

Scaling Into The Cloud

Jonathan Kupferman, Jeff Silverman, Patricio Jara, Jeff Browne
 {jkupferman, jdsilverman, pjara, jbrowne}@cs.ucsb.edu

Abstract—As the use of cloud computing becomes more prevalent among popular web services, the issue arises of effectively scaling resources in the cloud. There are services that provide scaling automatically but require the use of specific platforms. We analyze three platform-agnostic algorithms for scaling resources dynamically: one developed by RightScale, and two others that predict system loads based on linear regression and autoregression of order 1. In order to effectively compare these algorithms, we developed a novel scoring metric based on availability and the standard cost model provided by cloud hosting services.

Our results show that dynamic provisioning provides marked improvements over static allocation in terms of cost with minimal drop in availability. In addition, scaling algorithms that model past usage to predict future workloads tend to respond better to sharp changes in traffic.

Index Terms—Cloud Computing, Web Services, Scalability, Availability

I. INTRODUCTION

THE advent of cloud computing has reintroduced many problems associated with distributed computing in a new context. From business models to hardware implementation strategies, companies like Amazon and Google, as well as open source efforts like EUCALYPTUS[5] and AppScale[4] are exploring this computational paradigm and its implications for both small and large-scale problems. While many of the issues associated with managing clouds have been previously explored in other distributed computing contexts, some characteristics of clouds offer new opportunities for research. This paper addresses one of the issues that is more unique to the cloud: dynamic resource allocation.

At a basic level, cloud computing provides a changeable set of virtual machine instances distributed over a cluster. A feature of such implementations is the ability to add additional instances to a group with a single API call. Since resources are not statically fixed at the beginning of a computation, applications such as web servers can adjust to workload requirements by either requesting more machine instances (scaling up), or terminating currently held instances (scaling down). In addition to this flexibility, cloud infrastructure services like Amazon’s EC2[3] provide the illusion of limitless resource allocation for most users. Consequently, the flexible and unconstrained resource pool simplify the problem of resource management to the point where it can be automated, potentially leading to efficiency as well as availability gains for hosted web services.

This paper evaluates an algorithm developed by RightScale [8], as well as our own proposed linear regression-based and auto-regression-based models in their ability to automatically provision computing resources. In order to compare algorithms, we devised a scoring metric which gauges how well

a given algorithm performed based on availability and cost. Our findings show that RightScale’s algorithm proved the least responsive of the algorithms, and it is also highly dependent on user-defined threshold values. We found that linear regression is more susceptible to small fluctuations, but can be improved with minor optimizations. Autoregression was more reactive than the RightScale algorithm and less sensitive than linear regression, but it was also more susceptible to disadvantages in certain deallocation strategies. The contributions of this paper include evaluating the viability of known techniques to dynamic resource allocation in a cloud along with a novel scoring metric to compare scaling algorithms.

II. MOTIVATION

In the traditional batch or supercomputing model, when the user wants to run her program, she submits it to a scheduler whose responsibility is determining the best order of execution for all queued jobs. To allow more effective scheduling, users also request a set of resources the batch job will require to complete. When run, all of the requested resources are allocated to the job for its entire duration. Under this model, users are forced to make the trade off between requesting more resources so the task completes quickly and making do with less in order to receive better scheduling.

The cloud model is somewhat different in that, as opposed to submitting tasks in batch, users simply allocate the resources they need. Though it is an unnecessary abstraction in the cloud, in cases like scientific computing, the notion of jobs can make a simple transition without much modification. However, the shift to a cloud is more complicated when the concept of fixed-length jobs does not cleanly fit an application (e.g. web services).

Targeting web services specifically, Platform-as-a-Service (PaaS) providers like Google’s AppEngine[6] and Heroku[7] are able to handle scaling automatically. In order to perform this task, users are required to develop their applications specifically for a platform the vendor supports and then submit it to the service whose task is to host that application and ensure that it scales as necessary. Unfortunately, in order to ensure that scalability can be performed intelligently they restrict developers to Python with Django or Ruby with Rails respectively. While there are programs that fit into this specific domain, they only represent a small proportion of all applications being run in the cloud.

For a more flexible service many users rely on Infrastructure-as-a-Service(IaaS) providers like Amazon EC2. With an IaaS service, users are given access to virtualized hardware, which they are free to use as needed. One particularly popular function for IaaS service providers is hosting

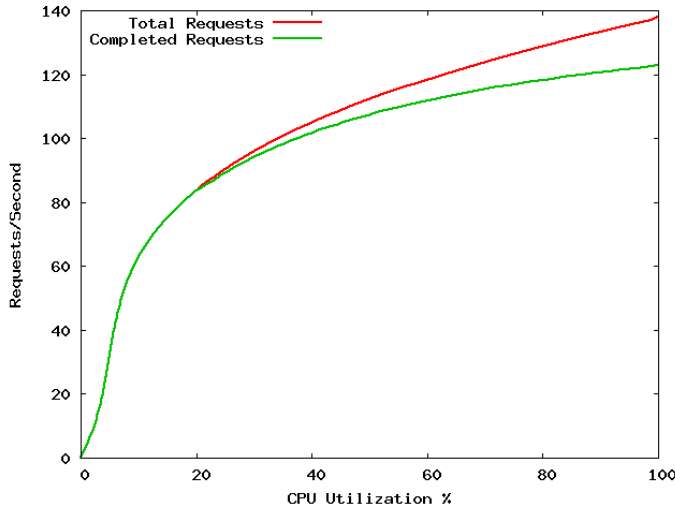


Fig. 1. Total and dropped requests as a function of CPU utilization

custom websites or web services, and the the ability to dynamically add or remove resources from a seemingly infinite pool allows websites to scale up without having to purchase and administer the physical hardware. Since users are charged on a pay-per-use model, they only pay for the CPU hours that they consume.

While IaaS services are generally more flexible and allow users to perform a wide variety of tasks, automatically scaling, as with PaaS, becomes much more difficult. Services like RightScale [8] offer tools to automatically scale cloud deployments, even without application-specific knowledge. These work by monitoring individual instance loads on top of the infrastructure level in real time. This paper explores some of the various algorithms, including RightScale’s, that could be utilized to perform automatic scaling in the cloud.

III. SIMULATOR

In order to perform multiple, repeatable trials for different algorithms, we chose to develop a simulator based on empirical findings. Using a simulator shortened the evaluation process, which is particularly important when testing multiple algorithms. Furthermore, using a simulator provided consistency across different trials, preventing external factors from influencing our results.

A. Design

The network traffic simulator is designed to model the workload from a single web application over a specified amount of time and keep track of the number of servers throughout the trial. It is made up of four parts that produce a score for a selected algorithm. The first part is an input file which lists the number of requests that occur during each simulated second of a trial. The simulator then distributes those requests to the number of machines currently running, and based on the number of requests, determines the CPU usage of each machine. For each second of a simulated traffic pattern, the list of machines and their CPU percentages are then passed

to the algorithm that instructs the simulator to start or terminate machines as necessary. At the end of each trial, the total and dropped requests along with cost are used to determine the algorithm’s score.

B. Traffic generation

Each of the algorithms described in section IV was run in our simulator using trace files generated beforehand. These trace files are intended to demonstrate how each of the algorithms performs with different types of traffic patterns. This section describes the different patterns used to evaluate the different algorithms.

1) *Weekly Oscillation*: Many studies have shown that while there is unpredictability in the web there is still a very common pattern among a large proportion of websites[11][12]. This traffic pattern, based on those patterns discussed by Wang et. al. in [13], models the traffic patterns of Monash University. What these papers demonstrate is that traffic begins to ramp up early in the morning, and continues increasing until midday where it peaks. Then, it gradually decreases into the late evening where it drops to an overnight lull. While this represents a single day, web traffic has been shown to be self-similar across different time periods, and the data from Monash confirmed the similarity of evening drops and lower request rates on weekends that our pattern exhibits.

2) *Large Spike*: This traffic pattern starts off similarly to the weekly traffic in the previous section but on the fourth day traffic spikes up and remains high for that day, and then returns to normal. This traffic pattern is similar to what might appear after a website has a big announcement or product release.

3) *Random*: This traffic pattern changes by zero-centered, normally-distributed values over time, and thus shows some stability. This is a worst-case, but still somewhat realistic traffic pattern for those algorithms that model past workloads to predict future usage.

C. CPU Conversion

As requests arrive, they are evenly distributed across all running machines, where individual request rates are converted (with some variance) into a CPU load percentage. In order to determine realistic CPU usage for a given request rate we ran an Apache web server on an Amazon EC2 instance that hosted a small PHP web page, which was dynamically generated to ensure it was not cached. We then used httperf to generate varying amounts of load to approximate different usage cases. The final mapping of request rates to CPU load was found by determining the best fit polynomial using the least-squares method.

Our results from using httperf demonstrated that the machines did not begin dropping requests until they reached approximately 20% CPU usage, after which the drop rate increased slowly up to 130 requests per second. Beyond that point, the drop rate increased linearly with requests per second, as every additional request was dropped. The results of these trials can be seen in figure 1.

We then repeated the same tests with multiple Amazon EC2 instances, using a load balancer to distribute the requests in a

round-robin fashion. These tests revealed that splitting the load across machines also splits the CPU load evenly. For example, a single computer receiving 90 requests per second had an average CPU usage of 24.75% while 180 requests split across two machines resulted in 24.92% on one machine and 24.76% on the other. The same trend continued for increasingly large numbers of requests per second.

Instances in this context generally run on equivalent hardware, thus we decided to treat machines as homogeneous. While in many cases there are in fact machines performing different roles, capturing the relationships between them and how they affect each other is beyond the scope of this paper. Furthermore, scaling properties can vary greatly between different machine roles (e.g. Web Server vs. Database) which causes additional complexity. Instead, we believe that sufficiently general scaling algorithms could be applied separately to machines in different roles to account for their particular scalability characteristics.

D. Scoring Algorithm

Given that we were testing multiple types of algorithms we decided to create a scoring metric that would allow us to objectively determine whether one algorithm performed better than another on a given trial. When judging how well an algorithm performed, a few factors included the number of total requests, the number of dropped requests, and the cost of running the machines. The number of total and dropped requests are aggregate numbers received by all of the machines over the duration of the trial. The cost is based off of the cost model used by both Amazon EC2 and Google AppEngine which charges \$0.10 per CPU hour at the start of each hour.

A simple scoring system would be to assign a value to each dropped request, combine that amount with the cost of the machines over the trial, and then minimize that sum. One issue with using this metric is determining the monetary cost of a dropped request since such a value is closely tied to a given website. For example, an e-commerce site that assumes a certain amount of revenue as a result of large, one-time purchases values individual requests more than a website that generates fractions of a cent per request on ad revenue. Instead, we propose a metric less dependant on a given web service's revenue model, which can nevertheless be tuned to balance the desired availability and cost to operate of the website.

Availability as a base metric accounts for dropped requests relative to the total traffic: $A = (\#servicedrequests)/(\#oftotalrequests)$. Given that many web service administrators already use availability percentages to gauge their capacity, we found this to be intuitive. Cost is also easily understood, expressed by: $C = \#CPU/hours * 0.10$.

Our scoring metric is based on the following web service characteristics

- 1) A scaling method's value increases exponentially with respect to availability.
A common expression of availability is in terms of "number of nines," whereby 99.99% is "four nines." Though typically used as an informal metric, this type of

mindset implies that the value assigned to a particular availability increases exponentially, since the increase from 99% to 99.9% is weighted at least as heavily as the increase from 99.9% to 99.99%. We express this relationship as:

$$A_{log} = -\log(1 + \delta_a - A), \quad \delta_a < 1$$

Where δ_a is a parameter used to ensure that 100% availability can be expressed as a bounded value ($\delta_a = 0$ implies that perfect availability has infinite weight). The final weight of availability can be expressed as

$$W_A = (A_{log})^\alpha$$

where α is a user-defined parameter used to express the value of availability to the service.

- 2) As cost increases, the perceived cost of each additional dollar decreases.

Expressed as

$$W_A/C$$

This characteristic is based on the notion of a web service's financial investment. For example, an additional \$15/week would be a much more noticeable difference for an administrator who was currently spending \$10/week than to an administrator who was spending \$1000/week.

- 3) The minimum perceived cost is nonzero and depends on availability.

Expressed as

$$\gamma C/A_{log}$$

where γ is a minimal cost weight parameter. While perceived cost decreases, it does not approach zero, but rather a minimum rate. This baseline perceived cost is a factor of the level of availability because highly available systems are expected to be more expensive. Thus, \$10 spent to maintain 99.99% availability is a better "deal" than \$10 spent to maintain 99% availability. Eventually, however, cost overtakes any gains in availability, presumably at the point where cost becomes prohibitive.

Combining these, our final scoring metric is

$$Score = \frac{(A_{log})^\alpha}{C} - \frac{\gamma C}{A_{log}} + \beta$$

where β is an optional baseline parameter, which can give meaning to a score of 0 as "exhausting our maximum cost for a given availability." This equation, in a general sense, gives the efficiency of a scaling algorithm in that it attempts to maximize server capacity with the smallest amount of resources. We believe this to be a better indicator of how well an algorithm performs than, say, CPU utilization, which is generally considered less important.

In our experiments, $\delta_a = 0.000000001$, $\alpha = 2$, $\gamma = 1$, and $\beta = 50$. We decided on these values based on experimentation.

IV. ALGORITHM DESIGN

This section describes the different algorithms we used to perform dynamic scaling.

A. Static Provisioning

The goal of static provisioning is to provide a baseline result against which to compare other algorithms. Static provisioning simply determines the minimum number of machines required to achieve 100% availability at the peak of a given trial, and then runs that number of machines for the duration of the trial. This provides a best case result for a static allocation that achieves 100% availability since there is prior knowledge of the number of machines required. This is unlikely to occur in practice since it is rarely the case that one knows a priori the exact number of machines which will be required. In fact it is common that websites will maintain multiple times the machines required at peak load to ensure they are able to serve their customers even at times of heavy traffic. [9]

B. The RightScale Algorithm

RightScale[8] provides a host of services that are designed specifically for the domain of cloud management, including a scaling system. Their algorithm is intended to be simple and general so it can be applied to an “array” (i.e. homogeneous set) of machines, allowing the resource utilization to dynamically scale based on load.[1] In its most basic form, the algorithm is a simple democratic voting process whereby, if a majority of the machines agree that they should scale up or down, that action is taken, otherwise no action occurs. Machines vote to scale up or scale down based on whether their current CPU value has been consistently above or below a predefined user threshold (default 30% and 85% respectively) for a given duration, which has a default of three minutes. In addition, there is a period called the “resize calm time” which must elapse after a given scaling action has occurred before the voting process can take place again[2]. This buffer period is intended to prevent the algorithm from continually allocating resources as new instances boot.

In trials with the RightScale algorithm we used the recommended values for all of the parameters defined above except for a few notable exceptions. First, we tuned the scaling thresholds to better match our test application in order to minimize dropped requests. Next, we set the resize calm time to three minutes instead of the recommended 15. RightScale recommends 15 minutes to ensure an action has completed because machines generally take between 5 to 10 minutes to start, and it is preferable to overestimate an instance’s boot time than to underestimate it. We chose a calm time equal to our machine boot time of three minutes so that the algorithm is able to make a decision immediately after a new machine becomes operational. Since this is the ideal situation we consider the results of the RightScale algorithm to be the best case scenario for the resize time.

C. Autoregressive of Order 1 (AR1) Model

Modeled after the predictor module from [10], this algorithm implements an AR(1) process in order to estimate future workload based on a finite history window. The application of autoregression to workload estimation treats the history of request rates as a signal with predictable properties. The quality

of the model’s fit to the actual trends therefore determines the accuracy of the estimation.

The process is modeled as operating over a sliding window of past intervals, predicting values to fill the adaptation window. Each interval in the history window, a_i , ($1 \leq i \leq N_h$) is the total requests per second averaged over the interval size. The size of the history window, N_h , determines the sensitivity of the algorithm to local versus global trends, while the size of the adaptation window, N_a , determines the magnitude of each projection (how far into the future the model extends). Our model takes as input instantaneous request rates (requests/second), projects the average request rate of the next interval, and scales accordingly.

Upon filling a new interval, the AR1 algorithm predicts N_a following intervals, a_i , ($N_h < i \leq N_h + N_a$) in the adaptation window, by the equation:

$$a_{i+1} = a_{avg} + \rho(1)(a_i - a_{avg}) + e_i$$

where a_{avg} is the mean of the history window, and e_i is a noise constant, kept at 0.

$\rho(1)$ is the autocorrelation function with a lag of 1 given by:

$$\rho(l) = \frac{1}{(N_a + N_h - l)} * \sum_{i=1}^{N_a + N_h - l} \frac{(a_i - a_{avg})(a_{i+l} - a_{avg})}{\sigma_h^2}$$

The final projected request rate is the mean of the adaptation window, so the larger the adaptation window, N_a , the farther out the prediction estimates. The scaling magnitude is determined by dividing the predicted rate by a maximum and minimum capacity threshold for a single machine, and adding or subtracting machines to keep each CPU’s load within that range.

For our trials, each interval spans 100 seconds, and the history and adaptation windows are 10 and 3 intervals respectively, though in [10] no particular parameter values are given. In practical terms, these values give our algorithm a granularity of roughly 2 minutes, and predicts the average load over the next 5 minutes based on trends of the last 20 minutes.

D. Linear Regression

Linear regression is a commonly used statistical method to determine the polynomial function that is closest to a set of points. The objective is to find a polynomial such that the distance from each of the points to the polynomial curve is as small as possible and therefore “fits” the data best.

In this algorithm we used linear least-squares to find a polynomial function that is close to our observations. Ideally, we would like our polynomial to pass through all the points in our data set, but this is typically impossible, which leaves us with a system of linear equations with no exact solution. We then define a function that is equal to the sum of the square distances of points to a generic polynomial. Differentiating the function to find the minimum yields the coefficients for the polynomial function that is closest to the data. In our trials we used a linear regression algorithm to predict the amount of requests per second that will occur in the future. In order to do this, the algorithm splits the previous requests per

second into tumbling windows of 100 seconds. Upon filling a window, it approximates the data with a quadratic equation, using least-squares linear regression. This equation predicts what the highest number of requests per second will be in the next 100 seconds, the algorithm then compares the predicted CPU usage with the current computing resources and scales up or down as necessary.

E. Intelligent Deallocation

While this paper deals with several different algorithms for scaling in a broad sense, ensuring that machines use an entire hour applies to every scaling decision. In general, machines are allocated on an hourly basis, charged upon instantiation. Therefore, even if the load is low and the machine is not required, the entire hour has been paid for, so there is no reason to terminate a server before the hour is over. A beneficial property of using this “smart kill” optimization is that overly sensitive algorithms will be tempered such that they do not “churn,” rapidly starting and then terminating machines due to short-lived lulls in workload. Care must be taken, however, in choosing how near the hour mark a machine must be in order to be terminated such that the algorithm has sufficient time to make a decision. For example, allowing the AR1 algorithm to scale down only within 60 seconds of an hour when its decision frequency is once every 100 seconds results in missed opportunities to deallocate unneeded machines.

V. RESULTS

In this section, we discuss the results of each algorithm’s performance over our simulated traffic patterns.

A. Static Provisioning

Static provisioning is by far the most expensive algorithm to run, especially as the number of machines required increases. In particular, the number of machines required to handle spiked traffic model is very large, yet aside from the peak demand period only a small fraction of the total machines are necessary. As a result of this, dynamic algorithms can reduce the cost by as much as 93% while reducing availability by as little as 0.06%.

However, in the oscillating and random trials the static provisioning scheme achieves scores on par with the other algorithms. This is due to the fact that our scoring metric parameters are heavily weighted towards availability, and static provisioning drops zero requests; thus its higher cost is overshadowed by the higher availability. It should be noted however that the trials are only one week in duration and thus total costs are relatively small. Over a longer period, the relative differences in cost between the algorithms will grow, while the availability should remain fixed, essentially weighting cost more heavily.

B. RightScale

Given how simple it is, the RightScale algorithm is able to perform quite well on all trials. In both the weekly oscillation and large spike models the RightScale algorithm scores only

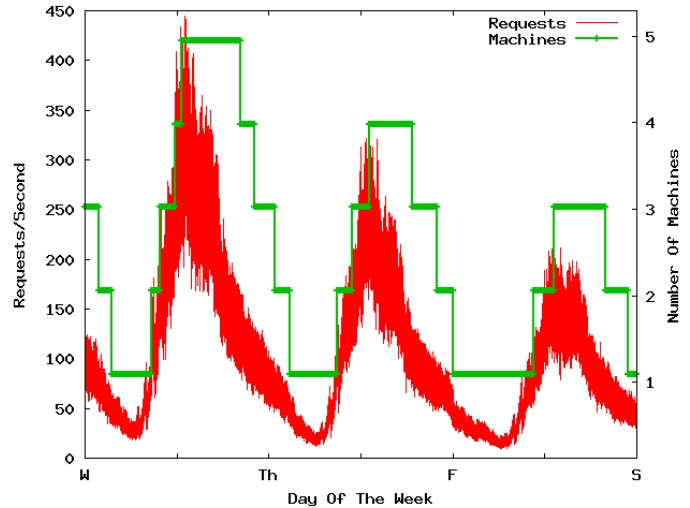


Fig. 2. RightScale on a portion of the weekly oscillation traffic pattern

slightly lower than AR1 and linear regression algorithms. While its availability is generally lower than the other algorithms, it tends to be cheaper and as a result is able to attain scores only slightly lower than the other algorithms.

We found that the RightScale algorithm is highly dependant on the values of the scaling thresholds since a majority of machines have to be entirely above or below a threshold for three minutes in order for scaling to happen. Unlike the other algorithms, RightScales is more reactive rather than predictive, so in order to achieve a reasonable amount of responsiveness from the algorithm the maximum CPU threshold needs to be tuned to quite a low value. While linear regression and AR1 adapted on their own in the spiked traffic pattern, the user-defined thresholds of the RightScale algorithm had to be manually tuned over all trials to account just for this possible workload.

While requiring machines to consistently be above or below a threshold in order to vote hampers the algorithms responsiveness, it provides benefits as displayed by its results in the random traffic model. The RightScale algorithm is able to achieve particularly high availability scores due to low CPU thresholds and the lack of consistency in the traffic. Early on, the algorithm starts enough machines to sustain the initial load then as a result of the erratic traffic was only able to achieve majority votes when scaling was absolutely necessary. Furthermore, the low CPU thresholds allowed the algorithm to stay comfortably above the number of machines required to handle the load. This accounts for why the algorithm achieved such high availability, but also cost 10-20% more than the other algorithms.

Though it is less expensive with smart kill enabled, one thing unique to the Rightscale algorithm is that smart kill negatively affected availability. This is seen in figure 2 where, as the request rate changes, the number of CPUs adapts once then plateaus for some time. This is because the RightScale algorithm’s decision to scale depends on the timing of its previous scaling event (i.e. calm time). Smart kill’s delayed machine termination spreads out scaling events, increasing the

TABLE I
SIMULATION RESULTS OVER OUR TRAFFIC PATTERNS

Algorithm	Standard Oscillation			Large Spike			Random		
	Availability	Cost	Score	Availability	Cost	Score	Availability	Cost	Score
Static Allocation	100%	\$135.20	50.75	100%	\$1,994.20	-21.79	100%	\$726.70	24.75
Rightscale	99.98%	\$45.10	46.29	99.4%	\$129.10	24.7	99.99998%	\$495.40	18.03
RightScale w/ Smartkill	99.98%	\$43.30	47.85	99.3%	\$125.10	25.52	99.9997%	\$489.90	11.11
AR1	99.996%	\$44.90	47.85	99.9%	\$130.60	30.43	99.96%	\$444.70	-6.73
AR1 w/ Smartkill	99.998%	\$46.50	48.43	99.9%	\$132.30	30.17	99.9999%	\$450.00	17.3
Linear Regression	99.93%	\$51.00	44.05	99.91%	\$144.00	29.85	99.995%	\$435.20	6.74
Linear Regression w/ Smartkill	99.999%	\$45.90	48.09	99.94%	\$158.80	28.99	99.996%	\$397.70	10.8

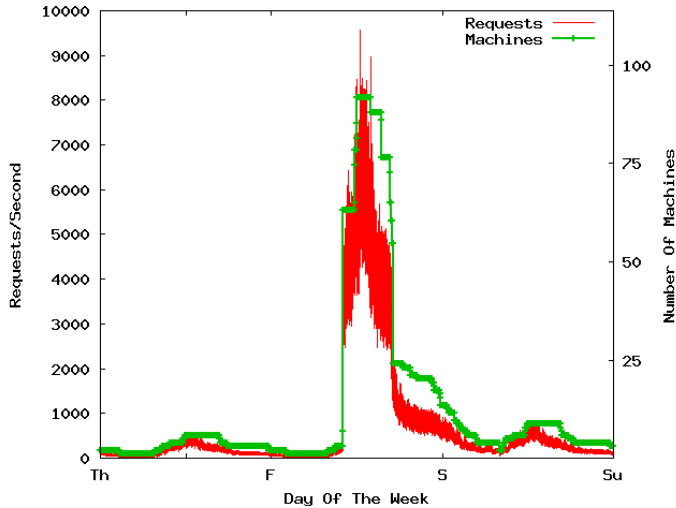


Fig. 3. AR1 on a portion of the large spike traffic pattern

number of periods where the algorithm is in “resize calm time” and cannot react to changes in load. This generally results in more dropped requests, which implies less availability, but also results in lower overall cost.

C. AR1

Like the RightScale algorithm, AR1’s performance depends largely on a set of user-defined parameters. Monitoring-interval length and the size of the history window greatly affected the sensitivity of the algorithm in preliminary tests, and the size of the adaptation window changed the magnitude of every predicted request rate. In defining the history window to be 10 intervals of be 100 seconds, for instance, we made a deliberate choice to account for trends of about 20 minutes, and to ignore variations smaller than roughly two minutes in length. This made the algorithm necessarily less responsive to local changes than the linear regression algorithm, as discussed in section V-D.

In addition, tuning the length of the adaptation window has a more dramatic effect of magnifying each prediction. Predicting over an adaptation window of 300 seconds (3 intervals) resulted in meaningful but tempered predictions since the algorithm was essentially tracing the current trend out only so far into the future. Any larger an adaptation window, and trends of even modest rate increases resulted in the algorithm consistently overestimating rate changes in the next interval.

The biggest difference between AR1 and Rightscale is that AR1 predicts future values by modeling past behavior. The consequences are most pronounced on the large spike trial, as the unprecedented load overwhelms RightScale’s statically set scaling factor, while AR1 adapts to current conditions and scales quickly enough to better maintain availability. However, the gains from variable prediction are offset by the low responsiveness of the algorithm, shown by the delay between a large increase in workload and the appropriate scaling call as seen in 3.

One consequence of using smart kill comes when machines are allocated en masse. Then, it is more difficult for machines to scale down except in large bursts at the end of their common hour. Machines allocated in smaller increments tend to spread out their hourly downscale boundaries, and thus have a higher chance of being deallocated in a given second. For example, if a request to scale down one machine comes at a random time uniformly distributed across an hour interval, and N machines share a start time, they each have $120/(N * 3600)$ chance of being de-allocated (given a 2 minute window for scaling down). Clustered machines essentially get to share the probability of scaling down among themselves, while isolated machines must have a more equal chance of being deallocated.

In the end, single start-time clustering due to large allocations results in the machine pool maintaining a set of “stale” instances with recently allocated instances coming and going more frequently. As seen in the random traffic trial, with quick increases in request rate the AR1 algorithm tends to allocate many machines at once, and only benefits from smart kill if the number of brief downscales due to churning is costlier than unnecessary machine hours due to clustered start times.

D. Linear Regression

Similar to the AR1 model, the linear regression algorithm predicts workloads based on past behavior. As such, it can scale more quickly or slowly as the situation requires, evidenced by the improved availability in the large spike trial. Unlike AR1, however, the linear regression algorithm models past behavior in much finer granularity, sampling over a window of one second request rates. This leads the algorithm to respond much more quickly to new rate changes as seen in 4. On its own, the algorithm will scale up and, unfortunately, scale down more aggressively than AR1, leading often to increased cost without significant gains in availability, as seen in the weekly oscillation and large spike trial without smart kill.

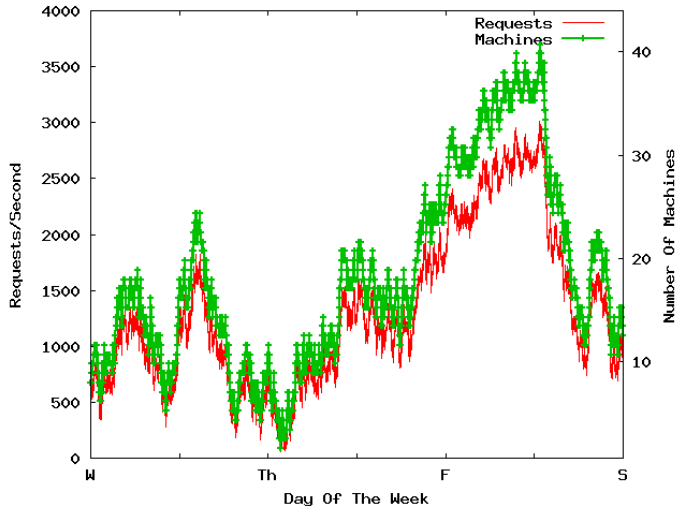


Fig. 4. Linear regression on a portion of the random traffic pattern

The linear regression algorithm tends to allocate large sets of machines at once, much like AR1, but unlike AR1, it sees significant benefit from the smoothing that smart kill provides. Since the linear regression model frequently churns, smart kill reduces the cost of constantly re-allocating the same machine, which also improves availability because it often eliminates the three minute start up period required of each new machine. With smart kill, therefore, the linear regression algorithm can most easily scale up aggressively when needed, yet scales down much more modestly, receiving a better score on the weekly oscillation and random trials. Linear regression suffered similarly to AR1 on the large spike trial with smart kill due to start-time clustering inefficiencies.

VI. RELATED WORK

In [10], the authors showed that an autoregressive prediction model is effective for mitigating the negative effects of unexpected traffic spikes. Sharing many of our motivations, they implemented a three module, generalized processor sharing (GPS) system to monitor, predict, and ultimately allocate for traffic patterns. Unlike our algorithms, however, their system could not allocate computing power dynamically as their focus was on managing fixed amounts of resources in data centers, while we could increase capacity on demand.

As a consequence, their system was notably more complex than our AR(1) implementation. First, their system monitored memory usage, and storage consumption in addition to CPU usage. Without prior knowledge of an application's behavior, these metrics required a sophisticated model to adjust mapping from request rates to server loads. Second, their model accounted for uneven request distributions for different server types (eg. application servers require more CPU per request, while database servers are storage bound), so the predictor had to estimate the distribution of incoming requests as well as the absolute amount. Third, their model allowed for QoS requirements for different request types, further constraining the allocation process so that high priority requests were given precedence.

VII. FUTURE WORK

Because the scope of our experiments is limited, some issues could benefit from additional research. In particular, studying the scaling properties of other systems and how they compare to those seen in web servers could be valuable. While web servers are a very popular, many other applications can be run in the cloud (e.g. MapReduce). Exploring the implications of these different usage models to the parameters of our scoring metric could generate a set of default values for each application domain. Our first impression is to evaluate different scoring parameters by profiling load characteristics (memory, storage and CPU usage per request, etc.) to gain insight into an application's behavior in the cloud.

Throughout this project, we assumed all computers were homogeneous. Considering different resources for each computer greatly complicates analysis and simulation, but is an obvious improvement over our current approach. In addition, our simulations only considered CPU usage to be the bottleneck, though extending our model to include memory and I/O considerations are definitely interesting areas to explore. One possible implication of complicating our simulation model is the importance of simplicity in each scaling algorithm. While linear regression and autoregression viably predicted CPU load, they may prove prohibitively slow when used in more complex, real-time resource prediction.

While adding smart kill to the algorithms improved their efficiency, in most cases it also proved to make them more costly. The inherent problem of clustered start-time deallocation discussed in section V-C can detract from an otherwise positive algorithm, and we believe that improving smart kill would be worthwhile. One possible fix could be adjusting the start time of each machine when several machines are started together, thus separating the windows at which they can be killed.

Several potential improvements could be explored regarding the linear regression algorithm. Our current implementation provides a tight fit for the data, and thus it constantly overreacts to small fluctuations. While this is good for following quick changes, increasing the size of each sample in a window as well as performing linear regression over a larger window of samples could calm the sensitivity sufficiently without decreasing availability. Another improvement could be regressing over windows of different sizes, and then outputting the mean of all the predictions. This would result in smoother outputs, while still being somewhat sensitive to recent changes.

The AR1 algorithm, however, has the opposite problem, in that it does not react quickly enough when the data changes. While the window size of 10 intervals seems to work well, interval lengths of 100 seconds are probably too long, considering the algorithm takes several intervals to react. Possible values of 10, 30, and 60 seconds could result in better fine-grain predictions.

It appears that the ideal algorithm would be somewhere in between linear regression and AR1: something that can react quickly to variations in the data, but also scales up or down in less erratic patterns. One idea that we believe could prove to be fruitful would be applying the technique

of boosting to algorithms used above. In essence, this method could utilize the best predictions of a large set of differently tuned scaling algorithms. It is clear in our results that each algorithm had cases where it performed particularly well, and as such a boosting algorithm that gives weight to algorithms that perform well on the small scale could improve overall results. Our scoring metric already provides the necessary mechanism to determine each algorithm's weight.

VIII. CONCLUSION

While each algorithm has its particular benefits and disadvantages, we have shown the significance of the ability for cloud computing users to allocate resources on demand. It is clear that the typical data center model of static provisioning is wasteful in terms of resources while only resulting in minor increases in availability. Dynamic provisioning, on the other hand, achieved greater than 99.9% availability in most cases while significantly reducing the overall cost.

Our scoring metric proved to be very useful for to comparing algorithms in that it is very effective at capturing and weighing the important parameters required to judge the different algorithms. We also believe that the scoring algorithm is flexible enough such that it can be applied to other configurations with different parameter weights with only minor modification.

As increasing numbers of services shift into the cloud, the demand for intelligent and cost-effective solutions for dynamically scaling machines will only increase. Also, representative metrics will have to emerge in order to realistically evaluate the different scaling approaches. While there are still open questions to be studied, we believe that we have made important steps into what is a new and exciting area of research.

ACKNOWLEDGMENT

The authors of this paper would like to thank RightScale and particularly Thorsten von Eicken whose talk was the inspiration for this project.

REFERENCES

- [1] 2009. [Online]. Available: http://wiki.rightscale.com/1_Tutorials/02-AWS/02-Website_Edition/How_do_I_set_up_Autoscaling%3f
- [2] 2009. [Online]. Available: http://wiki.rightscale.com/2_References/01-RightScale/01-RightScale_Dashboard/01-RightScale_Dashboard_User_Guide/01-Manage/02-Arrays/Create_an_Alert-based_Server_Array
- [3] "Amazon elastic compute cloud (amazon ec2)," 2009. [Online]. Available: <http://aws.amazon.com/ec2/>
- [4] "Appscale," 2009. [Online]. Available: <http://appscale.cs.ucsb.edu>
- [5] "Eucalyptus: Elastic utility computing architecture for linking your programs to useful systems," 2009. [Online]. Available: <http://eucalyptus.cs.ucsb.edu>
- [6] "Google app engine," 2009. [Online]. Available: <http://code.google.com/appengine/>
- [7] "Heroku," 2009. [Online]. Available: <http://heroku.com/>
- [8] "Rightscale inc." 2009. [Online]. Available: <http://www.rightscale.com>
- [9] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "Above the Clouds: A Berkeley View of Cloud Computing."

- [10] A. Chandra, W. Gong, and P. Shenoy, "Dynamic resource allocation for shared data centers using online measurements," in *SIGMETRICS '03: Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM, 2003, pp. 300–301.
- [11] M. Crovella and A. Bestavros, "Self-similarity in world wide web traffic: evidence and possible causes," *Networking, IEEE/ACM Transactions on*, vol. 5, no. 6, pp. 835–846, Dec 1997.
- [12] S. Manley and M. Seltzer, "Web facts and fantasy," in *USITS'97: Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*. Berkeley, CA, USA: USENIX Association, 1997, pp. 12–12.
- [13] X. Wang, A. Abraham, and K. A. Smith, "Intelligent web traffic mining and analysis," *J. Netw. Comput. Appl.*, vol. 28, no. 2, pp. 147–165, 2005.