactive* and (b) *Reactive* with extra capacity.

and always leaves those servers on. In our case, to meet the 95%ile SLA of 400ms, we set $k = \lceil R_{peak}/60 \rceil$, where $R_{peak} = 800$ req/s denotes the peak request rate into the system. Thus, k is fixed at $\lceil 800/60 \rceil = 14$.

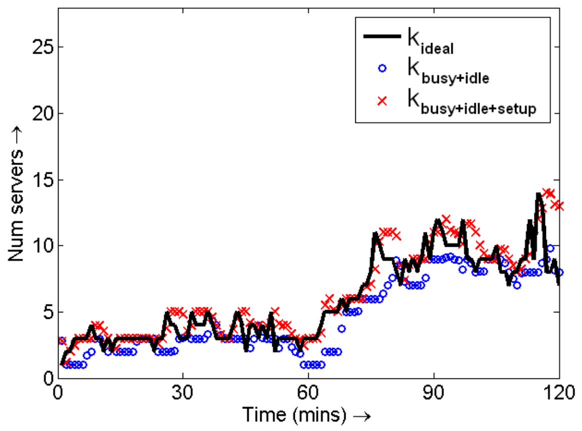
Realistically, one doesn't know R_{peak}, and it is common to overestimate R_{peak} by a factor of 2 (see, for example, [23]). In this paper, we empower *AlwaysOn*, by assuming that R_{peak} is known in advance.

Figure 3 shows the performance of *AlwaysOn*. The solid line shows k_{ideal}, the *ideal* number of servers/capacity which should be on at any given time, as given by $k(t) = \lceil R(t)/60 \rceil$. Circles are used to show $k_{busy+idle}$, the number of servers which are actually on, and crosses show $k_{busy+idle+setup}$, the actual number of servers that are on or in setup. For *AlwaysOn*, the circles and crosses lie on top of each other since servers are never in setup. Observe that $\mathbf{N_{avg}} = \lceil \frac{800}{60} \rceil = 14$ for *AlwaysOn*, while $\mathbf{P_{avg}} = 2323W$, with similar values for the different traces in Table 3.

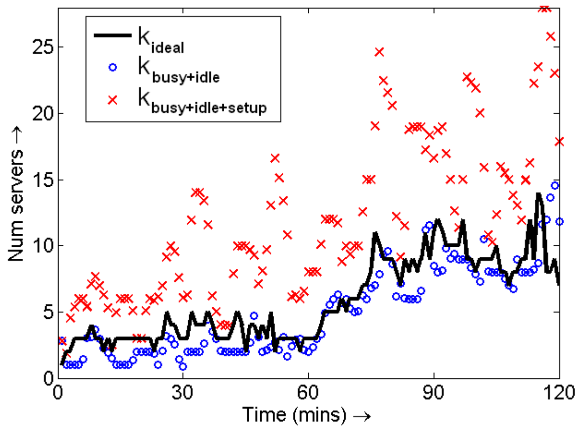
3.2 Reactive

The *Reactive* policy (see, for example, [36]) reacts to the current request rate, attempting to keep exactly $\lceil R(t)/60 \rceil$ servers on at time t, in accordance with the solid line. However, because of the setup time of 260s, *Reactive* lags in turning servers on. In our implementation of *Reactive*, we sample the request rate every 20 seconds, adjusting the number of servers as needed.

Figure 4(a) shows the performance of *Reactive*. By reacting to current request rate and adjusting the capacity accordingly, *Reactive* is able to bring down $\mathbf{P_{avg}}$ and $\mathbf{N_{avg}}$ by as much as a factor of two or more, when compared with *AlwaysOn*. This is a huge win. Unfortunately, the response time SLA is almost never met and is typically exceeded by a factor of at least 10-20 (as in Figure 4(a)), or even by a factor of 100 (see Table 3).



(a) *MWA*: $\mathbf{T_{95}}$=7,740ms, $\mathbf{P_{avg}}$=1,276W, $\mathbf{N_{avg}}$=6.3



(b) *LR*: $\mathbf{T_{95}}$=2,544ms, $\mathbf{P_{avg}}$=2,161W, $\mathbf{N_{avg}}$=11.8

Figure 5: (a) *Predictive: MWA* and (b) *Predictive: LR*.

3.3 Reactive with extra capacity

One might think the response times under *Reactive* would improve a lot by just adding some $x\%$ extra capacity at all times. This $x\%$ extra capacity can be achieved by running *Reactive* with a different r setting. Unfortunately, for this trace, it turns out that to bring $\mathbf{T_{95}}$ down to our desired SLA, we need 100% extra capacity at all times, which corresponds to setting $r = 30$. This brings $\mathbf{T_{95}}$ down to 487 ms, but causes power to jump up to the levels of *AlwaysOn*, as illustrated in Figure 4(b). It is even more problematic that each of our six traces in Table 1 requires a different $x\%$ extra capacity to achieve the desired SLA (with $x\%$ typically ranging from 50% to 200%), rendering such a policy impractical.

3.4 Predictive

Predictive policies attempt to predict the request rate 260 seconds from now. This section describes two policies that were used in many papers [5, 16, 32, 38] and were found to be the most powerful by [23].

Predictive - Moving Window Average (MWA)
In the *MWA* policy, we consider a "window" of some duration (say, 10 seconds). We average the request rates during that window to deduce the predicted rate during the 11th second. Then we slide the window to include seconds 2 through 11, and average those values to deduce the predicted rate during the 12th second. We continue this process of sliding the window rightward until we have predicted the request rate at time 270 seconds, based on the initial 10 seconds window.

If the estimated request rate at second 270 exceeds the current request rate, we determine the number of additional servers needed to meet the SLA (via the $k = \lceil R/r \rceil$ formula) and turn these on at time 11, so that they will be ready to run at time 270. If the estimated request rate at second 270 is lower than the current request rate, we look at the maximum request rate, M, during the interval from time 11 to time 270. If M is lower than the current request rate, then we turn off as many servers as we can while meeting the SLA for request rate M. Of course, the window size affects the performance of *MWA*. *We empower* MWA *by using the best window size for each trace.*

Figure 5(a) shows that the performance of *Predictive MWA* is very similar to what we saw for *Reactive*: low $\mathbf{P_{avg}}$ and $\mathbf{N_{avg}}$ values, beating *AlwaysOn* by a factor of 2, but high $\mathbf{T_{95}}$ values, typically exceeding the SLA by a factor of 10 to 20.

Predictive - Linear Regression (LR)
The *LR* policy is identical to *MWA* except that, to estimate the request rate at time 270 seconds, we use linear regression to match the best linear fit to the values in the window. Then we extend our line out by 260 seconds to get a prediction of the request rate at time 270 seconds.

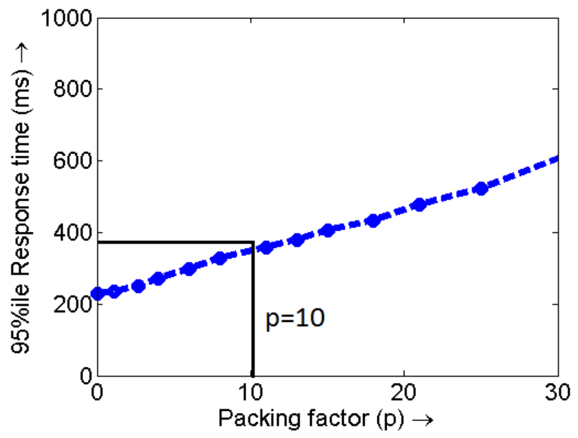
The performance of *Predictive LR* is worse than that of *Predictive MWA*. Response times are still bad, but now capacity and power consumption can be bad as well. The problem, as illustrated in Figure 5(b), is that the linear slope fit used in *LR* can end up overshooting the required capacity greatly.

3.5 AutoScale$--$

One might think that the poor performance of the dynamic capacity management policies we have seen so far stems from the fact that they are too slow to turn servers on when needed. However, an equally big concern is the fact that these policies are quick to turn servers *off* when not needed, and hence do not have those servers available when load subsequently rises. This rashness is particularly problematic in the case of bursty workloads, such as those in Table 1.

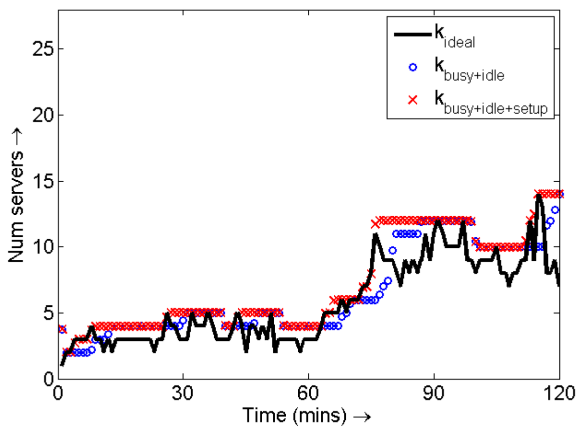
AutoScale$--$ addresses the problem of **scaling down** capacity by being very conservative in turning servers off while doing nothing new with respect to turning servers on (the turning on algorithm is the same as in *Reactive*). We will show that *by simply taking more care in turning servers off*, *AutoScale*$--$ is able to outperform all the prior dynamic capacity management policies we have seen with respect to meetings SLAs, while simultaneously keeping $\mathbf{P_{avg}}$ and $\mathbf{N_{avg}}$ low.

When to turn a server off?
Under *AutoScale*$--$, each server decides *autonomously* when to turn off. When a server goes idle, rather than turning off immediately, it sets a timer of duration t_{wait} and sits in the idle state for t_{wait} seconds. If a request arrives at the server during these t_{wait} seconds, then the server goes back to the busy state (with zero setup cost); otherwise the server is turned off. In our experiments for *AutoScale*$--$, we use a t_{wait} value of 120s. Table 2 shows that *AutoScale*$--$ is largely insensitive



95%ile response time vs. packing factor

Figure 6: For a single server, $p = 10$.



$\mathbf{T_{95}}$=491ms, $\mathbf{P_{avg}}$=1,297W, $\mathbf{N_{avg}}$=7.2

Figure 7: *AutoScale−−*.

to t_{wait} in the range $t_{wait} = 60s$ to $t_{wait} = 260s$. There is a slight increase in $\mathbf{P_{avg}}$ (and $\mathbf{N_{avg}}$) and a slight decrease in $\mathbf{T_{95}}$ when t_{wait} increases, due to idle servers staying on longer.

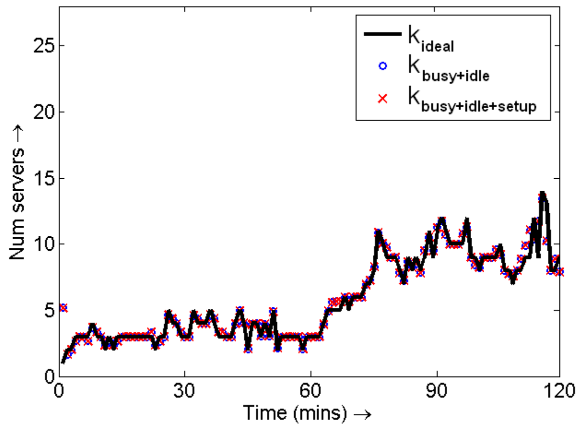
The idea of setting a timer before turning off an idle server has been proposed before (see, for example, [20, 26, 19]), however, *only for a single server*. For a multi-server system, *independently* setting timers for each server can be inefficient, since we can end up with too many idle servers. Thus, we need a more coordinated approach for using timers in our multi-server system which takes routing into account, as explained below.

How to route jobs to servers?
Timers prevent the mistake of turning off a server just as a new arrival comes in. However, they can also waste power and capacity by leaving too many servers in the idle state. We'd basically like to keep only a small number of servers (just the *right* number) in this idle state.

To do this, we introduce a routing scheme that tends to concentrate jobs onto a small number of servers, so that the remaining (unneeded) servers will naturally "time out." Our routing scheme uses an *index-packing* idea, whereby all on servers are indexed from 1 to n. Then we send each request to the lowest-numbered on server that currently has fewer than p requests, where p stands for *packing factor* and denotes the maximum number of requests that a server can serve concurrently and meet its response time SLA. For example, in Figure 6, we see that to meet a 95%ile

Trace \ t_{wait}		60s	120s	260s
Dual phase [31]	$\mathbf{T_{95}}$	503ms	491ms	445ms
	$\mathbf{P_{avg}}$	1,253W	1,297W	1,490W
	$\mathbf{N_{avg}}$	7.0	7.2	8.8

Table 2: The (in)sensitivity of *AutoScale−−*'s performance to t_{wait}.



$\mathbf{T_{95}}$=320ms, $\mathbf{P_{avg}}$=1,132W, $\mathbf{N_{avg}}$=5.9

Figure 8: *Opt*.

guarantee of 400 ms, the packing factor is $p = 10$ (in general, the value of p depends on the system in consideration). When all on servers are already packed with p requests each, additional request arrivals are routed to servers via the join-the-shortest-queue routing.

In comparison with all the other policies, *AutoScale*$--$ hits the "sweet spot" of low $\mathbf{T_{95}}$ as well as low $\mathbf{P_{avg}}$ and $\mathbf{N_{avg}}$. As seen from Table 3, *AutoScale*$--$ is close to the response time SLA in all traces except for the Big spike trace. Simultaneously, the mean power usage and capacity under *AutoScale*$--$ is typically significantly better than *AlwaysOn*, saving as much as a factor of two in power and capacity.

Figure 7 illustrates how *AutoScale*$--$ is able to achieve these performance results. Observe that the crosses and circles in *AutoScale*$--$ form flat constant lines, instead of bouncing up and down, erratically, as in the earlier policies. This comes from a combination of the t_{wait} timer and the index-based routing, which together keep the number of servers just slightly above what is needed, while also avoiding toggling the servers between on and off states when the load goes up and down. Comparing Figures 7 and 4(b), we see that the combination of timers and index-based routing is far more effective than using *Reactive* with extra capacity, as in Section 3.3.

3.6 Opt

As a yardstick for measuring the effectiveness of *AutoScale*$--$, we define an optimal policy, *Opt*, which behaves identically to *Reactive*, but with a setup time of zero. Thus, as soon as the request rate changes, *Opt* reacts by immediately adding or removing the required capacity, without having to wait for setup. Figure 8 shows that under *Opt*, the number of servers on scales exactly with the incoming request load. *Opt* easily meets the $\mathbf{T_{95}}$ SLA, and consumes very little power and resources (servers). Note that while *Opt* usually has a $\mathbf{T_{95}}$ of about 320-350ms, and thus it might seem like *Opt* is over-provisioning, it just about meets the $\mathbf{T_{95}}$ SLA for the Tri phase trace (see

Trace \ Policy		*AlwaysOn*	*Reactive*	*Predictive MWA*	*Predictive LR*	*Opt*	*AutoScale−−*
Slowly varying [18]	$\mathbf{T_{95}}$	271ms	673ms	3,464ms	618ms	366ms	435ms
	$\mathbf{P_{avg}}$	2,205W	842W	825W	964W	788W	1,393W
	$\mathbf{N_{avg}}$	14.0	4.1	4.1	4.9	4.0	5.8
Quickly varying	$\mathbf{T_{95}}$	303ms	20,005ms	3,335ms	12,553ms	325ms	362ms
	$\mathbf{P_{avg}}$	2,476W	1,922W	2,065W	3,622W	1,531W	2,205W
	$\mathbf{N_{avg}}$	14.0	10.1	10.6	22.1	8.2	15.1
Big spike [31]	$\mathbf{T_{95}}$	229ms	3,426ms	9,337ms	1,753ms	352ms	854ms
	$\mathbf{P_{avg}}$	2,260W	985W	998W	1,503W	845W	1,129W
	$\mathbf{N_{avg}}$	14.0	4.9	4.9	8.1	4.5	6.6
Dual phase [31]	$\mathbf{T_{95}}$	291ms	11,003ms	7,740ms	2,544ms	320ms	491ms
	$\mathbf{P_{avg}}$	2,323W	1,281W	1,276W	2,161W	1,132W	1,297W
	$\mathbf{N_{avg}}$	14.0	6.2	6.3	11.8	5.9	7.2
Large variations [31]	$\mathbf{T_{95}}$	289ms	4,227ms	13,399ms	20,631ms	321ms	474ms
	$\mathbf{P_{avg}}$	2,363W	1,391W	1,461W	2,576W	1,222W	1,642W
	$\mathbf{N_{avg}}$	14.0	7.8	8.1	16.4	7.1	10.5
Steep tri phase [34]	$\mathbf{T_{95}}$	377ms	>1 min	>1 min	661ms	446ms	463ms
	$\mathbf{P_{avg}}$	2,263W	849W	1,287W	3,374W	1,004W	1,601W
	$\mathbf{N_{avg}}$	14.0	5.2	7.2	20.5	5.1	8.0

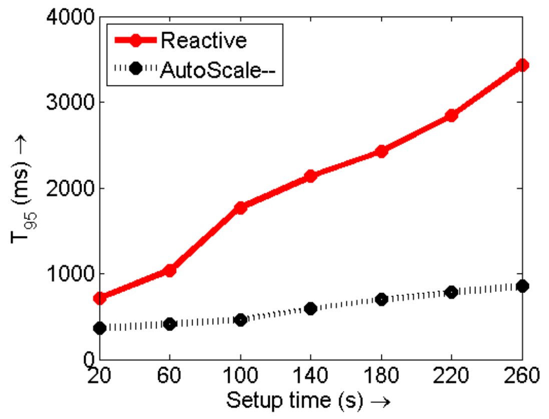
Table 3: Comparison of all policies. Setup time = 260s throughout.

Table 3) and hence cannot be made more aggressive.

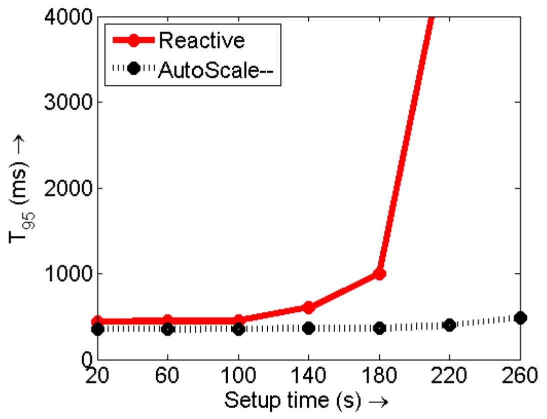
In support of *AutoScale−−*, we find that *Opt*'s power consumption and server usage is only 30% less than that of *AutoScale−−*, averaged across all traces, despite *AutoScale−−* having to cope with the 260s setup time.

3.7 Lower setup times

While production servers today are only equipped with "off" states that necessitate huge setup times (260s for our servers), future servers may support sleep states, which can lower setup times considerably. Further, with virtualization, the setup time required to bring up additional capacity (in the form of virtual machines) might also go down. In this section, we again contrast the performance of *AutoScale−−* with simpler dynamic capacity management policies, for the case of lower setup times. We achieve these lower setup times by tweaking our experimental testbed as discussed at the end of Section 2.1. Furthermore, for *AutoScale−−*, we reduce the value of t_{wait} in proportion to the reduction in setup time.



(a) Big spike



(b) Dual phase

Figure 9: Effect of lower setup times for (a) Big spike trace [31] and (b) Dual phase trace [31].

When the setup time is very low, approaching zero, then by definition, all policies approach *Opt*. For moderate setup times, one might expect that *AutoScale−−* does not provide significant benefits over other policies such as *Reactive*, since $\mathbf{T_{95}}$ should not rise too much during the setup time. This turns out to be false since the $\mathbf{T_{95}}$ under *Reactive* continues to be high even for moderate setup times.

Figure 9(a) shows our experimental results for $\mathbf{T_{95}}$ for the Big spike trace [31], under *Reactive* and *AutoScale−−*. We see that as the setup time drops, the $\mathbf{T_{95}}$ drops almost linearly for both *Reactive* and *AutoScale−−*. However, *AutoScale−−* continues to be superior to *Reactive* with respect to $\mathbf{T_{95}}$ for any given setup time. In fact, even when the setup time is only 20s, the $\mathbf{T_{95}}$ under *Reactive* is almost twice that under *AutoScale−−*. This is because of the huge spike in load in the Big spike trace that cannot be handled by *Reactive* even at low setup times. We find similar results for the Steep tri phase trace [34], with $\mathbf{T_{95}}$ under *Reactive* being more than three times as high as that under *AutoScale−−*. The $\mathbf{P_{avg}}$ and $\mathbf{N_{avg}}$ values for *Reactive* and *AutoScale−−* also drop with setup time, but the changes are not as significant as for $\mathbf{T_{95}}$.

Figure 9(b) shows our experimental results for $\mathbf{T_{95}}$ for the Dual phase trace [31], under *Reactive* and *AutoScale−−*. This time, we see that as the setup time drops below 100s, the $\mathbf{T_{95}}$ under *Reactive* approaches that under *AutoScale−−*. This is because of the relatively small fluctuations in load in the Dual phase trace, which can be handled by *Reactive* once the setup time is small enough. However, for setup times larger than 100s, *AutoScale−−* continues to be significantly better than *Reactive*. We find similar results for the Quickly varying trace and the Large variations trace [31].

In summary, depending on the trace, *Reactive* can perform poorly even for low setup times (see Figure 9(a)). We expect similar behavior under the *Predictive* policies as well. Thus, *AutoScale−−* can be very beneficial even for more moderate setup times. Note that *AlwaysOn* and *Opt* are not affected by setup times.

4 Results: Robustness

Thus far in our traces we have only varied the request rate over time. However, in reality there are many other ways in which load can change. For example, if new features or security checks are added to the application, the request size might increase. We mimic such effects by increasing the number of key-value lookups associated with each request. As a second example, if any abnormalities occur in the system, such as internal service disruptions, slow networks, or maintenance cycles, servers may respond more slowly, and requests may accumulate at the servers. We mimic such effects by slowing down the frequency of the application servers. All the dynamic capacity management policies described thus far, with the exception of *Opt*, use the request rate to scale capacity. However, using the request rate to determine the required capacity is somewhat *fragile*. If the request *size* increases, or if servers become *slower*, due to any of the reasons mentioned above, then the number of servers needed to maintain acceptable response times ought to be increased. In both cases, however, no additional capacity will be provisioned if the policies only look at request rate to scale up capacity.

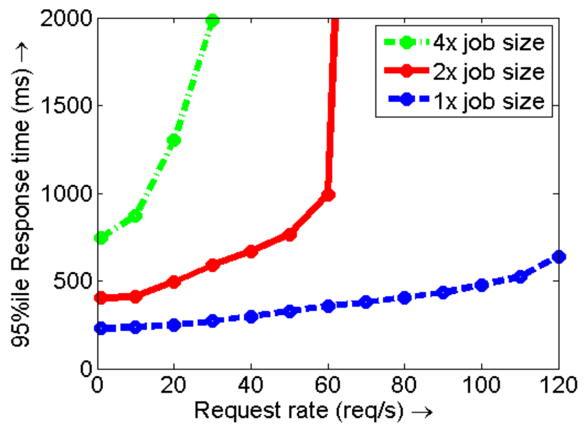
4.1 Why request rate is not a good control knob

In order to assess the limitations of using request rate as a control knob for scaling capacity, we ran *AutoScale*$--$ on the Dual phase trace with a 2x request size (meaning that our request size is now 240ms as opposed to the 120ms size we have used thus far). Since *AutoScale*$--$ does not detect an increase in request size, and thus doesn't provision for this, its $\mathbf{T_{95}}$ shoots up ($\mathbf{T_{95}} = 51,601ms$). This is also true for the *Reactive* and *Predictive* policies, as can be seen in Tables 4 and 5 for the case of increased request size and in Table 6 for the case of slower servers.

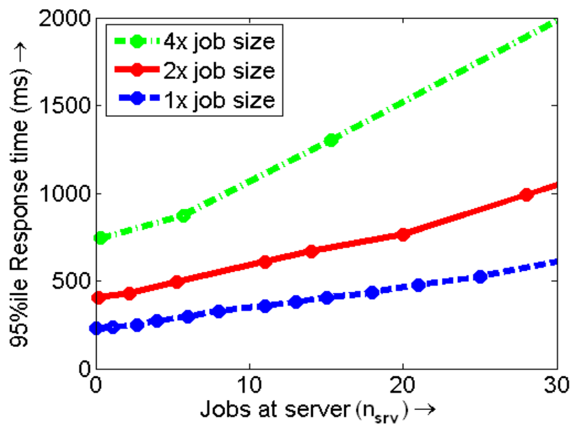
Figure 10 shows measured 95%ile response time at a single server versus request rate for different request sizes. It is clear that while each server can handle 60 req/s without violating the $\mathbf{T_{95}}$ SLA for a 1x request size, the $\mathbf{T_{95}}$ shoots up for the 2x and 4x request sizes. An obvious way to solve this problem is to determine the request size. However, it is not easy to determine the request size since the size is usually not known ahead of time. Trying to derive the request size by monitoring the response times doesn't help either since response times are usually affected by queueing delays. Thus, we need to come up with a better control knob than request rate or request size.

4.2 A better control knob that's still not quite right

We propose using the number of requests in the system, n_{sys}, as the control knob for scaling up capacity rather than the request rate. We assert that n_{sys} more faithfully captures the dynamic state of the system than the request rate. If the system is under-provisioned *either* because the request rate is too high *or* because the request size is too big *or* because the servers have slowed down, n_{sys} will tend to increase. If the system is over-provisioned, n_{sys} will tend to decrease below some



95%ile response time vs. request rate



95%ile response time vs. n_{srv}

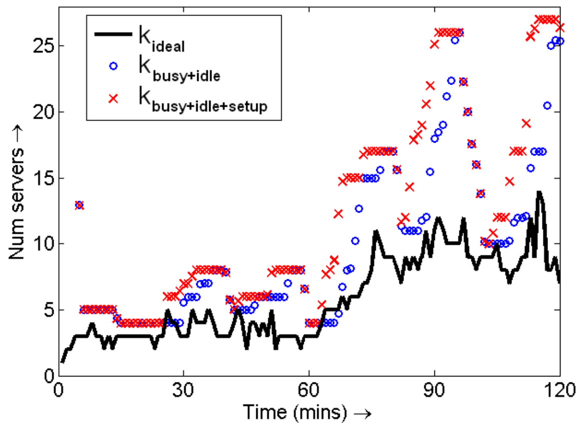
Figure 10: A single server can no longer handle 60 req/s when the request size increases.

Figure 11: For a single server, setting $n_{srv} = p = 10$ works well for all request sizes.

expected level. Further, calculating n_{sys} is fairly straightforward; many modern systems (including our Apache load balancer) already track this value, and it is instantaneously available.

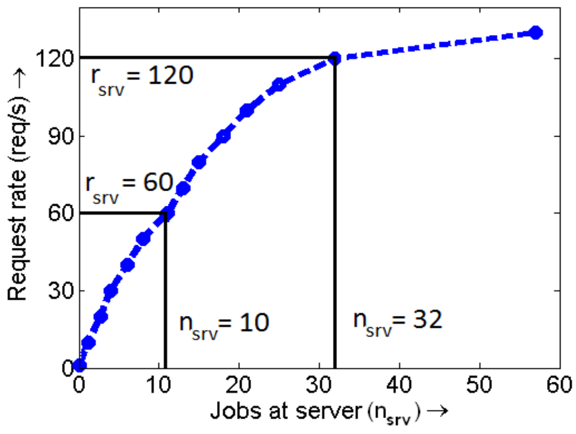
Figure 11 shows the measured 95%ile response time at a single server versus the number of jobs at a single server, n_{srv}, for different request sizes. Note that $n_{srv} = n_{sys}$ in the case of a single-server system. Surprisingly, the 95%ile response time values do not shoot up for the 2x and 4x request sizes for a given n_{srv} value. In fact, setting $n_{srv} = 10$, as in Section 3.5, provides acceptable $\mathbf{T_{95}}$ values for all request sizes (note that $\mathbf{T_{95}}$ values for the 2x and 4x request sizes are higher than 500ms, which is to be expected as the work associated with each request is naturally higher). This is because an increase in the request size (or a decrease in the server speed) increases the rate at which "work" comes into each server. This increase in work is reflected in the consequent increase in n_{srv}. By limiting n_{srv} using p, the packing factor (the maximum number of requests that a server can serve concurrently and meet its SLA), we can limit the rate at which work comes in to each server, thereby adjusting the required capacity to ensure that we meet the $\mathbf{T_{95}}$ SLA. Based on these observations, we set $p = 10$ for the 2x and 4x request sizes. Thus, *p is agnostic to request sizes for our system*, and only needs to be computed once. The insensitivity of p to request sizes is to be expected since p represents the degree of parallelism for a server, and thus depends on the specifications of a server (number of cores, hyper-threading, etc), and not on the request size.

Based on our observations from Figure 11, we propose a plausible solution for dynamic capacity management based on looking at the total number of requests in the system, n_{sys}, as opposed to looking at the request rate. The idea is to provision capacity to ensure that $n_{srv} = 10$ at each server. In particular, the proposed policy is exactly the same as *AutoScale*−−, except that it estimates the required number of servers as $k_{reqd} = \lceil n_{sys}/10 \rceil$, where n_{sys} is the total number of requests in the system at that time. In our implementation, we sample n_{sys} every 20 seconds, and thus, the proposed policy re-scales capacity, if needed, every 20 seconds. Note that the proposed policy uses



$\mathbf{T_{95}}$=441ms, $\mathbf{P_{avg}}$=2,083W, $\mathbf{N_{avg}}$=12.5

Figure 12: Our proposed policy overshoots while scaling up capacity.



Request rate vs. number of jobs.

Figure 13: A doubling of request rate can lead to a tripling of number of jobs at a single server.

the same method to scale down capacity as *AutoScale*−−, viz., using a timeout of 120s along with the index-packing routing.

Figure 12 shows how our proposed policy behaves for the 1x request size. We see that our proposed policy successfully meets the $\mathbf{T_{95}}$ SLA, but it clearly overshoots in terms of scaling up capacity when the request rate goes up. Thus, the proposed policy results in high power and resource consumption. One might think that this overshoot can be avoided by increasing the value of p, thus allowing n_{srv} to be higher than 10. However, note that the $\mathbf{T_{95}}$ in Figure 12 is already quite close to the 500ms SLA, and increasing the value of p beyond 10 can result in SLA violations.

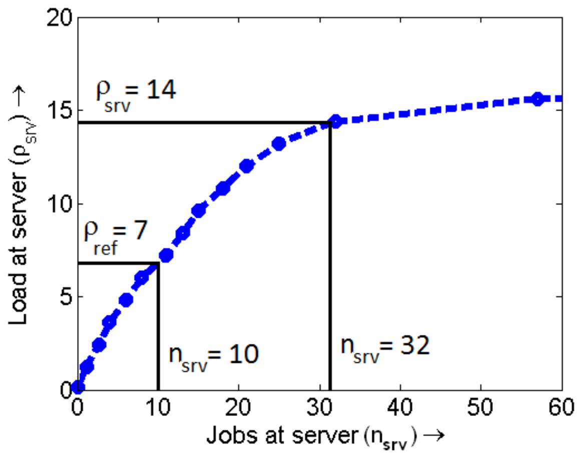
Figure 13 explains the overshoot in terms of scaling up capacity for our proposed policy. We see that when the request rate into a single server, r_{srv}, doubles from 60 req/s to 120 req/s, n_{srv} more than doubles from 10 to 32. Thus, our proposed policy scales up capacity by a factor of 3, whereas ideally capacity should only be scaled up by a factor of 2. Clearly our proposed policy does not work so well, even in the case where the request size is just 1x.

We now introduce our *AutoScale* policy, which solves our problems of scaling up capacity.

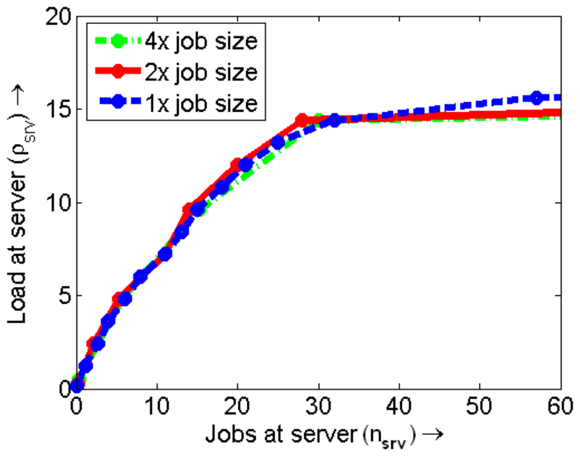
4.3 AutoScale: Incorporating the right control knob

We now describe the *AutoScale* policy and show that it not only handles the case where request rate changes, but also handles cases where the request size changes (see Tables 4 and 5) or where the server efficiency changes (see Table 6).

AutoScale differs from the existing capacity management policies in that it uses n_{sys} as the control knob rather than request rate. However, *AutoScale* does not simply scale up the capacity linearly with an increase in n_{sys}, as was the case with our proposed policy above. This is because



(a) ρ_{srv} vs. n_{srv} for the 1x request size.



(b) ρ_{srv} vs. n_{srv} for all request sizes.

Figure 14: Load at a server as a function of the number of jobs at a server for various request sizes. Surprisingly, the graph is invariant to changes in request size.

n_{sys} grows *super-linearly* during the time that the system is under-provisioned, as is well known in queueing theory. Instead, *AutoScale* tries to infer the *amount of work* in the system by monitoring n_{sys}. The amount of work in the system is proportional to both the request rate and the request size (the request size in turn depends also on the server efficiency), and thus, we try to infer the product of request rate and request size, which we call *system load*, ρ_{sys}. Formally,

$$\rho_{sys} = \begin{matrix}\text{request rate into} \\ \text{the data center } (R)\end{matrix} \times \begin{matrix}\text{average} \\ \text{request size,}\end{matrix}$$

where the average 1x request size is 120ms. Fortunately, there is an easy relationship (which we describe soon) to go from n_{sys} to ρ_{sys}, obviating the need to ever measure load or request rate or the request size. Once we have ρ_{sys}, it is easy to get to k_{reqd}, since ρ_{sys} represents the amount of work in the system and is hence proportional to k_{reqd}. We now explain the translation process from n_{sys} to ρ_{sys} and then from ρ_{sys} to k_{reqd}. We refer to this entire translation algorithm as the *capacity inference algorithm*. The full translation from n_{sys} to k_{reqd} will be given in Equation 3 below.

The capacity inference algorithm

In order to understand the relationship between n_{sys} and ρ_{sys}, we first derive the relationship between the number of jobs at a single server, n_{srv}, and the load at a single server, ρ_{srv}. Formally, the load at a server is defined as

$$\rho_{srv} = \begin{matrix}\text{request rate into} \\ \text{a single server } (r_{srv})\end{matrix} \times \begin{matrix}\text{average} \\ \text{request size,}\end{matrix} \qquad (1)$$

where the average 1x request size is 120ms and r_{srv} is the request rate into a single server. If the request rate, r_{srv}, is made as high as possible without violating the SLA, then the resulting ρ_{srv}

from Equation 1 is referred to as ρ_{ref}. For our system, recall that the maximum request rate into a single server without violating the SLA is $r_{srv} = 60$ req/s (see Figure 2). Thus,

$$\rho_{ref} = 60 \times 0.12 \approx 7, \tag{2}$$

meaning that a single server can handle a load of at most 7 without violating the SLA, assuming a 1x request size of 120ms.

Returning to the discussion of how ρ_{srv} and n_{srv} are related, we expect that ρ_{srv} should increase with n_{srv}. Figure 14(a) shows our experimental results for ρ_{srv} as a function of n_{srv}. Note that $\rho_{srv} = \rho_{ref}$ corresponds to $n_{srv} = p = 10$, where p is the packing factor. We obtain Figure 14(a) by converting r_{srv} in Figure 13 to ρ_{srv} using Equation 1 above. Observe that when ρ_{srv} doubles from 7 to 14, we see that n_{srv} more than triples from 10 to 32, as was the case in Figure 13.

We'll now estimate ρ_{sys}, the system load, using the relationship between n_{srv} and ρ_{srv}. To estimate ρ_{sys}, we first approximate n_{srv} as $\frac{n_{sys}}{k_{curr}}$, where k_{curr} is the current number of on servers. We then use n_{srv} in Figure 14(a) to estimate the corresponding ρ_{srv}. Finally, we have $\rho_{sys} = k_{curr} \cdot \rho_{srv}$.

Surprisingly, the relationship between n_{srv} and ρ_{srv} does not change when request size changes. Figure 14(b) shows our experimental results for the relationship between n_{srv} and ρ_{srv} for different request sizes. We see that the plot is invariant to changes in request size. Thus, while calculating $\rho_{sys} = k_{curr} \cdot \rho_{srv}$, we don't have to worry about the request size and we can simply use Figure 14(a) to estimate ρ_{sys} from n_{sys} *irrespective of the request size*. Likewise, we find that the relationship between n_{srv} and ρ_{srv} does not change when the server speed changes. This is because a decrease in server speed is the same as an increase in request size for our system.

The reason why the relationship between n_{srv} and ρ_{srv} is agnostic to request size is because ρ_{srv}, by definition (see Equation 1), takes the request size into account. If the request size doubles, then the request rate into a server needs to drop by a factor of 2 in order to maintain the same ρ_{srv}. These changes result in exactly the same *amount of work entering the system per unit time*, and thus, n_{srv} does not change. The insensitivity of the relationship between n_{srv} and ρ_{srv} to changes in request size is consistent with queueing-theoretic analysis [22]. Interestingly, this insensitivity, coupled with the fact that p is a constant for our system (see Section 4.2), results in ρ_{ref} *being a constant for our system*, since ρ_{ref} is the same as ρ_{srv} for the case when $n_{srv} = p = 10$ (see Figure 14(a)). Thus, we only need to compute ρ_{ref} once for our system.

Now that we have ρ_{sys}, we can translate this to k_{reqd} using ρ_{ref}. Since ρ_{sys} corresponds to the total system load, while ρ_{ref} corresponds to the load that a single server can handle, we deduce that the required capacity is:

$$k_{reqd} = \left\lceil \frac{\rho_{sys}}{\rho_{ref}} \right\rceil$$

In summary, we can get from n_{sys} to k_{reqd} by first translating n_{sys} to ρ_{sys}, which leads us to k_{reqd}, as outlined below:

$$n_{sys} \xrightarrow{\div k_{curr}} n_{srv} \xrightarrow{Fig.\ 14(a)} \rho_{srv} \xrightarrow{\times k_{curr}} \rho_{sys} \xrightarrow{\div \rho_{ref}} k_{reqd} \tag{3}$$

For example, if $n_{sys} = 320$ and $k_{curr} = 10$, then we get $n_{srv} = 32$, and from Figure 14(a), $\rho_{srv} = 14$, irrespective of request size. The load for the system, ρ_{sys}, is then given by $k_{curr} \cdot \rho_{srv} = 140$, and since $\rho_{ref} = 7$, the required capacity is $k_{reqd} = \lceil k \cdot \frac{\rho_{srv}}{\rho_{ref}} \rceil = 20$. Consequently, *AutoScale* turns on 10 additional servers. In our implementation, we reevaluate k_{reqd} every 20s to avoid excessive changes in the number of servers.

The insensitivity of the relationship between n_{srv} and ρ_{srv} allows us to use Equation 3 to compute the desired capacity, k_{reqd}, in response to any form of load change. Further, as noted above, p and ρ_{ref} are constants for our system, and only need to be computed once. These properties make *AutoScale* a very robust capacity management policy.

Performance of AutoScale
Tables 4 and 5 summarize results for the case where the number of key-value lookups per request (or the request size) increases by a factor of 2 and 4 respectively. Because request sizes are dramatically larger, and because the number of servers in our testbed is limited, we compensate for the increase in request size by scaling down the request rate by the same factor. Thus, in Table 4, request sizes are a factor of two larger than in Table 3, but the request rate is half that of Table 3. The $\mathbf{T_{95}}$ values are expected to increase as compared with Table 3 because each request now takes longer to complete (since it does more key-value lookups).

Looking at *AutoScale* in Table 4, we see that $\mathbf{T_{95}}$ increases to around 700ms, while in Table 5, it increases to around 1200ms. This is to be expected. By contrast, for all other dynamic capacity management policies, the $\mathbf{T_{95}}$ values exceed one minute, both in Tables 4 and 5. Again, this is because these policies react only to changes in the request rate, and thus end up typically under-provisioning. *AlwaysOn* knows the peak load ahead of time, and thus, always keeps $\mathbf{N_{avg}} = 14$ servers on. As expected, the $\mathbf{T_{95}}$ values for *AlwaysOn* are quite good, but $\mathbf{P_{avg}}$ and $\mathbf{N_{avg}}$ are very high. Comparing *AutoScale* and *Opt*, we see that *Opt*'s power consumption and server usage is again only about 30% less than that of *AutoScale*.

Table 6 illustrates another way in which load can change. Here, we return to the 1x request size, but this time all servers have been slowed down to a frequency of 1.6 GHz as compared with the default frequency of 2.26 GHz. By slowing down the frequency of the servers, $\mathbf{T_{95}}$ naturally increases. We find that for all the dynamic capacity management policies, except for *AutoScale*, the $\mathbf{T_{95}}$ shoots up. The reason is that these other dynamic capacity management policies provision capacity based on the request rate. Since the request rate has not changed as compared to Table 3, they typically end up under-provisioning, now that servers are slower. The $\mathbf{T_{95}}$ for *AlwaysOn* does not shoot up because even in Table 3, it is greatly over-provisioning by provisioning for the peak load at all times. Since the *AutoScale* policy is robust to all changes in load, it provisions correctly, resulting in acceptable $\mathbf{T_{95}}$ values. $\mathbf{P_{avg}}$ and $\mathbf{N_{avg}}$ values for *AutoScale* continue to be much lower than that of *AlwaysOn*, similar to Table 3.

Figure 15 shows the server behavior under *AutoScale* for the Dual phase trace for request sizes of 1x, 2x and 4x. Clearly, *AutoScale* is successful at handling the changes in load due to both, changes in request rate and changes in request size.

Trace \ Policy		*AlwaysOn*	*Reactive*	*Predictive MWA*	*Predictive LR*	*Opt*	*AutoScale*
Slowly varying [18]	$\mathbf{T_{95}}$	478ms	> 1 min	> 1 min	> 1 min	531ms	701ms
	$\mathbf{P_{avg}}$	2,127W	541W	597W	728W	667W	923W
	$\mathbf{N_{avg}}$	14.0	3.2	2.7	3.8	4.0	5.4
Dual phase [31]	$\mathbf{T_{95}}$	424ms	> 1 min	> 1 min	> 1 min	532ms	726ms
	$\mathbf{P_{avg}}$	2,190W	603W	678W	1,306W	996W	1,324W
	$\mathbf{N_{avg}}$	14.0	3.0	2.6	6.6	5.8	7.3

Table 4: Comparison of all policies for 2x request size[2].

Trace \ Policy		*AlwaysOn*	*Reactive*	*Predictive MWA*	*Predictive LR*	*Opt*	*AutoScale*
Slowly varying [18]	$\mathbf{T_{95}}$	759ms	> 1 min	> 1 min	> 1 min	915ms	1,155ms
	$\mathbf{P_{avg}}$	2,095W	280W	315W	391W	630W	977W
	$\mathbf{N_{avg}}$	14.0	1.9	1.7	2.1	4.0	5.7
Dual phase [31]	$\mathbf{T_{95}}$	733ms	> 1 min	> 1 min	> 1 min	920ms	1,217ms
	$\mathbf{P_{avg}}$	2,165W	340W	389W	656W	985W	1,304W
	$\mathbf{N_{avg}}$	14.0	1.7	1.8	3.2	5.9	7.2

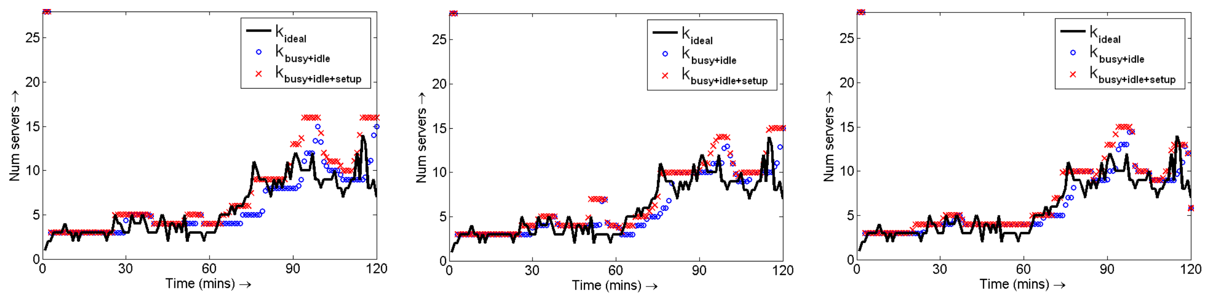
Table 5: Comparison of all policies for 4x request size[2].

Trace \ Policy		*AlwaysOn*	*Reactive*	*Predictive MWA*	*Predictive LR*	*Opt*	*AutoScale*
Slowly varying [18]	$\mathbf{T_{95}}$	572ms	> 1 min	> 1 min	3,339ms	524ms	760ms
	$\mathbf{P_{avg}}$	2,132W	903W	945W	863W	638W	1,123W
	$\mathbf{N_{avg}}$	14.0	5.7	5.9	4.8	4.0	7.2
Dual phase [31]	$\mathbf{T_{95}}$	362ms	24,401ms	23,412ms	2,527ms	485ms	564ms
	$\mathbf{P_{avg}}$	2,147W	1,210W	1,240W	2,058W	1,027W	1,756W
	$\mathbf{N_{avg}}$	14.0	6.3	7.4	12.2	5.9	10.8

Table 6: Comparison of all policies for lower CPU frequency.

Tables 4, 5 and 6 clearly indicate the superior robustness of *AutoScale* which uses n_{sys} to respond to changes in load, allowing *AutoScale* to respond to all forms of changes in load.

[2]For a given arrival trace, when request size is scaled up, the size of the application tier should ideally be scaled up as well so as to accommodate the increased load. However, since our application tier is limited to 28 servers, we follow up an increase in request size with a proportionate decrease in request rate for the arrival trace. Thus, the peak load (request rate times request size) is the same before and after the request size increase, and our 28 server application tier suffices for the experiment. In particular, *AlwaysOn*, which knows the peak load ahead of time, is able to handle peak load by keeping 14 servers on even as the request size increases.



(a) 1x: $\mathbf{T_{95}}$=474ms $\mathbf{P_{avg}}$=1,387W $\mathbf{N_{avg}}$=7.6

(b) 2x: $\mathbf{T_{95}}$=726ms $\mathbf{P_{avg}}$=1,324W $\mathbf{N_{avg}}$=7.3

(c) 4x: $\mathbf{T_{95}}$=1,217ms $\mathbf{P_{avg}}$=1,304W $\mathbf{N_{avg}}$=7.2

Figure 15: Robustness of *AutoScale* to changes in request size. The request size is 1x (or 120ms) in (a), 2x (or 240ms) in (b), and 4x (or 480ms) in (c).

5 Limitations of our work

Our evaluation thus far has demonstrated the potential benefits of using *AutoScale*. However, there are some limitations to our work, which we discuss below.

1. The design of *AutoScale* includes a few key parameters: t_{wait} (see Table 2), p (derived in Figure 6), ρ_{ref} (derived in Equation 2), and the ρ_{srv} vs. n_{srv} relationship (derived in Figure 14(a)). In order to deploy *AutoScale* on a given cluster, these parameters need to be determined. Fortunately, all of the above parameters only need to be determined once for a given cluster. This is because these parameters depend on the specifications of the system, such as the server type, the setup time, and the application, which do not change at runtime. Request rate, request size, and server speed, can all change at runtime, but these do not affect the value of the above key parameters (see Section 4 for more details).

2. In Section 4, we considered a few different forms of changes in load, such as changes in request size and changes in server speed, as well as changes in request rate. However, in production environments, load can change in many additional ways. For example, consider a scenario where some of the servers slow down due to software updates, while other servers are being backed up, and the rest of the servers are experiencing network delays. Evaluating *AutoScale* under all such scenarios is beyond the scope of this paper.

3. Our experimental evaluation is limited to a multi-tier testbed consisting of 38 servers, serving a web site with a key-value workload. Our testbed comprises an Apache load balancer, a homogenous application tier running php, and a memcached tier with a persistent back-end database. There are a variety of other application testbeds that we could have considered, ranging from single-tier stateless applications to complex multi-tier applications that are deployed in the industry today. The key feature that *AutoScale* depends on is having some servers that are stateless, and can thus be turned off or repurposed to save power/cost. Fortunately, many applications have this feature. For example, Facebook [12], Amazon [10] and

Windows Live Messenger [8], all use stateless servers as part of their platform. Thus, even though we have a very specific testbed, it is representative of many real-world applications.

6 Prior Work

Dynamic capacity management can be divided into two types: reactive (a.k.a. control-theoretic) approaches and predictive approaches. Reactive approaches, e.g., [24, 29, 13, 39, 40, 11], all involve reacting immediately to the current request rate (or the current response time, or current CPU utilization, or current power, etc.) by turning servers on or off. When the setup time is high (260s), these can be inadequate for meeting response time goals because the effect of the increased capacity only happens 260 seconds later.

Predictive approaches, e.g., [23, 33, 6, 17], aim to predict what the request rate will be 260 seconds from now, so that they can start turning on servers now if needed. Predictive or combined approaches work well when workload is periodic or seasonal, e.g. [8, 9, 4, 37, 15, 36, 14]. However when traffic is bursty and future arrivals are unknown, it is clearly hard to predict what will happen 260 seconds into the future.

We now discuss in detail the relevant prior work in predictive approaches and reactive approaches.

Predictive approaches
Krioukov et al. [23] use various predictive policies, such as Last Arrival, *MWA*, Exponentially Weighted Average and *LR*, to predict the future request rate (to account for setup time), and then accordingly add or remove servers from a heterogenous pool. The authors evaluate their dynamic provisioning policies by simulating a multi-tier web application. The authors find that *MWA* and *LR* work best for the traces they consider (Wikipedia.org traffic), providing significant power savings over *AlwaysOn*. However, the *AlwaysOn* version used by the authors does not know the peak request rate ahead of time (in fact, in many experiments they set *AlwaysOn* to provision for twice the historically observed peak), and is thus not as powerful an adversary as the version we employ.

Chen et al. [8] use auto-regression techniques to predict the request rate for a seasonal arrival pattern, and then accordingly turn servers on and off using a simple threshold policy. The authors evaluate their dynamic provisioning policies via simulation for a single-tier application. The authors find that their dynamic provisioning policy performs well for periodic request rate patterns that repeat, say, on a daily basis. The authors evaluate their policies via simulation in a single-tier setting. While the setup in [8] is very different (seasonal arrival patterns) from our own, there is one similarity to *AutoScale* in their approach: like *AutoScale*, the authors in [8] use the index-based routing (see Section 3.5). However, the policy in [8] does not have any of the robustness properties of AutoScale, nor the t_{wait} timeout idea.

Reactive and mixed approaches
Horvath et al. [17] employ a reactive feedback mechanism, similar to the *Reactive* policy in this paper, coupled with a non-linear regression based predictive approach to provision capacity for a multi-tier web application. In particular, the authors monitor server CPU utilization and job response times, and react by adding or removing servers based on the difference between observed response time and target response time. The authors evaluate their reactive approach via implementation in a multi-tier setting.

In Urgaonkar et al. [36] and Gandhi et al. [14], the authors assume a different setup from our own, whereby request rate is divided into two components, a long-term trend which is predictable, and short-term variations which are unpredictable. The authors use predictive approaches to provision servers for long-term trends (over a few hours) in request rates and then use a reactive controller, similar to the *Reactive* used in this paper, to react to short-term variations in request rate.

While the above hybrid approaches can leverage the advantages of both predictive and reactive approaches, they are not robust to changes in request size or server efficiency (see Section 4). In fact, none of the prior work has considered changes in request size or server efficiency.

There is also a long list of papers that look at dynamic capacity management in the case of negligible setup times (see, for example, [7, 25]). However, our focus in this paper is on dynamic capacity management *in the face of setup times*.

7 Conclusion and Future Work

This paper considers dynamic capacity management policies for data centers facing bursty and unpredictable load so as to save power/resources without violating response time SLAs. The difficulty in dynamic capacity management is the large setup time associated with getting servers back on. Reactive approaches that simply scale capacity based on the current request rate are too rash to turn servers off, especially when request rate is bursty. Given the huge setup time needed to turn servers back on, response times suffer greatly when request rate suddenly rises. Predictive approaches that work well when request rate is periodic or seasonal, perform very poorly in our case where traffic is unpredictable. Furthermore, as we show in Section 3.3, leaving a fixed buffer of extra capacity is also not the right solution.

AutoScale takes a fundamentally different approach to dynamic capacity management than has been taken in the past. First, *AutoScale* does not try to predict the future request rate. Instead, *AutoScale* introduces a smart policy to automatically provision spare capacity, which can absorb unpredictable changes in request rate. We make the case that to successfully meet response time SLAs, it suffices to simply manage existing capacity carefully and not give away spare capacity recklessly (see Table 3). Second, *AutoScale* is able to handle unpredictable changes not just in the request rate but also unpredictable changes in the request size (see Tables 4 and 5) and the server efficiency (see Table 6). *AutoScale* does this by provisioning capacity using not the request rate, but rather the number of requests in the system, which it is able to translate into the correct capacity via a novel, non-trivial algorithm. As illustrated via our experimental results in Tables 3 to 6, *AutoScale* outclasses existing optimized predictive and reactive policies in terms of consistently meeting response time SLAs. While *AutoScale*'s 95%ile response time numbers are usually less than one second, the 95%ile response times of existing predictive and reactive policies often exceed one full minute!

Not only does *AutoScale* allow us to save power while meeting response time SLAs, but it also allows us to save on rental costs when leasing resources (physical or virtual) from cloud service providers by reducing the amount of resources needed to successfully meet response time SLAs.

While one might think that *AutoScale* will become less valuable as setup times decrease (due to, example, sleep states or virtual machines), we find that this is not the case. *AutoScale* can significantly lower response times when compared to existing policies even for low setup times (see Figure 9). In fact, even when the setup time is only 20s, *AutoScale* can lower 95%ile response times by a factor of 3.

References

[1] Amazon Inc. Amazon elastic compute cloud (Amazon EC2), 2008.

[2] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, 2009.

[3] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007.

[4] N. Bobroff, A. Kochut, and K. Beaty. Dynamic Placement of Virtual Machines for Managing SLA Violations. In *Proceedings of the 10th IFIP/IEEE International Symposium on Integrated Network Management*, IM '07, pages 119–128, Munich, Germany, 2007.

[5] Peter Bodík, Rean Griffith, Charles Sutton, Armando Fox, Michael Jordan, and David Patterson. Statistical machine learning makes automatic control practical for internet datacenters. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, HotCloud '09, San Diego, CA, 2009.

[6] Malu Castellanos, Fabio Casati, Ming-Chien Shan, and Umesh Dayal. ibom: A platform for intelligent business operation management. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 1084–1095, Tokyo, Japan, 2005.

[7] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, and Amin M. Vahdat. Managing energy and server resources in hosting centers. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 103–116, Chateau Lake Louise, Banff, Canada, 2001.

[8] Gong Chen, Wenbo He, Jie Liu, Suman Nath, Leonidas Rigas, Lin Xiao, and Feng Zhao. Energy-aware server provisioning and load dispatching for connection-intensive internet services. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '08, pages 337–350, San Francisco, CA, 2008.

[9] Yiyu Chen, Amitayu Das, Wubi Qin, Anand Sivasubramaniam, Qian Wang, and Natarajan Gautam. Managing server energy and operational costs in hosting centers. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '05, pages 303–314, Banff, Alberta, Canada, 2005.

[10] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, Stevenson, WA, 2007.

[11] E.N. Elnozahy, Michael Kistler, and Ramakrishnan Rajamony. Energy-efficient server clusters. In *Proceedings of the 2nd Workshop on Power-Aware Computing Systems*, WPACS '02, pages 179–196, Cambridge, MA, 2002.

[12] Facebook. Personal communication with Facebook., 2011.

[13] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 13–23, San Diego, CA, 2007.

[14] A. Gandhi, Y. Chen, D. Gmach, M. Arlitt, and M. Marwah. Minimizing data center sla violations and power consumption via hybrid resource provisioning. In *Proceedings of the 2nd International Green Computing Conference*, IGCC '11, Orlando, FL, 2011.

[15] Daniel Gmach, Stefan Krompass, Andreas Scholz, Martin Wimmer, and Alfons Kemper. Adaptive quality of service management for enterprise services. *ACM Trans. Web*, 2(1):1–46, 2008.

[16] Dirk Grunwald, Charles B. Morrey, III, Philip Levis, Michael Neufeld, and Keith I. Farkas. Policies for dynamic clock scheduling. In *Proceedings of the 4th Conference on Symposium of Operating System Design and Implementation*, OSDI '00, San Diego, CA, 2000.

[17] Tibor Horvath and Kevin Skadron. Multi-mode energy management for multi-tier server clusters. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 270–279, Toronto, Ontario, Canada, 2008.

[18] ita. The internet traffic archives: WorldCup98. http://ita.ee.lbl.gov/html/contrib/WorldCup.html, 1998.

[19] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: a disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Proceedings of the eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 117–130, Banff, Alberta, Canada, 2001.

[20] J. Kim and T. S. Rosing. Power-aware resource management techniques for low-power embedded systems. In S. H. Son, I. Lee, and J. Y-T Leung, editors, *Handbook of Real-Time and Embedded Systems*. Taylor-Francis Group LLC, 2006.

[21] Avi Kivity. kvm: the Linux virtual machine monitor. In *Proceedings of the 2007 Ottawa Linux Symposium*, OLS '07, pages 225–230, Ottawa, Canada, 2007.

[22] Leonard Kleinrock. *Queueing Systems, Volume I: Theory*. Wiley-Interscience, 1975.

[23] Andrew Krioukov, Prashanth Mohan, Sara Alspaugh, Laura Keys, David Culler, and Randy Katz. Napsac: Design and implementation of a power-proportional web cluster. In *Proceedings of the First ACM SIGCOMM Workshop on Green Networking*, Green Networking '10, pages 15–22, New Delhi, India, 2010.

[24] Julius C.B. Leite, Dara M. Kusic, and Daniel Mossé. Stochastic approximation control of power and tardiness in a three-tier web-hosting cluster. In *Proceeding of the 7th International Conference on Autonomic Computing*, ICAC '10, pages 41–50, Washington, DC, 2010.

[25] Seung-Hwan Lim, Bikash Sharma, Byung Chul Tak, and Chita R. Das. A dynamic energy management in multi-tier data centers. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '11, pages 257–266, Austin, TX, 2011.

[26] Yung-Hsiang Lu, Eui-Young Chung, Tajana Šimunić, Luca Benini, and Giovanni De Micheli. Quantitative comparison of power management algorithms. In *Proceedings of the conference on Design, Automation and Test in Europe*, DATE '00, pages 20–26, Paris, France, 2000.

[27] David Meisner, Christopher M. Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. Power management of online data-intensive services. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 319–330, 2011.

[28] David Mosberger and Tai Jin. httperf—A Tool for Measuring Web Server Performance. *ACM Sigmetrics: Performance Evaluation Review*, 26(3):31–37, 1998.

[29] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: Managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 237–250, Paris, France, 2010.

[30] M. E. J. Newman. Power laws, pareto distributions and zipf's law. *Contemporary Physics*, 46:323–351, December 2005.

[31] nlanr. National Laboratory for Applied Network Research. Anonymized access logs. ftp://ftp.ircache.net/Traces/, 1995.

[32] Trevor Pering, Tom Burd, and Robert Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the International Symposium on Low Power Electronics and Design*, ISLPED '98, pages 76–81, Monterey, CA, 1998.

[33] W. Qin and Q. Wang. Modeling and control design for performance management of web servers via an IPV approach. *IEEE Transactions on Control Systems Technology*, 15(2):259–275, March 2007.

[34] sap. SAP application trace from anonymous source., 2011.

[35] Bill Snyder. Server virtualization has stalled, despite the hype. http://www.infoworld.com/print/146901, December 2010.

[36] Bhuvan Urgaonkar and Abhishek Chandra. Dynamic provisioning of multi-tier internet applications. In *Proceedings of the Second International Conference on Automatic Computing*, ICAC '05, pages 217–228, Seattle, WA, 2005.

[37] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. An analytical model for multi-tier internet services and its applications. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '05, pages 291–302, Banff, Alberta, Canada, 2005.

[38] Akshat Verma, Gargi Dasgupta, Tapan Kumar Nayak, Pradipta De, and Ravi Kothari. Server workload analysis for power minimization using consolidation. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX '09, San Diego, CA, 2009.

[39] X. Wang and M. Chen. Cluster-level feedback power control for performance optimization. In *Proceeding of the 14th IEEE International Symposium on High-Performance Computer Architecture*, HPCA '08, pages 101–110, Salt Lake City, UT, 2008.

[40] Timothy Wood, Prashant J. Shenoy, Arun Venkataramani, and Mazin S. Yousif. Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the 4th USENIX conference on Networked Systems Design and Implementation*, NSDI '07, pages 229–242, Cambridge, MA, 2007.