# SOFTScale: Stealing Opportunistically For Transient Scaling

**Anshul Gandhi**[*] **Timothy Zhu**[*]
**Mor Harchol-Balter**[*] **Michael Kozuch**[†]

August 2012
CMU-CS-12-111R

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; load spikes; resource provisioning; power management

## Abstract

Dynamic capacity provisioning is a well studied approach to handling *gradual* changes in data center load. However, *abrupt* spikes in load are still problematic in that the work in the system rises very quickly during the setup time needed to turn on additional capacity. Performance can be severely affected even if it takes only 5 seconds to bring additional capacity online.

In this paper, we propose SOFTScale, an approach to handling load spikes in multi-tier data centers without having to over-provision resources. SOFTScale works by opportunistically stealing resources from other tiers to alleviate the bottleneck tier, even when the tiers are carefully provisioned at capacity. SOFTScale is especially useful during the transient overload periods when additional capacity is being brought online.

Via implementation on a 28-server multi-tier testbed, we investigate a range of possible load spikes, including an artificial doubling or tripling of load, as well as large spikes in real traces. We find that SOFTScale can meet our stringent 95th percentile response time Service Level Agreement goal of 500ms without using any additional resources even under some extreme load spikes that would normally cause the system (without SOFTScale) to exhibit response times as high as 96 seconds.

# 1 Introduction

Data centers play an important role in today's IT infrastructure. Government organizations, hospitals, financial trading firms, and major IT companies, such as Google, Facebook and Amazon, all rely on data centers for their daily business activities. A primary goal for data center operators is to provide good response times to users; these response time targets typically translate to some response time Service Level Agreements (SLAs). A secondary goal is to reduce operational costs by exploiting the variability in user demand. By scaling capacity to match current demand, operators can either: (i) reduce power consumption by turning off unneeded servers, or (ii) save on rental costs by releasing unneeded virtual machines, or (iii) get additional work done by repurposing unneeded servers for other tasks.

Data center services today are often organized as multiple tiers. Typically, one of these tiers is an *application tier* that processes requests, and another tier is the *data tier* that is responsible for efficiently delivering data back to the application tier. While it is possible to physically collocate the application tier and the data tier on the same servers, dividing the architecture into physically different tiers is preferable because it makes it easier to scale and manage the individual tiers [14, 34, 36]. The data tier is stateful, and is almost never turned off [35, 9], even if there is a significant drop in load [8]. The application tier, on the other hand, is usually stateless and can be dynamically scaled using existing reactive [25, 28, 32], predictive [22, 18] or mixed [36, 17, 16] approaches, *provided that the load does not change too abruptly*.

Unfortunately, abrupt changes in load, or load spikes, are all too common in today's data centers. Important events, such as the September 11 attacks [24, 19], earthquakes or other natural disasters [39], slashdot effects [3], Black Friday shopping [12], or sporting events, such as the Super Bowl [30] or the Soccer World Cup [7], are common causes of load spikes for website traffic. Service outages [31] or server failures [33] can also result in abrupt changes in load caused by a sharp drop in capacity. While some of the above events are predictable, most of them *cannot* be predicted in advance.

Abrupt changes in load are especially problematic since adding capacity requires some time, which we call *setup time*, denoted by $t_{setup}$. Even if we instantaneously detect a spike in load, it will still take the system at least the setup time to add the required capacity. In our lab, the setup time for turning on an additional server is approximately 5 minutes. Likewise, the setup time needed to create virtual machines (VMs) can range anywhere from 30 seconds – 1 minute if the VMs are locally created (based on our measurements using kvm [21]) or 5 – 10 minutes if the VMs are obtained from a cloud computing platform (see, for example, [5]). All these numbers are extremely high, and can result in long periods where the SLA is violated.

Throughout the paper, we focus on the performance of the system during the setup time following a load spike. Since no additional capacity can be added during the setup time, the system has a fixed number of servers online, and we refer to such a system as the **baseline**. A typical SLA requires that the 95th percentile of response time, denoted by $\mathbf{T_{95}}$, stay below 500ms[1]. In

[1]Our choice of SLA is motivated by recent studies [36, 22, 13] which indicate that 95th percentile guarantees of several hundred milliseconds are typical.

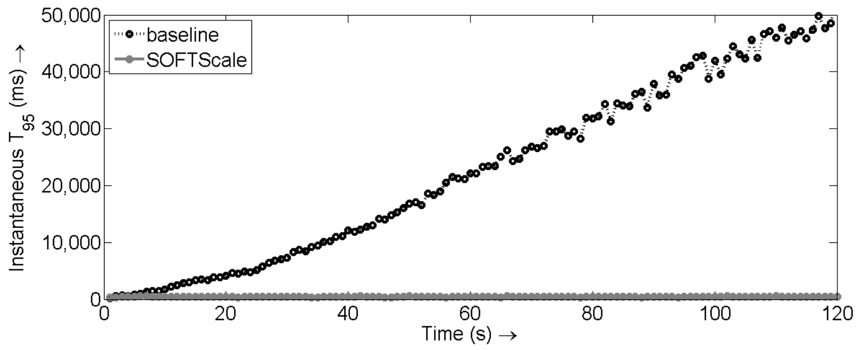


Figure 1: *Using SOFTScale, we can meet response time SLAs even under a 15% to 30% load jump. Note that the y-axis ranges from 0s to 50s.*

this paper, we consider the more difficult goal of meeting the $T_{95}$ requirements during the setup time (i.e., after the onset of the spike, and before additional servers can be brought online). This is equivalent to saying that *no more than 5% of all requests that arrive during the setup time are allowed to exceed the 500ms response time*. In addition to the $T_{95}$ (which measures over the entire setup time), in some plots, we also show the "instantaneous $T_{95}$", which is the 95th percentile of response times collected every second.

Consider a system which has the appropriate number of application servers turned on to ensure that the 95th percentile of response times stays below 500ms at the current load of 15% of peak load. Here, peak load refers to the maximum load that our system can handle (see Section 2 for details of our experimental testbed). Now, imagine that the load suddenly increases to 30%. The time needed to turn on the necessary additional servers is the setup time, say 5 minutes. We say that our system can "handle" a load jump if $T_{95} \leq 500ms$ during the setup time. As shown in Figure 1, our baseline system is not able to handle the 15% to 30% load jump. The black dots in Figure 1 show the increase in instantaneous $T_{95}$ during the first two minutes of the setup time under the baseline, where the system is clearly under-provisioned during this time. The data for Figure 1 is generated from experiments running on our implementation testbed using a key-value based workload (see Section 2 for full details of our experimental testbed). As shown in the figure, instantaneous $T_{95}$ increases rapidly over time, reaching 50 seconds after only two minutes. Even if future hardware reduces this setup time to 10 seconds, we see that instantaneous $T_{95}$ can be well over 3 seconds.

In order to avoid setup times, data center operators typically over-provision capacity at all times (since load spikes are often unpredictable). For example, to handle a 15% to 30% load jump, one needs to over-provision resources by a factor of 2. Clearly, such an approach is quite expensive.

We propose SOFTScale, an approach that allows data centers to handle load spikes without having to over-provision resources and incur costs. SOFTScale leverages the fact that the data tier in a multi-tier data center is always left on [35, 9, 8]. Thus, during the setup time following a load spike, we can use these "always on" data tier servers to do some of our application work. SOFTScale involves running the application tier software on the data tier servers, where this software is only used during the setup time. We refer to this notion as "stealing" of the data tier capacity. SOFTScale requires no additional resources and can even handle a doubling of load, so long as the final load is not too high. Returning to our example where the load instantaneously doubles from 15% to 30%, we see that SOFTScale, denoted by the flat gray line in Figure 1, allows the instantaneous $T_{95}$ to stay within the 500ms SLA at all times. While stealing from the data tier can increase the latency of data operations, the overall benefit of being able to meet SLAs during setup times makes a compelling case for using SOFTScale. Note that one could theoretically use SOFTScale even after the setup time, however, the (non-zero) increase in latency of data operations as a result of using SOFTScale suggests otherwise. The SOFTScale middleware is depicted in Figure 2, and is described in detail in Section 3.4.

Almost all papers on dynamic capacity management (see, for example, [25, 28, 32, 22, 18, 36, 17, 16]) deal with new approaches to scale capacity in response to changes in load. However, such approaches can be ineffective during the setup time, as shown in Figure 1. SOFTScale is a complementary solution that aims to improve performance *specifically during the setup time*, and is meant to be used in conjunction with any existing dynamic capacity management approach.

While the concept behind SOFTScale seems obvious, there are some practical difficulties that may have led researchers to dismiss this idea as "unworkable", hence the lack of publications on this idea. First, there's the question of *when* is SOFTScale useful. Since the data tier is provisioned to handle peak load, invoking SOFTScale when the data tier is already bottlenecked will lead to SLA violations. Second, there's the question of *how much* can we steal from the data tier. If we end up stealing too much from the data tier, the overall system performance might degrade. Third, there's the fear that running application work on the data tier servers will interfere with data delivery work, and can possibly lead to SLA violations. Finally, there's the fear that implementing SOFTScale is too complicated.

In this paper, we demonstrate via implementation that the SOFTScale middleware is a practical solution that allows us to meet response time SLAs even when load increases suddenly by a factor of 2, provided that the load is not too high. In particular, this paper makes the following contributions:

- We determine *load regimes* for which SOFTScale can be successfully applied to handle load spikes (see Section 3.1). This addresses the question of *when* to invoke SOFTScale. Further, identifying load regimes where SOFTScale is *not* beneficial avoids accidental overload of the data tier.
- We determine *how much* data tier capacity can be leveraged by SOFTScale for a given load (see Section 3.2). This enables us to steal the *right amount* of capacity from the data tier without hurting overall response time.
- We show that it is possible to avoid interference between the application work and the data delivery work on the data servers by simply *isolating* these processes to different CPU cores (see Section 3.3).
- We outline the steps needed to implement the SOFTScale middleware (see Section 3.4). In

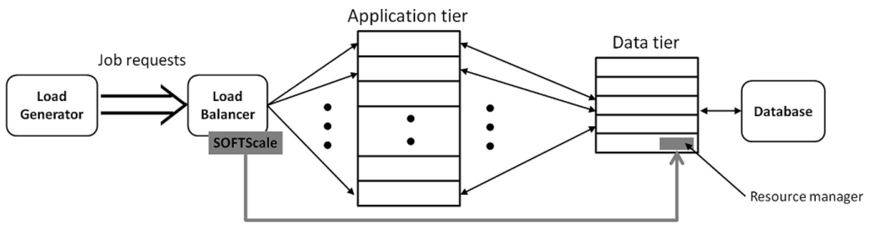


Figure 2: *Our experimental testbed.*

our testbed, we implemented SOFTScale by adding less than a thousand lines of code in the Apache load balancer.

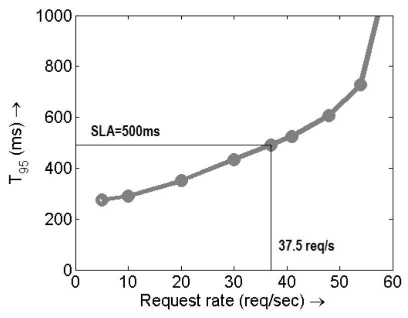
- We present an analytical model that estimates the system performance under SOFTScale (see Section 3.5), which allows us to predict the performance of SOFTScale for a range of multi-tier systems.

We evaluate SOFTScale via implementation on a 28-server multi-tier testbed hosting a key-value based application built along the lines of Facebook or Amazon. Our implementation results show that SOFTScale can be used to handle instantaneous load spikes (see Section 4.1), load spikes seen in real-world traces (see Section 4.2) as well as load spikes caused by server failures (see Section 4.3). To fully investigate the applicability of SOFTScale, we experiment with multiple setup times ranging from 5 minutes (see Section 4) all the way down to 5 seconds (see Section 5). Our results indicate that SOFTScale can provide huge benefits across the entire spectrum of setup times. We also investigate the applicability of SOFTScale in future server architectures which may have a larger number of CPU cores per server. Our results (see Section 6) indicate that SOFTScale will be even more beneficial in such cases.

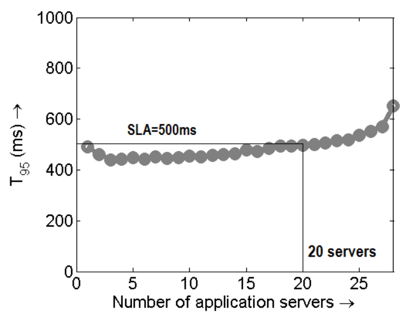
## 2 Our experimental testbed

Figure 2 illustrates our experimental testbed. The gray components make up SOFTScale, and will be described in detail in Section 3.4. We employ one server as the front-end load generator running httperf [27]. Another server is used as a load balancer running the Apache HTTP Server, which distributes incoming PHP requests to the application servers. Each application server communicates with the data tier, which in our setup comprises memcached servers, to retrieve data required to service the requests. Another server is used to store the entire data set, a billion key-value pairs, on a database.

Throughout this paper we measure power consumption and use that as a proxy for all operational (resource) costs. We monitor the power consumption of individual servers by reading the power values from the power distribution unit. 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. The setup time for our servers is about $t_{setup} = 5$ minutes. How-



(a) Single application server.



(b) Application tier scaling.

Figure 3: *Figure (a) shows that a single application server can handle 37.5 req/s per server. Figure (b) shows that once we have more than 20 application servers, they can no longer handle 37.5 req/s per server because the memcached tier becomes the bottleneck.*

ever, we also examine the effects of lower $t_{setup}$[2]. We replicate this effect by not routing requests to a server if it is marked for sleep. When the server is marked for setup, we wait for $t_{setup}$ seconds before sending requests to the server.

## 2.1 Workload

We design a key-value workload to model realistic multi-tier applications such as the social networking site, Facebook, or e-commerce sites like Amazon [13]. Each generated request (or job) is a PHP script that runs on the application server. A request begins with the application server requesting a value for a random 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 2,200 key-value requests per job, which translates to a mean service time of approximately 200 ms, assuming no resource contention. The job size distribution is highly variable, with the largest job requiring roughly 20 times as many key-value requests as the smallest job.

In this paper, we use the Zipf [29] distribution to model the popularity of the initial random key request. To minimize the effects of misses in the memcached layer (which could result in an unpredictable fraction of the requests violating the response time SLA), we tune the parameters of the Zipf distribution so that only a negligible fraction of requests miss in the memcached layer.

[2]Lower setup times 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.

## 2.2 Provisioning

In order to demonstrate the effectiveness of SOFTScale, we tune our implementation testbed to have no spare capacity at the memcached tier at peak load. Our memcached tier comprises 5 servers, each with a 6-core Intel Xeon X5650 processor and 48GB of memory. However, we offline two cores[3] per server to be consistent with the specifications that were published by Facebook [26], leaving us with 4-core memcached servers. We now determine how many application servers we need to fully saturate the memcached tier.

Each of our application servers is a powerful 8-core (dual-socket) Intel Xeon E5520 processor-based server. We run an experiment where we have one application server and all five memcached servers, and we flood the system. We find that the application server can handle at most 37.5 req/s without violating the SLA, as shown in Figure 3(a).

We now examine how well the system scales as we add more application servers. Ideally, if we have $x$ application servers, the system should be able to handle a maximum request rate of at least $37.5 \times x$ req/s without violating the 500ms SLA. Figure 3(b) shows our scaling results, where we vary the number application servers from 1 to 28, and use a request rate of 37.5 req/s times the number of application servers. We see that the system scales perfectly up to 20 application servers. Once we have more than 20 application servers, we see that they can no longer handle 37.5 req/s per server. This is because at this *peak load*, which corresponds to $37.5 \times 20 = 750$ req/s, the memcached tier starts becoming a bottleneck. We validate our claim by ensuring that the other components in the system, namely the load generator, the load balancer, and the application servers, are not a bottleneck. Further, by monitoring the network bandwidth, we ensure that it is not a bottleneck. With this ratio of 20 application servers to 5 memcached servers, we ensure that the memcached tier is saturated. Thus, at least 5 memcached servers are needed to handle peak load (using more than 5 memcached servers only improves the performance of SOFTScale). This 4:1 ratio of application servers to memcached servers is consistent with Facebook [2].

Based on the above experiments, we conclude that the 5 memcached servers can handle at most 750 job req/s before they become a bottleneck. Thus, in our experiments, we limit our total request rate to 750 req/s, which we also refer to as peak load or 100% load. At peak load, we do not have any spare capacity on the memcached servers. Thus, we cannot "steal" any resources from memcached servers at high load without violating the 500ms SLA.

When running the system, the 5 memcached servers are always kept on. By contrast, the number of application servers needed at any time is $\lceil \frac{r}{37.5} \rceil$, where $r$ is the current request rate into the system. For example, if the current request rate is 15% of the peak (or 112 req/s), we provision $\lceil \frac{112}{37.5} \rceil = 3$ application servers. Now, if the load suddenly doubles from 15% (112 req/s) to 30% (225 req/s), we need 6 application servers in total. Thus, the 3 application servers that are currently on, become the bottleneck.

Note that the ratio of application servers to memcached servers is no longer 4:1 if the applica-

[3]Observe that weakening the memcached servers greatly hurts SOFTScale in that there is less capacity to steal, but we do this purposely to create a fully saturated memcached tier.

tion tier is scaled down. For example, if the current request rate is 25% of the peak (or 187 req/s), then we provision $\lceil \frac{187}{37.5} \rceil = 5$ application servers. This gives us a 1:1 ratio of application servers to memcached servers.

## 3 SOFTScale

The key idea behind SOFTScale is to leverage the computational power at the always on data tier servers to do some of our application work during the setup time while additional application tier capacity is being brought online. The motivation behind this idea is that, while our memcached servers are provisioned to have exactly the right amount of resources at high load (for our system, peak load is 750 req/s), there are extra resources available at low load. Thus, when the system load is low, we should be able to "steal" resources from the memcached servers to offset some of the workload at the bottlenecked application servers.

SOFTScale works by enhancing the Apache load balancer to route some of the application requests to the memcached servers during load spikes. Note that the software needed to process the application work will first have to be installed on the data tier servers. For our experimental testbed, this only involved installing the Apache web server with PHP support on the memcached servers. Further, our application software does not consume a lot of memory.

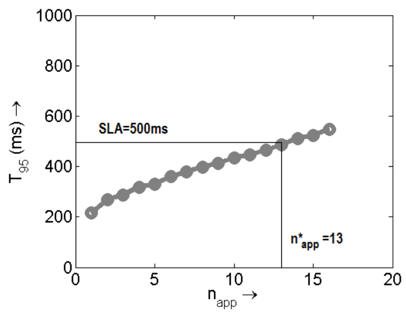
While SOFTScale sounds like a promising idea, exploiting the full potential of SOFTScale is challenging. We now describe SOFTScale by discussing the design decisions behind the algorithm.

### 3.1 When to invoke SOFTScale?

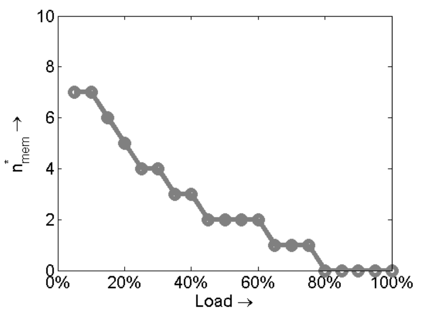
SOFTScale must be invoked *as soon as there is a spike in load.* A spike in load could be caused *either* by an increase in request rate *or* by a loss in application tier capacity (server failures or service outages).

If the spike in load is caused by a sudden increase in request rate, then the obvious approach to detect this spike would be to monitor request rate periodically. Unfortunately, request rate is a time-average value, and is thus not instantaneous enough to detect load spikes. We propose monitoring the *number of active requests* at each application server, $n_{app}$, to detect load spikes. If the system is under-provisioned because the request rate is too high, then $n_{app}$ will immediately increase. Monitoring $n_{app}$ is fairly straightforward, and many modern systems, including the Apache load balancer, already track this value.

Spikes in load can also be caused by a sudden loss in application tier capacity (server failures or service outages). In this case, request rate cannot be used to detect the spike. Fortunately, $n_{app}$ is immediately responsive to server failures, since it increases instantaneously when the application tier capacity drops.



(a) Application server.



(b) Memcached server.

Figure 4: *Figure (a) shows that we should invoke SOFTScale whenever the number of requests at the application server exceeds 13. Figure (b) shows $n^*_{mem}$, the optimal number of application requests that can be simultaneously handled by a memcached server without violating the 500ms SLA, as a function of the total system load.*

We must invoke SOFTScale when $n_{app}$ becomes so high that the $T_{95}$ SLA is in danger of being violated. In particular, if $n^*_{app}$ is the maximum number of simultaneous requests that a single application server can handle without violating the SLA, then we invoke SOFTScale as soon as $n_{app}$ exceeds $n^*_{app}$ for all application servers. Of course, one can also be conservative and invoke SOFTScale even when $n_{app}$ is below $n^*_{app}$.

An easy way to determine $n^*_{app}$ is by profiling the application servers. We run a closed-loop experiment with a single application server where we fix the number of simultaneous requests in the system ($n_{app}$), and monitor $T_{95}$. Figure 4(a) shows our results. We see that, for our system, $n^*_{app} = 13$. This same technique (profiling the application servers) can be used for determining $n^*_{app}$ for different systems as well. Note that $n^*_{app}$ corresponds to the 37.5 req/s that each application server can handle. Since we provision the application tier so as not to exceed 37.5 req/s at each server, *a reading of* $n_{app} > 13$ *indicates overload*. Thus, we invoke SOFTScale as soon as the load balancer detects that $n_{app}$ has exceeded 13 for all the application servers.

## 3.2 How much application work can memcached handle?

Now that we know *when* to invoke SOFTScale (and thus, *when* to attempt to steal resources from the data tier), the next design question is: *how much* can we steal? The memcached servers are primarily responsible for providing data to the application work. Thus, we cannot overload memcached servers with too much application work. Figure 4(b) shows $n^*_{mem}$, the maximum number of application requests that a memcached server can handle simultaneously without violating the SLA. We see that $n^*_{mem}$ depends on the overall system load, as should be expected. When the system load is low ($< 20\%$), each memcached server can handle almost half the work capacity of an application server, whereas when the load is high ($\geq 80\%$), memcached servers cannot handle any application work.

In order to determine $n^*_{mem}$, we use the following methodology. For each initial load, we determine the number of application servers needed to handle this load, say $k$ servers, using the arguments in Section 2.2. We then run a closed-loop experiment with $k$ application servers and all 5 memcached servers, and we fix the number of simultaneous requests in the system to $k \cdot n^*_{app}$. At this point, the application servers cannot handle any more simultaneous requests. We now increase the number of simultaneous requests in the system beyond $k \cdot n^*_{app}$, sending the additional requests to the memcached servers, and monitor $T_{95}$. Based on the initial load, each of the memcached servers will be able to handle some number of simultaneous requests, $n^*_{mem}$, before the $T_{95}$ for the system rises above the SLA. Thus, we can determine $n^*_{mem}$ using a simple profiling approach.

## 3.3 Need for isolation

While we have successfully overloaded the functionality of the memcached servers, we have not eliminated interference between the memcached work and the application work at the memcached servers. One way of reducing interference is to "isolate" these two processes at the memcached servers, by partitioning the four cores at the memcached server between the memcached work and the application work. We achieve this core isolation by using the `taskset` command in Linux. A logical way of partitioning the cores is in a 2:2 ratio, with 2 cores dedicated to memcached work and 2 cores dedicated to application work. However, we find that the performance of SOFTScale improves greatly if we *dynamically* adjust the partitioning based on total system load. For example, when the system load is extremely low, we can get away with restricting memcached to only one core at each memcached server and reserving the remaining three cores for application work in case of a load spike (1:3 partitioning). On the other hand, when the system load is very high, we need all four cores for memcached work (4:0 partitioning). Figure 5 shows $n^*_{mem}$ for the memcached servers with dynamic isolation and without any isolation (same as Figure 4(b)). Note the four discrete horizontal levels for dynamic isolation. These refer to a 4-core partitioning between the memcached work and application work in the ratio of 1:3, 2:2, 3:1 and 4:0 respectively. We see that dynamic isolation greatly enhances the capacity of memcached servers to handle application work. Henceforth, when we use SOFTScale, it will be implied that we are referring to SOFTScale with dynamic isolation.

We obtain Figure 5 by repeating the same closed-loop experiments described at the end of Section 3.2 for each possible partitioning of the 4 cores among the memcached work and the application work. We then select the partitioning that gives us the highest $n^*_{mem}$ value.

## 3.4 The SOFTScale algorithm

We are now ready to describe our SOFTScale algorithm, which is implemented in the load balancer, and is depicted in gray in Figure 2. We send application requests to the application servers, via Join-the-Shortest-Queue routing, as long as any server has less than $n^*_{app}$ simultaneous requests. If all of the application servers have at least $n^*_{app}$ requests, SOFTScale is invoked. SOFTScale sends

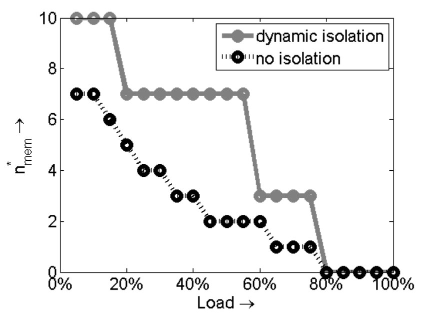


Figure 5: *The figure illustrates enhancement in SOFTScale using dynamic isolation.*

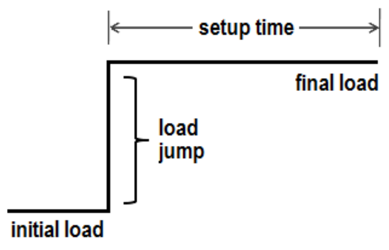


Figure 6: *The figure illustrates the load jumps we use in our experiments. Note that we only evaluate the system during the setup time.*

any additional requests above the $n^*_{app}$ requests to the memcached servers. The resource manager (see Figure 2) at each memcached server is responsible for invoking the software that will serve the incoming application requests. In our case, this software is the Apache web server with PHP support, which is invoked upon boot. The resource manager also isolates the application work from the memcached work. We limit the number of requests that we send to each memcached server to $n^*_{mem}$. Recall that $n^*_{mem}$, which is the optimal number of simultaneous application requests that a memcached server can handle, is not a constant, and in fact varies with load as specified in Section 3.3 and Figure 5. Note that $n^*_{mem} = 0$ if load is greater than or equal to 80% of peak load. Thus, SOFTScale will not send application requests to the memcached servers if load is high. Once we have $n^*_{mem}$ requests at all memcached servers, then we load balance additional requests among the application servers.

## 3.5 An analytical model for estimating SOFTScale's performance

We now present a simple analytical model that allows us to estimate the range of load jumps that SOFTScale can handle for a given multi-tier system. Let $k_{app}$ and $k_{mem}$ denote the total number of application servers and memcached servers in the system, respectively. If the current system load is x% of the peak load, where $0 \leq x \leq 100$, then the number of application servers on is roughly $k_{app} \cdot \frac{x}{100}$, assuming the application tier is dynamically scaled. Suppose that each memcached server can handle $n^*_{mem}$ simultaneous application requests at load x%. Then, the total number of application requests that the memcached tier can handle is $k_{mem} \cdot n^*_{mem}$. Note that the number of simultaneous requests that the system can handle without SOFTScale at load x% is $k_{app} \cdot \frac{x}{100} \cdot n^*_{app}$, where $n^*_{app}$ is the number of simultaneous requests than an application server can handle. Thus, at x% load, the fraction of additional load that the system can handle with SOFTScale is:

$$\text{Fraction of additional load that SOFTScale can handle} \approx \frac{k_{mem} \cdot n^*_{mem}}{k_{app} \cdot \frac{x}{100} \cdot n^*_{app}} \quad (1)$$

Equation (1) suggests that the additional load that SOFTScale can handle goes down as the system load (x%) increases, as expected (note that $n^*_{mem}$ also drops with system load, as shown in Figure 5). As we will show in Sections 4.1 and 6, Equation (1) matches our experimental results for

SOFTScale's performance. Thus, we can use Equation (1) to predict SOFTScale's performance for systems whose $k_{app}$, $k_{mem}$, $n^*_{app}$ or $n^*_{mem}$ values are different from ours.

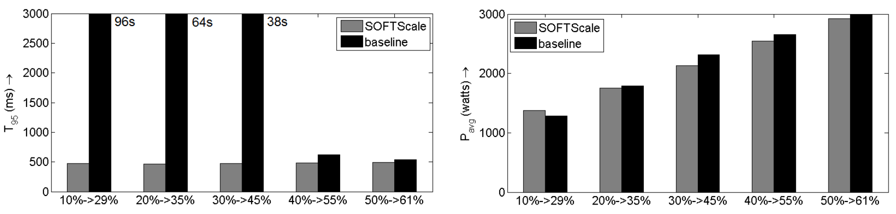
## 4 Results

We now evaluate the performance of SOFTScale for a variety of load spikes. We start in Section 4.1, where we consider a range of *instantaneous* load jumps and characterize the space of jumps that SOFTScale can handle. Then, in Section 4.2, we examine the performance of SOFTScale under real-world load spikes. Finally, in Section 4.3, we examine the performance of SOFTScale for load spikes that are caused by service outages or server failures. For all the experiments in this section, we consider $t_{setup} = 5$ minutes, which is the setup time for our servers. Later, in Section 5, we examine SOFTScale under lower setup times.

### 4.1 Characterizing the range of load jumps that SOFTScale can handle

In this section, we consider *instantaneous* jumps in load, as shown in Figure 6, and examine the system *only during the setup time*. We assume the system is properly provisioned for the initial load, and thus, is *under-provisioned* after the load jump, during the setup time. Under SOFTScale, although the application tier is under-provisioned during the setup time, we can use the memcached tier to compensate. By contrast, under the "baseline" architecture, we are limited to the capacity of the under-provisioned application tier. We compare SOFTScale with the "baseline" architecture by examining the following metrics: $T_{95}$, the 95th percentile of response times during the 5 minute setup time, and $P_{avg}$, the average power consumed by the application servers and the memcached servers during the setup time. Note that $P_{avg}$ is proportional to the amount of resources being used, and can thus be thought of as a proxy for operational costs. For a given load jump, if the system has $T_{95} \leq 500ms$, we say that it can "handle" the load jump.

Figure 7(a) shows the effect of SOFTScale on $T_{95}$ for specific load jumps. We choose these specific load jumps since they correspond to the maximum jump that SOFTScale can handle at each of the initial loads. For example, if the initial load is 10% of the peak, then SOFTScale can handle a maximum jump of $10\% \rightarrow 29\%$, where the load changes instantaneously from an initial load of 10% to a final load of 29%. We see that SOFTScale provides huge benefits in $T_{95}$, as long as the final load is less than 50%. In particular, the $T_{95}$ under SOFTScale is less than 500ms for the $10\% \rightarrow 29\%$ jump, as compared with 96s under the baseline. Likewise, SOFTScale lowers $T_{95}$ from 64s to less than 500ms for the $20\% \rightarrow 35\%$, and from 38s to less than 500ms for the $30\% \rightarrow 45\%$ load jump. SOFTScale provides these performance improvements by opportunistically stealing resources from the memcached servers to handle the critical application work. When the load jumps from $40\% \rightarrow 55\%$ and $50\% \rightarrow 61\%$, SOFTScale still provides improvement in $T_{95}$, but these improvements are not as dramatic. This is because the memcached tier is optimally provisioned (see Section 2.2), and thus has very little spare capacity at high loads.



(a) $T_{95}$ (b) $P_{avg}$

Figure 7: *SOFTScale meets $T_{95} = 500ms$ SLA without consuming any extra resources for a range of load jumps.*

By contrast, the baseline architecture (no SOFTScale) would have to resort to significant over-provisioning to handle the load jumps. For example, for the $10\% \rightarrow 29\%$ jump, the baseline would have to over-provision the application tier by about 190% to meet SLA goals during the setup time. Clearly, this is a huge waste of resources.

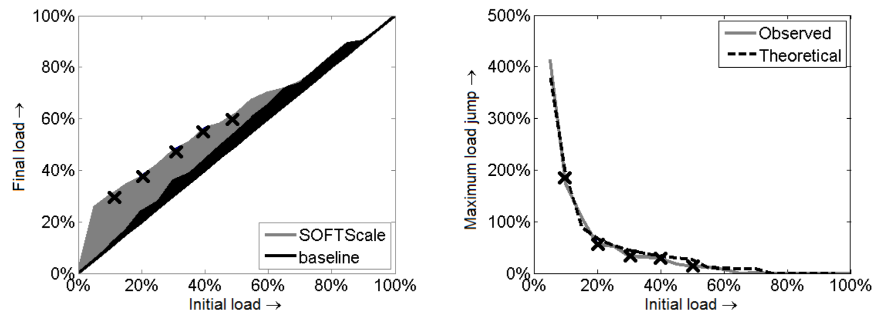
Figure 7(b) plots $P_{avg}$, the average power consumed by the application servers and the memcached servers, for SOFTScale and the baseline. We see that SOFTScale does not consume any additional power as compared to baseline. This is because the total amount of work done by all servers under SOFTScale and under baseline is about the same, for a given load level. Thus, $P_{avg}$, which is a proxy for operational costs, does not change significantly when using SOFTScale.

Figure 8 shows the full set of results for SOFTScale. In Figure 8(a), the gray region shows the solution space, or regimes, of load jumps that SOFTScale can handle without violating the 500ms SLA, while the black region shows the load jumps that the baseline can handle without violating the SLA. Note that SOFTScale's solution space is a superset of the baseline's solution space. The crosses in the figure refer to the specific load jump cases we showed in Figure 7, namely the maximum load jumps that SOFTScale can handle for each of the initial loads.

Since the system is optimally provisioned (see Section 2), the baseline cannot handle any significant load jumps. In particular, when the initial load is either too low or too high, the baseline cannot handle any load jumps. However, because of the inherent elasticity in the system, the baseline can handle some small load jumps when the initial load is moderate. For example, when the initial load is 20%, the black region indicates that baseline can handle a maximum jump of $20\% \rightarrow 24\%$.

By contrast, SOFTScale can handle a much larger range of load jumps as compared to the baseline. For example, when the initial load is 20%, the gray region indicates that SOFTScale can handle a maximum jump of $20\% \rightarrow 35\%$.

In Figure 8(b), we plot the maximum load jump (in %) that SOFTScale can handle for each initial load using the solid gray line. Again, the crosses in the figure refer to the specific load jump cases we showed in Figure 7. For example, the first cross from the left corresponds to



(a) Solution space for $t_{setup} = 5$ minutes. (b) Improvement for $t_{setup} = 5$ minutes.

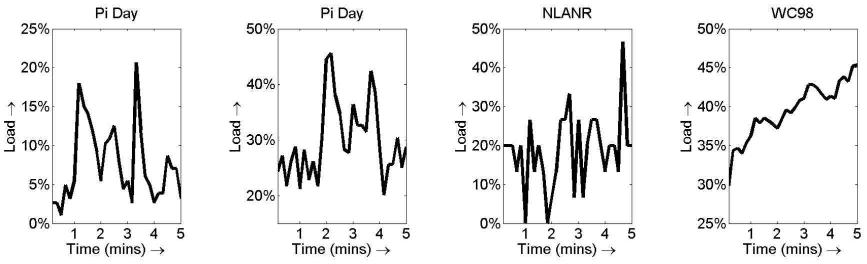
Figure 8: *Full range of results for SOFTScale. The crosses in the figures refer to the specific load jump cases shown in Figure 7. Note that SOFTScale's solution space in Figure (a) is a superset of the baseline's solution space.*

the $10\% \rightarrow 29\%$ load jump, which amounts to a $190\%$ jump in load. The dashed line shows our estimates for the maximum load jump that SOFTScale can handle, given by Equation (1) (with a few extra $\%$ due to the elasticity in the system). We see that our estimates match our implementation results. As expected, Figure 8(b) shows that SOFTScale can handle huge jumps when the initial load is low, but can only handle moderate load jumps when the initial load is high.

## 4.2 Spikes in real-world traces

In addition to evaluating SOFTScale under instantaneous load jumps (as in Section 4.1), we also evaluate SOFTScale under the real-world traces, Pi Day [6], NLANR [1] and WC98 [7], shown in Figure 9. We re-scale each trace so that the peak load corresponds to 750 req/s, and then consider five minute ($t_{setup}$) snippets that highlight load spikes. The load numbers in Figure 9 correspond to the post-scaled traces. We assume the system is well provisioned at time $t = 0$, and then examine the system performance for the next five minutes, during which additional capacity is being brought online.

Although the initial load ranges from 5% to 30% across the different traces, SOFTScale achieves a $T_{95}$ of less than 500ms for all cases (see Figures 9(a) to 9(d)). By contrast, the baseline results in a $T_{95}$ of over 115s in Figure 9(a), where the load quadruples from 5% to $20\%$. In Figure 9(b), where the load roughly doubles from 25% to 46%, the $T_{95}$ under the baseline is just over a second, in contrast to SOFTScale's 470ms. The superiority of SOFTScale over the baseline for the trace in Figure 9(b) is further illustrated in Figure 10, which depicts the instantaneous $T_{95}$ (collected every second) over the trace.



(a) Baseline: $T_{95} = 115,730ms.$ SOFTScale: $T_{95} = 418ms.$ (b) Baseline: $T_{95} = 1,050ms.$ SOFTScale: $T_{95} = 470ms.$ (c) Baseline: $T_{95} = 3,477ms.$ SOFTScale: $T_{95} = 439ms.$ (d) Baseline: $T_{95} = 620ms.$ SOFTScale: $T_{95} = 474ms.$

Figure 9: *Real-world trace snippets used for our experiments.*

## 4.3 Spikes created by server faults

Thus far, we considered the case where load spikes are caused by a sudden increase in request rate. However, load spikes can also result because of a sudden drop in capacity. Service outages [31] and server failures [33] are common causes for a sudden (and unpredictable) drop in capacity. SOFTScale is useful regardless of the cause of load spikes since SOFTScale is invoked when the number of jobs at a server increases (see Section 3.1). We now illustrate the fault-tolerance benefits of SOFTScale.

Consider a system that is well provisioned to handle 30% initial load. Suppose a failure takes down half of the provisioned capacity, resulting in a system that can now only handle 15% load. We refer to this as a $30\% \rightarrow 15\%$ capacity drop. Figure 11(a) shows our experimental results for instantaneous $T_{95}$ (collected every second) under a $30\% \rightarrow 15\%$ capacity drop, which is triggered at the 10s mark. Apache's load balancer is very quick to recognize that some of the application servers are offline, and thus stops sending additional requests to them. In Figure 11(a), while SOFTScale successfully handles the capacity drop, the baseline completely falls apart. The power

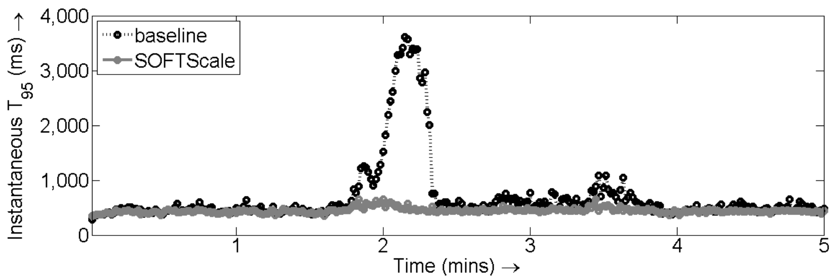
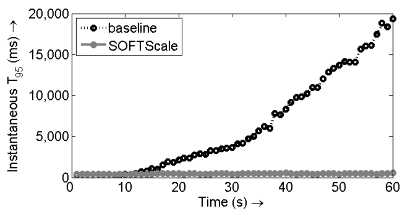
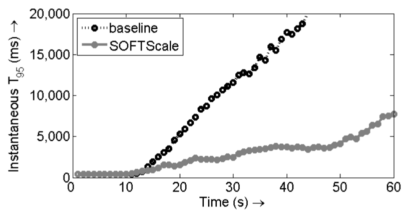


Figure 10: *The plot illustrates the superiority of SOFTScale over the baseline for the Pi Day [6] trace in Figure 9(b).*



(a) 30% → 15% capacity drop.



(b) 50% → 20% capacity drop.

Figure 11: *SOFTScale provides significant benefits even when load spikes are caused by a sudden drop in capacity. In the figures above, we drop capacity at the 10s mark.*

consumption for SOFTScale and the baseline are about the same, and are thus omitted due to lack of space.

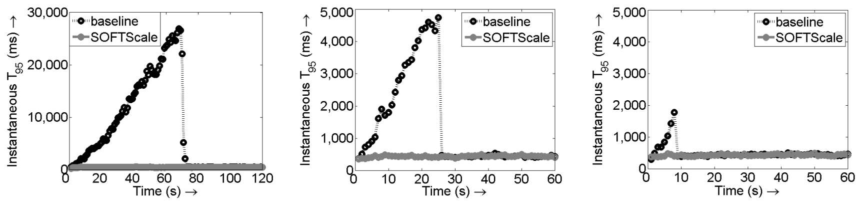
Figure 11(b) shows our experimental results for instantaneous $T_{95}$ under a very severe $50\% \rightarrow 20\%$ capacity drop, which is produced by taking down 6 of the 10 application servers at the 10s mark. This time, we see that instantaneous $T_{95}$ rises sharply for both SOFTScale and the baseline. However, the rate at which instantaneous $T_{95}$ increases under SOFTScale is significantly lower than that under the baseline. Thus, we conclude that SOFTScale is useful even when load spikes are caused by a sudden drop in capacity.

## 5 Lower setup times

While production servers today are only equipped with "off" states that necessitate a huge setup time ($t_{setup} = 5$ minutes 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 analyze SOFTScale for the case of lower setup times by tweaking our experimental testbed as discussed in Section 2. Intuitively, for low setup times, one might expect that SOFTScale is not needed since instantaneous $T_{95}$ should not rise too much during the setup time. This turns out to be false.

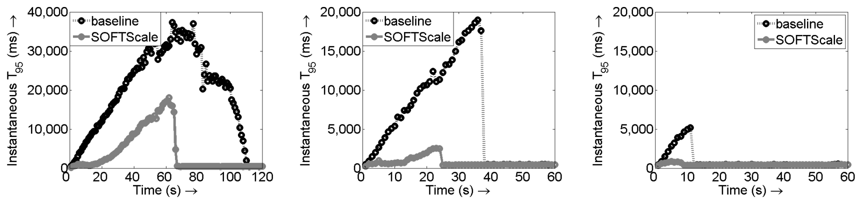
Figure 12 shows our experimental results for instantaneous $T_{95}$ under the $15\% \rightarrow 30\%$ load jump, for a range of $t_{setup}$ values. We change the scale for Figure 12(a) to fully capture the effect of the 50s setup time. Recall from Figure 8(a) that SOFTScale can handle the $15\% \rightarrow 30\%$ load jump, even if $t_{setup} = 5$ minutes. Thus, it is not surprising that SOFTScale can handle the $15\% \rightarrow 30\%$ load jump for $t_{setup} =$ 50s, 20s and 5s in Figure 12.

By contrast, the instantaneous $T_{95}$ for the baseline quickly grows and exceeds the 500ms SLA during the entire setup time duration, even for the $t_{setup} = 5s$ case. However, the instantaneous $T_{95}$ values for the baseline are not too high under lower setup times. This is because *when the*



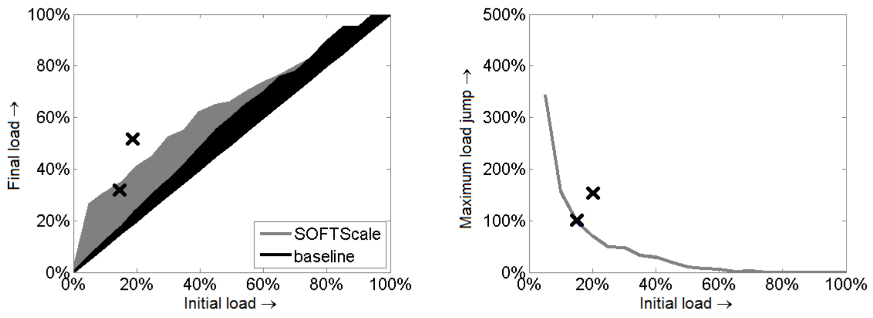
(a) $t_{setup} = 50s$. (b) $t_{setup} = 20s$. (c) $t_{setup} = 5s$.

Figure 12: *Effect of $t_{setup}$ on instantaneous $T_{95}$ for a 15% → 30% jump in load.*



(a) $t_{setup} = 50s$. (b) $t_{setup} = 20s$. (c) $t_{setup} = 5s$.

Figure 13: *Effect of $t_{setup}$ on instantaneous $T_{95}$ for a 20% → 50% jump in load.*



(a) Solution space for $t_{setup} = 20s$. (b) Improvement for $t_{setup} = 20s$.

Figure 14: *Full range of results for SOFTScale under $t_{setup} = 20s$. The crosses in the figures refer to the specific load jump cases shown in Figures 12 (*$15\% \rightarrow 30\%$ *load jump) and 13 (*$20\% \rightarrow 50\%$ *load jump).*

*setup time is low, the overload period is very short.* Observe that instantaneous $T_{95}$ does not drop immediately after the setup time because of the backlog created during the setup time.

Figure 14 shows the full set of results for SOFTScale for the case of $t_{setup} = 20s$. In Figure 14(a), we show the solution space of load jumps that SOFTScale and the baseline can handle without violating the 500ms $T_{95}$ SLA (over the 20s setup time). The crosses in the figure refer to the specific load jump cases we showed in Figures 12 and 13. We see that SOFTScale can handle a much larger range of load jumps (gray region) as compared to the baseline (black region), just

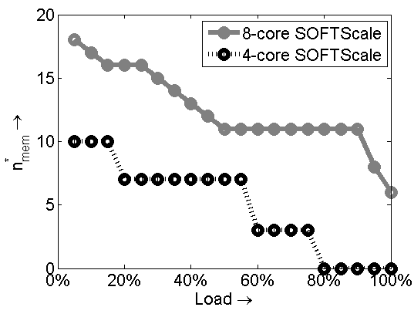


Figure 15: *Using 8-core memcached servers significantly enhances SOFTScale's ability to handle load jumps.*

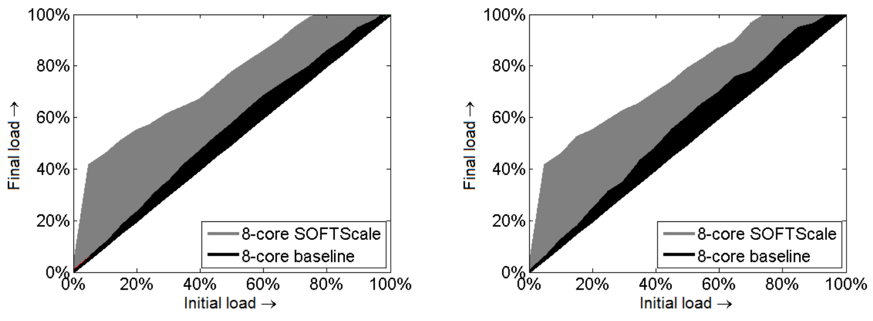
as we observed in Figure 8(a) for $t_{setup} = 5$ minutes. In Figure 14(b), we plot the maximum load jump (in $\%$) that SOFTScale can handle for each initial load. Again, as expected, SOFTScale can handle huge jumps when the initial load is low, but can only handle moderate load jumps when the initial load is high.

It is very interesting to note that the performance degradation caused by load spikes for the baseline case does not go away even when the setup time is really low. Thus, *there is a need for SOFTScale even under low setup times*. Comparing Figures 8 and 14, we see that the range of load jumps that the baseline (and SOFTScale) can handle increases only slightly under the much lower setup time of 20s. The reason that this increase is so small is that most of the "damage" to $T_{95}$ has already occurred after only a few seconds.

## 6 Future architectures

In our implementation testbed (see Section 2), we use 4-core servers for the memcached tier. In the near future, it is likely that 4-core processors will be replaced by 8 (or more) core processors, even though their memory capacity is unlikely to increase significantly [20]. Thus, we would still need just as many memcached servers. On the other hand, data replication needs may require additional memcached servers. In either case, the memcached tier will now have more spare compute capacity that can be exploited by the application tier via SOFTScale. In this section, we investigate the performance of SOFTScale for the case where we have 8-core memcached servers. Figure 15 shows $n^*_{mem}$, the optimal number of application requests that a memcached server can handle simultaneously without violating the 500ms SLA, for 8-core and 4-core memcached servers. We see that using 8-cores allows us to put a lot more application work on the memcached servers. Thus, SOFTScale should be able to handle much higher load jumps with 8-core memcached servers.

Figure 16(a) shows the full set of results for SOFTScale and the baseline, both with 8-core memcached servers, for the case of $t_{setup} = 5$ minutes. We see that SOFTScale with 8-core memcached servers can handle a significantly larger range of load jumps. For example, SOFTScale



(a) Solution space for $t_{setup} = 5$ minutes. (b) Solution space for $t_{setup} = 20s$.

Figure 16: *Full range of results for SOFTScale with 8-core memcached servers under (a)* $t_{setup} = 5$ *minutes and (b)* $t_{setup} = 20s$*. We see that 8-core memcached servers provide huge benefits for SOFTScale regardless of the setup time.*

can handle a $10\% \rightarrow 50\%$ load jump as compared to the maximum jump of $10\% \rightarrow 29\%$ using 4-core memcached servers, as was shown in Figure 8(a). Further, SOFTScale can now handle load jumps even when the load is as high as 80%, since the memcached work requires at most 4 cores at peak load (see Section 2), still leaving 4 cores at each memcached server for application work. We also estimated the maximum load jump that SOFTScale can handle via Equation (1), and found that our estimates match our implementation results. Figure 16(b) shows the full set of results for SOFTScale for the case of $t_{setup} = 20s$. These results are very similar to those in Figure 16(a). Thus, even though there is a cost (monetary cost and increased power consumption) involved in switching to 8-core memcached servers, it might make sense to deploy these servers for the memcached tier to handle severe load spikes using SOFTScale.

## 7 Prior work

There is a lot of prior work that deals with dynamic capacity management. These works can be classified into reactive [25, 28, 32], predictive [22, 18] and mixed [36, 17, 16] approaches. While these approaches can handle gradual changes in load, they cannot handle abrupt changes, especially load spikes that occur almost instantaneously. This claim was also verified by other authors [10].

There has been some prior work specifically dealing with load spikes [10, 23]. Chandra et al. [10] show that existing dynamic capacity management algorithms are not good at handling flash crowds in an internet data center. In order to handle flash crowds, the authors advocate either having spare servers that are always available (over-provisioning), or finding a way to lower setup times. However, as our work shows (see Figures 12(c) and 13(c)), even a 5s setup time can result in severe SLA violations. Further, by using SOFTScale, we do not have to pay for any additional resources, which is not the case when over-provisioning via spare servers. Lassettre et al. [23] propose a short-term forecasting approach to handle load spikes for a multi-tier system with a setup time of 30s. While [23] is very effective at handling load spikes that gradually build over time, it is not well suited for the instantaneous load spikes we consider in this paper since the forecasting in [23] itself requires at least 10s, and we have shown that even a 5s setup time is detrimental. Observe that SOFTScale is actually complementary to the above approaches, and can be used in conjunction with them.

There has also been recent work looking at data spikes, where a particular web object becomes extremely popular. Data spikes can be handled by caching or replication techniques (see, for example [35]), and are not the focus of our paper.

To handle load spikes for small websites with only static content, a possible solution is to host their content on a cloud computing platform. These platforms are able to handle load spikes by over-provisioning more economically since they host multiple websites, and load spikes on individual websites are often not correlated (statistical multiplexing) [15]. For multi-tiered cloud computing environments, SOFTScale can be used in conjunction with statistical multiplexing.

Finally, there is also a lot of prior work [37, 4, 38, 11] that deals with managing overload conditions by allowing for performance degradation. Some of the popular techniques that have been used to regulate performance degradation include admission control and request prioritization. By contrast, SOFTScale handles load spikes without any performance degradation, provided the load is not high. If the load is high, SOFTScale can be coupled with techniques like those in [37, 4, 38, 11] to minimize the damage caused by load spikes.

## 8 Conclusion

In this paper, we consider load spikes, which are all too common in today's data centers [12, 30, 24, 19, 7, 31, 33]. Our results in Figures 12 and 13 show that ignoring load spikes can result in severe SLA violations, even if it takes only 5 seconds of setup time to bring capacity online. The obvious solution of over-provisioning resources is quite expensive since load spikes are often unpredictable.

We propose SOFTScale, an approach to handling load spikes in multi-tier data centers without consuming any extra resources. In multi-tier data centers, the application tier is typically stateless, and can be dynamically provisioned, whereas the data tier is stateful, and is always left on. SOFTScale works by opportunistically stealing resources from the data tier to alleviate the overload at the application tier during the setup time needed to bring additional application tier capacity online. Since tiers in a data center are typically carefully provisioned for peak load, SOFTScale must steal from the data tier without hurting overall performance. SOFTScale does this by first determining how much spare capacity can be stolen from the data tier without violating SLAs at different load levels, and then dynamically isolating the application work and the data delivery work at the data tier to avoid interference.

Our implementation results on a 28-server testbed demonstrate that SOFTScale can handle various load spikes for a range of setup times (see Figures 8 and 14). Specifically, SOFTScale can handle instantaneous load jumps ranging from $5\% \rightarrow 25\%$ to $50\% \rightarrow 61\%$, even when the setup time is 5 minutes. SOFTScale works extremely well for real-world load spikes (see Figure 9), and significantly improves performance (typically a 2X – 100X factor improvement) when compared to the baseline. Even more benefits are possible for future many-core servers (see Figure 16).

While our implementation testbed mimics a web site of the type seen in Facebook or Amazon with an application tier and a memcached tier, we believe SOFTScale will also be applicable when the memcached tier is replaced by any other data tier. Since the data tier is stateful, there will always be a subset of servers that will not be turned off. Thus, SOFTScale can leverage these servers to alleviate the bottleneck at the application tier during load spikes.

## References

[1] National Laboratory for Applied Network Research. Anonymized access logs. ftp://ftp.ircache.net/Traces/.

[2] Personal communication with Facebook.

[3] Stephen Adler. The Slashdot Effect: An Analysis of Three Internet Publications. http://ssadler.phy.bnl.gov/adler/SDE/SlashDotEffect.html.

[4] Atul Adya, William J. Bolosky, Ronnie Chaiken, John R. Douceur, Jon Howell, and Jacob Lorch. Load management in a large-scale decentralized file system. *MSR-TR*, 2004-60, July 2004.

[5] Amazon Inc. Amazon elastic compute cloud (Amazon EC2). http://aws.amazon.com/ec2/, 2008.

[6] David G. Andersen. Trace of web site activity on Pi day (3/14/2011)from domains hosted by angio.net. Personal communication, December 2011.

[7] Martin Arlitt and Tai Jin. Workload characterization of the 1998 world cup web site. *IEEE Network*, 1999.

[8] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Sigmetrics 2012*, London, UK.

[9] Roy Bryant, Alexey Tumanov, Olga Irzak, Adin Scannell, Kaustubh Joshi, Matti Hiltunen, Andres Lagar-Cavilla, and Eyal de Lara. Kaleidoscope: cloud micro-elasticity via vm state coloring. In *EuroSys 2011*, Salzburg, Austria.

[10] Abhishek Chandra and Prashant Shenoy. Effectiveness of dynamic resource allocation for handling internet flash crowds. Technical Report TR03-37, Department of Computer Science, University of Massachusetts at Amherst, November 2003.

[11] Ludmila Cherkasova and Peter Phaal. Session-based admission control: A mechanism for peak load management of commercial web sites. *IEEE Trans. Comput.*, 51, June 2002.

[12] Josh Constine. Walmarts black friday disaster: Website crippled, violence in stores. http://techcrunch.com/2011/11/25/walmart-black-friday, November 2011.

[13] 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 *SOSP 2007*, Stevenson, WA, USA.

[14] Wayne W. Eckerson. Three tier client/server architecture: Achieving scalability, performance, and efficiency in client server applications. *Open Information Systems*, 10, January 1995.

[15] Jeremy Elson and Jon Howell. Handling flash crowds from your garage. In *USENIX ATC 2008*, Boston, MA, USA.

[16] Anshul Gandhi, Yuan Chen, Daniel Gmach, Martin Arlitt, and Manish Marwah. Minimizing data center sla violations and power consumption via hybrid resource provisioning. In *IGCC 2011*, Orlando, FL, USA.

[17] 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.

[18] Tibor Horvath and Kevin Skadron. Multi-mode energy management for multi-tier server clusters. In *PACT 2008*, Toronto, ON, Canada.

[19] Jim Hu and Greg Sandoval. Web acts as hub for info on attacks. *CNET news*, September 2001.

[20] Yoongu Kim, Vivek Seshadri, Donghyuk Lee, Jamie Liu, and Onur Mutlu. A case for exploiting subarray-level parallelism (salp) in dram. In *ISCA 2012*, Portland, OR, USA.

[21] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux virtual machine monitor. In *Linux Symposium 2007*, Ottawa, ON, Canada.

[22] Andrew Krioukov, Prashanth Mohan, Sara Alspaugh, Laura Keys, David Culler, and Randy Katz. Napsac: Design and implementation of a power-proportional web cluster. In *Green Networking 2010*, New Delhi, India.

[23] Ed Lassettre, David W. Coleman, Yixin Diao, Steve Froehlich, Joseph L. Hellerstein, Lawrence S. Hsiung, Todd W. Mummert, Mukund Raghavachari, Geoffrey Parker, Lance Russell, Maheswaran Surendra, Veronica Tseng, Noshir Wadia, and Pery Ye. Dynamic surge protection: An approach to handling unexpected workload surges with resource actions that have lead times. In *DSOM 2003*, Heidelberg, Germany.

[24] William LeFebvre. CNN.com: Facing A World Crisis. *Invited Talk, USENIX ATC 2002*.

[25] 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 *ICAC 2010*, Washington, DC, USA.

[26] Mark LaPedus. Facebook Wants New and Cheaper Memories. http://semimd.com/blog/2011/11/08/facebook-wants-new-and-cheaper-memories, November 20011.

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

[28] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: Managing performance interference effects for qos-aware clouds. In *EuroSys 2010*, Paris, France.

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

[30] Kathleen Ohlson. Victorias secret knows ads, not the web. *Computer World*, February 1999.

[31] Peter Pachal. Amazon apologizes for cloud outage, issues credit to customers. *PCMag*, April 2011.

[32] Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated control of multiple virtualized resources. In *EuroSys 2009*, Nuremberg, Germany.

[33] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: a large-scale field study. In *SIGMETRICS 2009*, Seattle, WA, USA.

[34] George Schussel. Client/server: Past, present and future. `http://www.dciexpo.com/geos/dbsejava.htm`, September 2006.

[35] Beth Trushkowsky, Peter Bodík, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. The scads director: scaling a distributed storage system under stringent performance requirements. In *FAST 2011*, San Jose, CA, USA.

[36] Bhuvan Urgaonkar and Abhishek Chandra. Dynamic provisioning of multi-tier internet applications. In *ICAC 2005*, Washington, DC, USA.

[37] Bhuvan Urgaonkar and Prashant Shenoy. Cataclysm: Scalable overload policing for internet applications. *Journal of Network and Computer Applications*, 31(4):891 – 920, 2008.

[38] Thiemo Voigt, Renu Tewari, Douglas Freimuth, and Ashish Mehra. Kernel mechanisms for service differentiation in overloaded web servers. In *USENIX ATC 2001*, Boston, MA, USA.

[39] Lisa A. Wald and Stan Schwarz. The 1999 southern california seismic network bulletin. *Seismological Research Letters*, 71:401–422, July 2000.