Design and Implementation of a Power-Aware Load Balancer

Ram Raghunathan
Advised By: Professor Mor Harchol-Balter, Anshul Gandhi

Abstract

Energy costs for data centers are doubling every five years and have already crossed \$19 billion. However, much of this power is wasted as servers are mostly idle. Idle servers can also consume as much as 60% of peak power consumption.

We introduce a power management algorithm called `AutoScale` which reduces power consumption by over 30% while delivering response times that are only slightly longer, and still meet service level agreements. `AutoScale` works by dynamically provisioning data center capacity as needed. `AutoScale` is load-oblivious and can also be deployed as a distributed application. It is also computationally cheap.

We evaluate `AutoScale` in a testbed structured as a multi-tier data center. A new benchmark is developed to test jobs which work by performing key-value requests upon a data storage. Emphasis is given on testing by implementation rather than by simulation and on comparison against existing power management techniques.

1 Introduction

Data centers have now become a large part of today's IT infrastructure. Government institutions, hospitals, financial firms as well as technology firms like Hewlett-Packard, IBM, Amazon, and Google all use data centers to perform or aid their activities. Nowadays, a typical data center is structured as shown in Figure 1.

In this structure, we have:

1. Front-end proxy servers
2. Application servers
3. Back-end Cache servers
4. Back-end Database servers

The typical request flow is as follows: First, a request to the data center is routed to one of the Front-end proxy servers. This server's duty is to simply route the request onto one of the Application servers. The application server does the actual job of serving the request. In the past, any persistent data was stored on the back-end database servers. However, accessing the database is slow. Hence, many data centers employ a tier of cache servers lying in between the application and database servers. These cache servers are used to cache data for faster access later. In this way, most requests use the

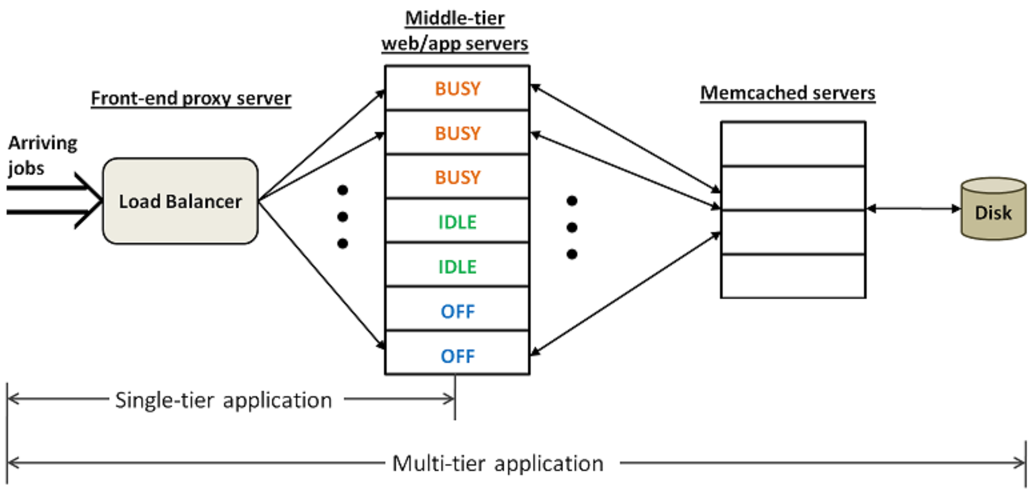


Figure 1: Structure of a Data Center

cache and only a small fraction of the requests will incur the response time increase due to accessing the database.

Popular uses of data centers is in social networking e.g. Facebook, and in e-commerce and banking e.g. Amazon and EBay. The data centers for these services typically follow a key-value storage system for data [2, 3]. This is true for data both on the cache as well as in the database. When a request arrives, the application server looks up the corresponding value for a key from the cache. If the cache fails, it looks it up from the database. Depending on the request, there could be many value requests, and even value requests depending on the value returned from a previous value request [2]. Hence, each request essentially generates a value request tree.

These value request trees inherently change the bottleneck of the request. Notice that each value request has both an I/O component as well as a processing component. Requesting a value is I/O bound and interpreting the value is CPU-bound. Hence, a value tree is has large breadth but low depth will be CPU-bound whereas a tree that has large depth but low breadth will be I/O-bound. In addition, the I/O component can include reads or writes to the cache or database, or a combination of these. Despite the increasing prevalence of this value tree based workload in data centers, no benchmark exists to test out the performance of the data center in this workload case. We introduce a simple benchmark that is highly tunable in 4.3 that effectively simulates the key-value paradigm.

Data centers are designed to meet certain service level agreements (SLA's). Many times, these SLA's are specified as percentile request time guarantees e.g. 95% of requests must be served in at most 200 milliseconds. However, due to rising costs of running a data center, energy efficiency has recently been considered as a parameter for evaluating data centers. Indeed, more money is spent on running and cooling a server than it is on purchasing it.

Data centers consume a lot of energy while running. Energy costs are doubling every five years, and have already increased to $19 billion [1]. Much of this energy consumed is wasted. Servers tend to be busy for only 10% to 30% of the time [4, 5]. However, an idle server can consume up to 60% of the power that a busy server does.

Since load varies over time, data centers are often provisioned for peak load to meet the SLA's. Hence, data centers are grossly over-provisioned for much of the time, since peak load occurs for only a fraction of the time. The solution seems to be to simply turn servers off when not needed and turn them back on when they are. However, this situation only works in an ideal theoretical setup. Many servers have large setup times before which they can be productive. For example, the servers used in our experiments took 200 seconds from the power on command to be ready to accept requests. This overhead is enough to cause SLA violations, hence making this technique inviable in real data centers.

There are many ways of provisioning data centers so as to reduce power. The one employed now is to simply leave all servers on, whereby no power is saved. While this guarantees the best response times, it also wastes power excessively. We call this policy `AlwaysOn`. There has also been extensive research into more aggressive provisioning algorithms. These algorithms can be divided into reactive and predictive approaches. However, most of these algorithms have only been studied via simulation, and not on a data center.

Reactive algorithms typically have thresholds of parameters to determine when to turn a server on or off. For example, they may have an upper and lower limit response time, power usage, or a combination of both. This is an optimal solution in an ideal environment of zero setup time. However, in a real environment, where setup time is significant, this method may perform badly. This is because it starts to fix the problem only when it is detected, and the effects of trying to fix the problem are only seen once the server is set up.

Due to the inadequate performance of reactive approaches in non-ideal environments, research has turned to predictive algorithms (for example, see [6, 7]). These methods seek to predict future load and pro-actively provision the data center for the forecasted load. These predictions typically use data from a moving window of past information. This is especially true of techniques like Moving Window Average (referred to as the `MWA` algorithm) and Linear Regression (referred to as the `LR` algorithm).

Predictive algorithms tend to be effective in periodic or seasonal load, which varies slowly. However, we doubt their effectiveness for bursty load patterns. Some papers e.g. [7] have shown significant improvements by predictive approaches, but these were always compared against a weak version of `AlwaysOn` where twice the peak number of servers are kept on at all times. While this tends to be industry standard, it is to be expected that aggressive data center provisioning algorithms such as the aforementioned predictive approaches perform well against it.

One of the major problems of predictive approaches is that it tries to estimate load far in the future (typically for a server's setup time) using limited past data. Clearly, not all of past data can be used as that is not representative of current load conditions. However, less data leads to less accurate forecasts of future conditions. If predicted load is wrong, predictive approaches can struggle to cope. Predictive approaches tend to be designed with minimal tolerance for deviations from predictions. This leads to poor responses when reality does not meet expectations. Finally most predictive approaches have large overheads in computation time.

A subtle problem with previous research is that their load balancing algorithms seek to balance load across all servers evenly using a load balancing algorithm like Round-Robin or Join-Shortest-Queue. While this is effective from a performance perspective, it is clear that it is the most effective when the goal is to meet performance goals *and* reduce power consumption. For more information about prior work, please refer to section 2.

Our approach to data center provisioning, `AutoScale` is very different. It does not try to predict future load, and is very simple. It can also be implemented as a distributed solution, leading to efficient deployment. `AutoScale` consists of two components, the provisioning algorithm and the load balancing algorithm. The provisioning algorithm is trivial. Each server can independently decide when it is to turn off based on certain "idle-timers" which are timers deciding how long a server is permitted to remain idle before being turned off. These timers inherently build tolerance into the algorithm for large changes in demand in short periods of time. The load balancing algorithm tries to saturate a server to a preset number of jobs before allowing load to be routed to another server. Hence, load is sent to the minimal number of servers necessary. This allows more servers to be turned off, lowering power usage.

In our tests, we used a data center setup in same way as shown in Figure 1. We use the data center provisioning algorithms only on the application-tier servers. Our benchmark consists of value request tree. We compare `AutoScale` against `AlwaysOn`, `Reactive`, `MWA`, and `LR`. For more information about the experimental structure, please refer to Section 4. For more information about the algorithms that `AutoScale` is compared against, please refer to 5

We find that `AutoScale` performs well against the other algorithms. Response time is only about 20% more than for `AlwaysOn` while power consumption is less. In contrast, the other algorithms have extremely long response times, although they do use less power. However, meeting SLA's takes priority over energy consumption, so the lower power is meaningless.

2 Prior Work

There are many methods of reducing power consumption in a data center. For example, we can optimize the server architecture (see for example [8, 9]), use frequency scaling and dynamic voltage adjustments for processors (see for example [10, 11]), or use virtualization (see for example [12, 13]). We limit my discussion of prior work to those methods that dynamically provision data center capacity i.e. turn servers on and off.

There are typically two approaches to dynamic provisioning algorithm. The first is the reactive approach, also known as the control-theoretic approach. For example, both Horvath et al [15] and Wang et al [14] use feedback mechanisms to control the power-performance tradeoff in multi-tier systems. The authors used Dynamic Frequency and Voltage Scaling in addition to data center provisioning to achieve their results. Reactive approaches work well in ideal situations where the setup time for a server is zero. However, they do not work well in reality. By definition, reactive approaches only attempt to solve a problem once it occurs. However, due to setup time, the attempt to fix the problem only shows after the server is setup. While it is being setup, it is entirely possible that the server becomes unneeded, or SLA violations occur. This is not acceptable for a commercial data center.

The second approach to data center provisioning is predictive. Krioukov et al [7] used various predictive methods such as Last Arrival, Moving Window Average (referred to as the `MWA` algorithm), Exponentially Weighted Average, and Linear Regression (referred to as the `LR` algorithm). These methods were used to predict future load by a server's setup time, thereby turning on servers now to be ready in time for future load. The authors considered Wikipedia.org traffic and found that `MWA` and `LR` did best. Using simulation, the authors found that `MWA` and `LR` provided significant power advantages over `AlwaysOn`. However, the authors used a version of `AlwaysOn` that does not know the peak arrival rate ahead of time. Hence, their `AlwaysOn` algorithm is weaker than ours where exactly as many servers are provisioned in `AlwaysOn` as necessary, and no more. Chen et al [6] use an auto-regression method to predict future load. They then use simple thresholds to decide whether or not a server should be turned on or off. These thresholds are implemented in the load balancing algorithm much like `AutoScale` does. Each server has a preset threshold for number of active jobs and the load balancer seeks to saturate a server before routing requests to another server. However, the authors do not consider tolerance of the system, and turn off servers as soon as possible. They also test their method only on a single-tier data center architecture using simulations. The authors conclude that their method performs well on periodic load which repeats daily, say.

While predictive methods tend to perform well for periodic loads, it is uncertain whether they perform as well on non-periodic loads. We find that they are not as effective, whereas `AutoScale` is effective for both periodic and non-periodic loads. In addition, these methods tend to be computationally-intensive due to the complexity of the prediction algorithm.

3 `AutoScale`

When designing a data center provisioning algorithm, there are three questions that need to be answered:

1. When should a server be turned on?
2. When should a server be turned off?
3. Which server should a request be routed to?

3.1 When should a server be turned on?

This question's answer depends heavily on the load balancing algorithm being used. However, regardless of algorithm, a server must be turned on when the load balancing algorithm determines that the existing set of servers is insufficient to handle the existing load. We will discuss how we answer the question in subsection 3.3.

3.2 When should a server be turned off?

Prior work concentrates on using a centralized system in which a "master" will dictate when a server is to turn off (Note: by turn off, we mean complete any pending jobs and then turn off). The master will base it's decision on future prediction of load in case of a predictive algorithm.

Since a data center is typically distributed, it is awkward to see a centralized system being deployed for power management. In addition, we notice that these methods do not allow for any tolerance of unexpected conditions. Every server is either busy, off, or in setup. None are left idle. While this will work well when setup costs are low or predictions are perfect, this is not the case in reality.

To allow for a distributed design, `AutoScale` permits each server to decide when to turn itself off, simply informing the load balancer that it is no longer available. In addition to this, `AutoScale` builds tolerance into it's system by setting a time t_{wait} for which each server remains idle before turning itself off. If a request is routed to a server while it is idle, but before it has been idle for t_{wait} time, it simply accepts the request. When it becomes idle once more, it waits for t_{wait} once again.

3.3 Which server should a request be routed to?

We notice that much of prior work uses routing algorithms that are proven to optimize response times, such as Round-Robin or Join-Shortest-Queue. However, it is not apparent that these algorithms lend themselves to energy efficiency. Hence, `AutoScale` uses a routing algorithm based on the load-skewing request dispatch algorithm discussed in [6]. In our load balancing algorithm, we first preset a packing factor p. When a request is dispatched, we send it to the first server in our list that has less than p jobs pending. Hence, we tend to saturate a server before routing to other servers. This lowers the number of servers being used. More servers will turn off, and hence power consumption reduces.

The packing factor must be chosen with care. A good choice would be the maximum number of jobs that a single server can handle while still meeting SLA's. For example, from Figure 2 we notice that a 95% guarantee of 500 millisecond response time means that the we should set p to 30.

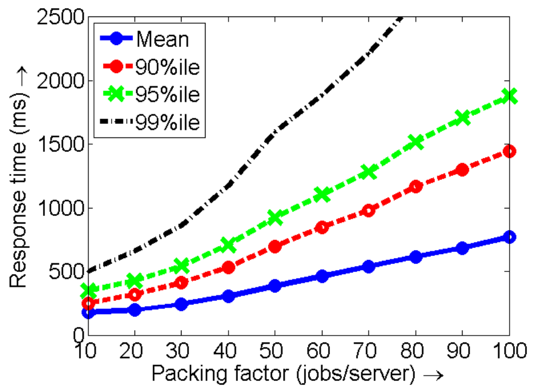


Figure 2: Response Time vs Packing Factor under `AutoScale`

This method of load balancing provably minimizes the number of servers utilized. Contrast this with Join-Shortest-Queue. In Join-Shortest-Queue, a certain number of servers N are available for load balancing, and the algorithm seeks to use each equally. This does not differentiate between servers and no server can be turned off, although each is minimally used.

As mentioned in subsection 3.1, the load balancing algorithm has the responsibility of deciding when to turn on a server. In `AutoScale` we turn on a server as soon as all available servers are saturated. While waiting for the server to turn on, load-balancing falls back onto Join-Shortest-Queue if all servers are saturated.

4 Experimental Setup

In this section, we discuss the 4-tier architecture used in our experiments. We discuss the software used and modifications made to them. We finish the section with a description of the job used as our benchmark and the different workloads we tested on.

4.1 4-tier Testbed Architecture

Our testbed is modeled after a typical data center architecture as shown in Figure 1. This architecture consists of a front-end proxy server that does request routing; application servers actually execute the request. Data is cached on the third-tier cache servers. Persistent storage is found on the back-end database servers.

Our testbed consisted of 23 machines that used Intel Xeon E5520 processors. Each machine has two quad-core processors as well 16GB of memory. We employ one of these servers as a load generator using a modified version of httperf [16]. Another server is used as the front-end load balancer. This server runs a modified version of the Apache webserver [17]. Modified software is described in more detail in subsection 4.2. 16 servers are used as the application servers running an unmodified Apache server [17]. Four servers are used as cache servers running memcached, a popular distributed cache used even in many commercial data centers [18]. Finally, 1 server is used to store the database. The database is approximately 500 GB in size, and is stored in the BerkeleyDB format [19]. We use IPMItool [20] to remotely turn servers on and off.

The servers used have the following characteristics:

- Consume 140W when idle
- Consume 200W when busy and in setup
- Consume 0W when off
- Take about 200 seconds to setup

4.2 Software Used

Load generation was done using httperf [16]. This software reports accurate statistics regarding response times for web servers. It is also capable of generating high request rates, which are necessary to mimic commercial data centers as well as to load up the testbed sufficiently. Httperf only supports a single request rate, however. It was modified to also support variable request rates, where a certain number of requests are sent at a particular request rate, and the next set of requests are sent at a different request rate.

The load balancer used was the Apache web server [17] along with it's mod_proxy_balancer extension [21]. This extension allows one to specify a set of servers to balance among. It also allows one to specify the algorithm used to perform load balancing. This extension was modified significantly. First, support for the packed load balancing algorithm described in subsection 3.3 was added to the extension. After this, full support for `AutoScale` was added. This was done by spawning a thread at the beginning of Apache's execution to monitor the application servers. Whenever the job queue size was more than the set of ready servers was able to handle, another server was turned on. When a server was idle for a predetermined time, it was shut down. The commands to turn on and off a server, as well as the `AutoScale` parameters p and t_{wait} were set as user-defined parameters, for ease of use. Finally, support was added to output statistics at a certain rate.

The application servers ran an unmodified version of the Apache [17] webserver.

The cache servers used memcached [18]. Memcached is a popular distributed cache that has a key-value paradigm. It is even used in commercial data centers.

The database server runs memcachedb [22], a simple key-value database that is based on memcached. It uses BerkeleyDB [19] as its storing mechanism.

4.3 Job Used

Many commercial data centers use a key-value paradigm for their activities. A request spawns off one or more value requests at the application servers. Depending on the responses, more value requests may occur. Hence, every request is essentially a request job tree. This job has the nice feature of requiring both I/O and CPU. In fact, the value request tree can be skewed to be either I/O-bound or CPU-bound. If the value request tree is very broad but not very deep, the request becomes CPU-bound. On the other hand, if it is very deep but not very broad, the request becomes I/O-bound. However, there does not exist any benchmark for this kind of job.

To create a key-value benchmark, we first populate the database with keys and corresponding values. The values themselves are a set of keys. When a request comes in, the application server first requests the value of a random key. The value is then parsed to obtain the set of keys. Each key is then requested again. This process can continue iteratively. We refer to the number of value requests made as the depth and the number of keys in a value request as the breadth. In this way, each request creates a value request tree. By varying the maximum depth and breadth, we can change the behavior of the value request tree. In our experiments, we set breadth to three and depth to eight. This results in a job size of approximately 3500 value requests. On a server with no resource contention and with 100% hit rate in the cache, a job takes an average of 400 milliseconds to complete.

By picking the first key out of a distribution, we can also affect the probability of a certain key being selected. This directly affects the hit rate of the cache. In our experiments, we used the Zipf distribution [23], where probability of a generating a key is inversely proportional to the power of a key.

4.4 Workload

We used realistic arrival traces to generate the workload in our experiments. We use a wide variety which have unique qualities in each. The full list is seen in Table 1. Results are discussed in conjunction with the Quickly Varying and Slowly Varying trace. However, results were collected for all traces.

5 Other Algorithms Used

We compared `AutoScale` to many other algorithms. Here, we will describe those algorithms and their behaviors.

5.1 `AlwaysOn`

`AlwaysOn` is the simplest algorithm that we compare against. It is the algorithm being used commercially, in which all servers are on.

We want to keep as many servers on as needed to meet SLA's for the peak load. For example, in Figure 3, we notice that for a server to meet a 95% response time of 400 milliseconds, it should not be allowed to handle more than 60 requests per second. Hence, if the peak arrival rate is 630 requests per second,

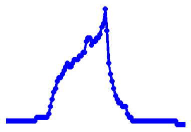
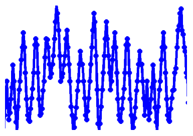
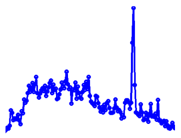
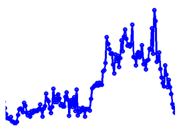
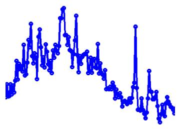
Name	Trace	Plot
Slowly Varying	ITA [24]	
Quickly Varying	Synthetic	
Quickly Varying with Big Spike	NLANR 1 [25]	
Dual Phase	NLANR 2 [25]	
Dual Phase with large Variations	NLANR 3 [25]	

Table 1: Traces Used in Experiments

we will need to provision the data center with $\lceil \frac{630}{60} \rceil = 11$ servers.

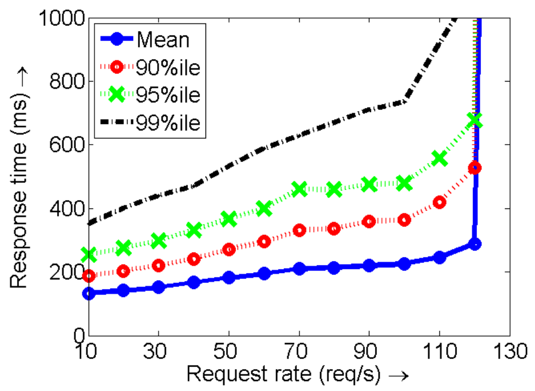


Figure 3: Response Time vs Request Rate for a single server

In reality, it is difficult to know what the peak arrival rate is going to be. As such, most implementations of `AlwaysOn` provision the data center for more than the peak request rate. This clearly wastes much power. Some papers choose to use a weak version of `AlwaysOn`. For example Krioukov et al [7] use a version of `AlwaysOn` which provisions the data center for twice the peak arrival rate. In our experiments, we empower `AlwaysOn` by knowing the peak arrival rate beforehand. Hence, our version of `AlwaysOn` will use the minimum number of servers required to meet SLA's. More importantly, this version of `AlwaysOn` is provably best of all versions.

We expect `AlwaysOn` to have high power consumption but low response times.

5.2 `Reactive`

The reactive algorithm reacts to the request rate every second. Assuming that the incoming request rate is R and the maximum request rate a server can handle to meet SLA's is r, the algorithm seeks to adjust the number of servers to $\lceil \frac{R}{r} \rceil$. Of course, the effect of turning on a server won't be noticed for 200 seconds (the setup time) and the effect of turning off a server won't be seen until the server finishes the pending jobs on it.

5.3 `MWA`

The Moving Window Average algorithm is referred to as `MWA`. It was explored in many papers and was also considered most powerful by [7]. In this algorithm, we look at a window of previous request rates for some time t, which is granular on a second level, say. To estimate the request rate for time $T+1$, where T is the current time, we average the request rates seen in the $(T-t, T]$ interval. Given this prediction for $T+1$, we can predict the request rate for $T+2$ by averaging the rates in the $((T+1)-t, T+1]$ interval. By using this iterative method, we can predict the request rate at time $T+p$ for any p. In our experiments, we set $t = 10$ and $p = 200$, which is the setup time for a server.

Given the predicted request rate for the time $T+200$, we can then make decisions about whether to turn a server on, off, or to not do anything. If the predicted request rate is R and the maximum request rate a server can handle to meet SLA's is r, we predict that we need $N = \lceil \frac{R}{r} \rceil$ servers at time $T+200$ to handle the load. If N is more than the number of servers currently ready, we turn on more servers now to make up the difference. The effect will be seen at time $T+200$.

If the number of servers required is less than currently provisioned, we look at the maximum arrival rate M predicted in the interval $(T, T+200]$. If a request rate of M requires less servers than currently provisioned, we turn some servers off. We will not need them for the next 200 seconds, assuming our prediction is true.

5.4 `LR`

In Linear Regression, we also use a window of 10 seconds as our past data. We then predict load 200 seconds in the future by linear regression.

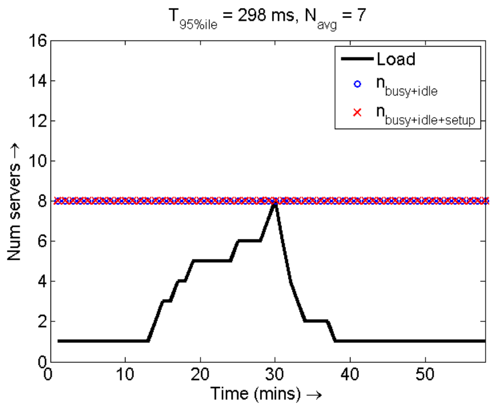
We use the same method as outlined in subsection 5.3 for turning servers on and off.

6 Results

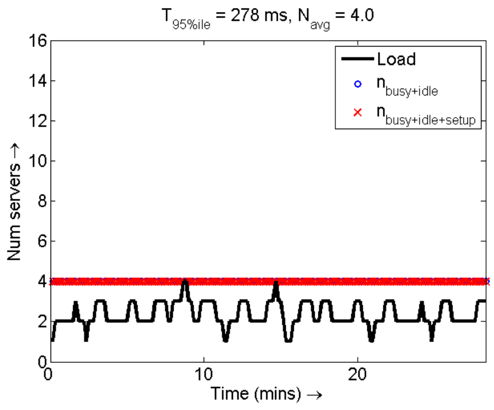
In this section we focus on results of a simplified experiment run without any interaction with the database. All requests were reads from the cache. We focus on the results of the Slowly Varying and Quickly Varying traces shown in Table 1.

6.1 `AlwaysOn`

Figure 4 plots the state of servers as a function of time, overlaid on the trace load. Obviously, no server changes state during the course of the test. We find that the Slowly Varying trace has a 95^{th} percentile response time of 298 milliseconds with an average power consumption of 1132W. The Quickly Varying trace has a 95^{th} percentile response time of 278 milliseconds with an average power consumption of 798W. As expected, response time is very low while power consumption is high.

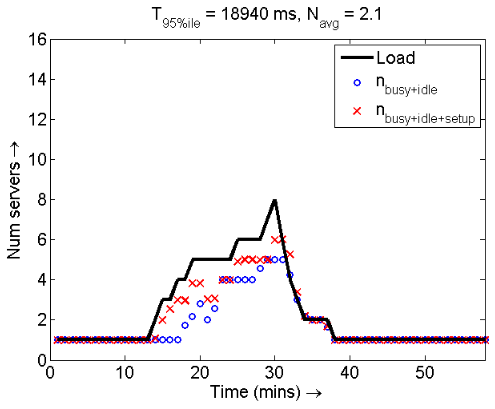


(a) Slowly Varying Trace

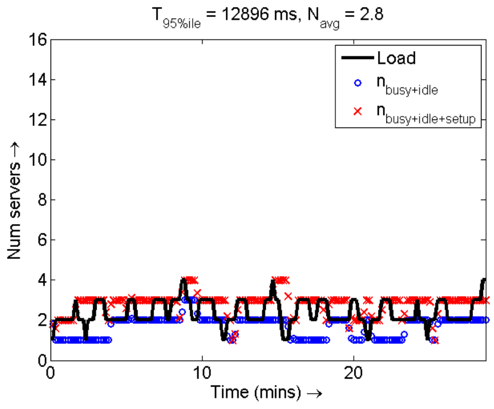


(a) Quickly Varying Trace

Figure 4: Plot of `AlwaysOn` server states

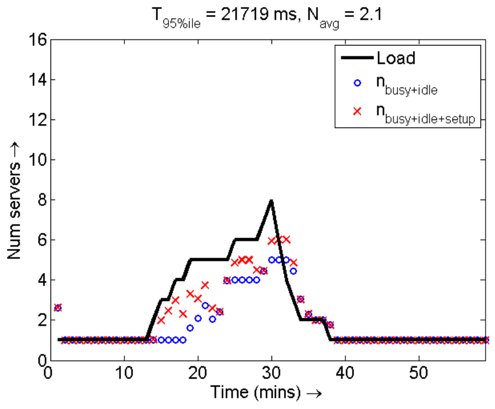


(a) Slowly Varying Trace

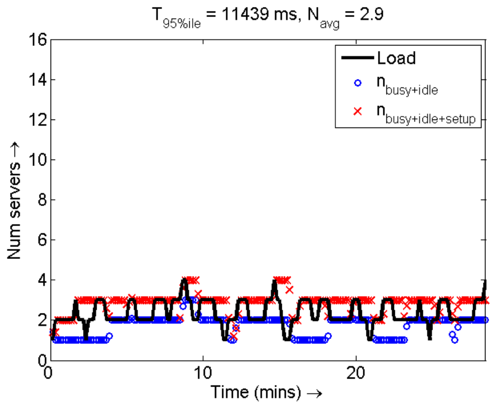


(a) Quickly Varying Trace

Figure 5: Plot of `Reactive` server states

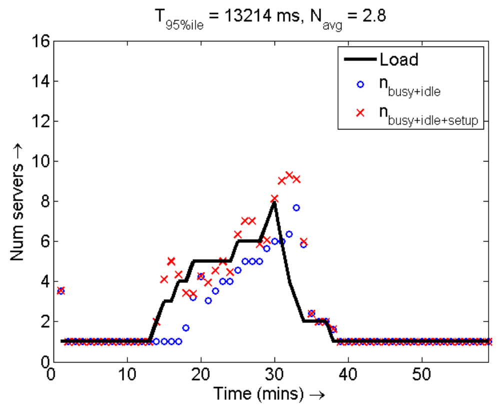


(a) Slowly Varying Trace

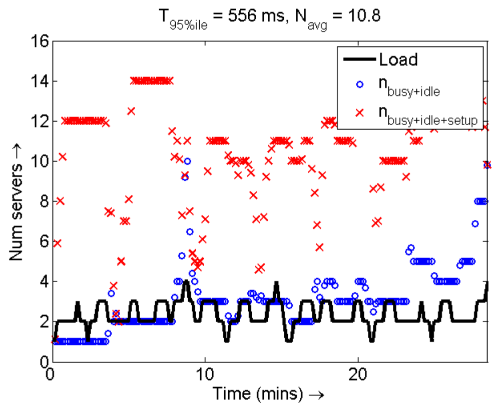


(a) Quickly Varying Trace

Figure 6: Plot of `MWA` server states

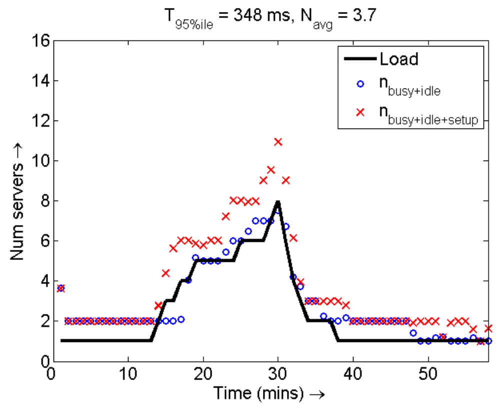


(a) Slowly Varying Trace

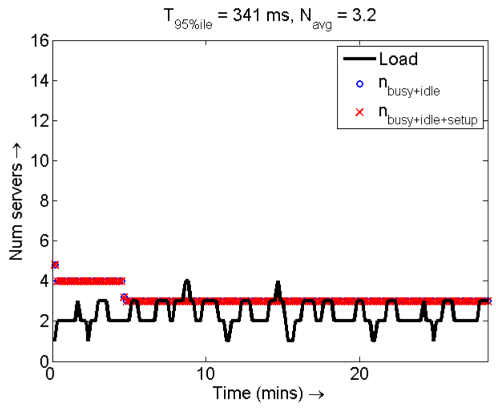


(a) Quickly Varying Trace

Figure 7: Plot of `LR` server states



(a) Slowly Varying Trace



(a) Quickly Varying Trace

Figure 8: Plot of `AutoScale` server states

6.2 `Reactive`

Figure 5 plots the state of servers as a function of time, overlaid on the trace load. We see that servers only change state as soon as a change in arrival rate occurs, and no sooner. We find that the Slowly Varying trace has a 95th percentile response time of 18940 milliseconds with an average power consumption of 414W. We also find that the Quickly Varying trace has a 95th percentile response time of 12896 milliseconds with an average power consumption of 568W. Although power consumption is relatively low as servers are only provisioned when a change occurs, the response time is extremely high, and a gross violation of SLA's.

6.3 `MWA`

Figure 6 plots the state of servers as a function of time, overlaid on the trace load. We see that servers change state only as soon as a change occurs, due to the nature of averaging. However, during a change, MWA appropriately reacts. We find that the Slowly Varying trace has a 95th percentile response time of 21719 milliseconds with an average power consumption of 414W. We also find that the Quickly Varying trace has a 95th percentile response time of 11439 milliseconds with an average power consumption of 576W. Although power consumption is relatively low as servers are only provisioned when a change occurs or is happening, due to the nature of `MWA`, the provisioning is not done fast enough. This leads to SLA violations.

6.4 `LR`

Figure 7 plots the state of servers as a function of time, overlaid on the trace load. On the Slowly Varying trace we find that servers are provisioned correctly, due to the slow variation, but too late. However, `LR` does not do well on the quickly varying trace. Due to the quick variations, predicting 200 seconds ahead results in far too aggressive predictions. We find that the Slowly Varying trace has a 95th percentile response time of 13214 milliseconds with an average power consumption of 546W. We also find that the Quickly Varying trace has a 95th percentile response time of 556 milliseconds with an average power consumption of 2163W. For the Slowly Varying trace, we find that response time is quite high, probably due to the unpredictable nature at certain areas of the load due to past data. We find that the Quickly Varying trace has a low response time, due to the large number of servers provisioned. However, power consumption is very high, even higher than `AlwaysOn`

6.5 `AutoScale`

Figure 8 plots the state of servers as a function of time, overlaid on the trace load. On the Slowly Varying trace we find that servers are provisioned accurately, with an adequate number of servers available to service the load. We also find in the Quickly Varying trace that servers are provisioned a bit aggressively, but never conservatively. We find that the Slowly Varying trace has a 95th percentile response time of 348 milliseconds with an average power consumption of 715W. We also find that the Quickly Varying trace has a 95th percentile response time of 341 milliseconds with an average power consumption of 583W. For both traces, we see low response times as well as low power consumption levels. While more power is consumed compared to the other algorithms (with the exception of `AlwaysOn`), response times are within rea-

Algorithm / Trace	AlwaysOn		Reactive		MWA		LR		AutoScale	
	T_{95}	P_{avg}	T_{95}	P_{avg}	T_{95}	P_{avg}	T_{95}	P_{avg}	T_{95}	P_{avg}
Slowly Varying	298ms	1132W	18940ms	414W	21719ms	414W	13214ms	546W	348ms	725W
Quickly Varying	278ms	798W	12896ms	568W	11349ms	576W	556ms	2163W	341ms	583W

Table 2: Comparison of All Algorithms

sonable limits, and power is still considerably low.

6.6 Discussion

The results of the test are summarized in Table 2

As we can see, `AutoScale` consistently gets the best response times, excepting `AlwaysOn`. The response time is also very comparable to `AlwaysOn`, with a 17% increase for the Slowly Varying trace and a 23% increase for the Quickly Varying trace. In addition, we find that `AutoScale` has a competitive power consumption that is approximately 36% less than `AlwaysOn`. For the Slowly Varying trace, `AutoScale`'s power consumption is more than the other algorithms. However, the other algorithms all have extremely high response times that would violate most SLA's. For the Quickly Varying trace, power consumption is comparable to the other algorithms, but once again, the other algorithms have extremely high response times. The exception to this is `LR`. However, `LR` consumes too much power, even more than `AlwaysOn`.

It is clear from these results that `AutoScale` shows promising results in balancing power and performance.

7 Future Work

`AutoScale` shows much promise in balancing the power-performance tradeoff. In the future we plan to explore using sleep states of processors instead of turning off servers to reduce setup time. However, low-power sleep states do not yet exist in desktop processors, although they do exist in mobile processors. We also seek to combine `AutoScale` with predictive approaches to better provision the data center for anticipated load. We will also see how `AutoScale` scales with data center size.

8 Conclusion

Data center costs are mounting each year, and much of these costs are directly attributable to energy requirements of data centers. One of the many ways to reduce power consumption is to dynamically provision the data center capacity to meet demand with the minimum number of servers. Compared to past research that yielded reactive and predictive approaches, `AutoScale` performs admirably. `AutoScale` is also a simple algorithm and lends itself well to the distributed nature of modern data centers. We believe that `AutoScale` effectively manages power consumption and meets SLA's all while being simple and computationally cheap.

References

[1] Greed Grid. Unused Servers Survey Results Analysis. http://www.thegreengrid.org/en/Global/Content/white-papers/UnusedServersSurveyResultsAnalysis, 2010.

[2] 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, New York, NY, USA, 2007. ACM.

[3] Paul Saab. Scaling memcached at Facebook. http://www.facebook.com/note.php?note_id=39391378919, December 2008.

[4] L. A. Barroso and U. Holzle. The case for energy-proportional computing. Computer, 40(12):33-37, 2007.

[5] Michael Armbrust, Armando Fox, Rean Grifth, 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, Feb 2009.

[6] 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 NSDI '08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation, pages 337-350, Berkeley, CA, USA, 2008. USENIX Association

[7] Andrew Krioukov, Prashanth Mohan, Sara Alspaugh, Laura Keys, David Culler, and Randy Katz. Napsac: Design and implementation of a power-proportional web cluster. In First ACM SIGCOMM Workshop on Green Networking, August 2010.

[8] David Meisner, Brian T. Gold, and Thomas F. Wenisch. Powernap: eliminating server idle power. In ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems, pages 205-216, New York, NY, USA, 2009. ACM.

[9] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: a fast array of wimpy nodes. In SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, pages 1-14, New York, NY, USA, 2009. ACM.

[10] Anshul Gandhi, Mor Harchol-Balter, Rajarshi Das, and Charles Lefurgy. Optimal power allocation in server farms. In SIGMETRICS '09: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems, pages 157-168, New York, NY, USA, 2009. ACM.

[11] Qiang Wu, Philo Juang, Margaret Martonosi, and Douglas W. Clark. Voltage and Frequency Control With Adaptive Reaction Time in Multiple-Clock-Domain Processors. In HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture, pages 178-189, Washington, DC, 2005.

[12] Xiaoying Wang, Zhihui Du, Yinong Chen, and Sanli Li. Virtualization-based autonomic resource management for multi-tier web applications in shared data center. Journal of Systems and Software, 81(9):1591 - 1608, 2008.

[13] P. Ranganathan R. Nathuji S. Kumar, V. Talwar and K. Schwan. M-channels and m-brokers: Coordinated management in virtualized systems. In Workshop on Managed Multi-Core Systems, 2008.

[14] Peijian Wang, Yong Qi, Xue Liu, Ying Chen, and Xiao Zhong. Power management in heterogeneous multi-tier web clusters. International Conference on Parallel Processing, pages 385-394, 2010.

[15] Tibor Horvath and Kevin Skadron. Multi-mode energy management for multi-tier server clusters. In PACT-08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques, 2008.

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

[17] The Apache HTTP Server Project. See http://httpd.apache.org.

[18] Brad Fitzpatrick. Distributed caching with memcached. Linux J., 2004(124):5, 2004.

[19] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley db. In Proceedings of the annual conference on USENIX Annual Technical Conference, pages 43-43, Berkeley, CA, USA, 1999. USENIX Association.

[20] IPMItool. See http:// ipmitool.sourceforge.net.

[21] Mod_Proxy_Balancer Extension for Apache. See http:// httpd.apache.org/docs/2.2/mod/ mod_proxy_balancer.html.

[22] Memcachedb. See http:// memcachedb.org.

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

[24] The internet traffic archives: WorldCup98. Available at http:/ /ita.ee.lbl.gov/html/contrib/ WorldCup.html.

[25] U.S. Environmental Protection Agency. Epa report on server and data center energy efficiency. 2007.