AGILE: elastic distributed resource scaling for Infrastructure-as-a-Service

Hiep Nguyen, Zhiming Shen, Xiaohui Gu
North Carolina State University
{*hcnguye3,zshen5*}*@ncsu.edu, gu@csc.ncsu.edu*

Sethuraman Subbiah
NetApp Inc.
sethu.subbiah@netapp.com

John Wilkes
Google Inc.
johnwilkes@google.com

Abstract

Dynamically adjusting the number of virtual machines (VMs) assigned to a cloud application to keep up with load changes and interference from other uses typically requires detailed application knowledge and an ability to know the future, neither of which are readily available to infrastructure service providers or application owners. The result is that systems need to be over-provisioned (costly), or risk missing their performance Service Level Objectives (SLOs) and have to pay penalties (also costly). AGILE deals with both issues: it uses wavelets to provide a medium-term resource demand prediction with enough lead time to start up new application server instances before performance falls short, and it uses dynamic VM cloning to reduce application startup times. Tests using RUBiS and Google cluster traces show that AGILE can predict varying resource demands over the medium-term with up to 3.42× better true positive rate and 0.34× the false positive rate than existing schemes. Given a target SLO violation rate, AGILE can efficiently handle dynamic application workloads, reducing both penalties and user dissatisfaction.

1 Introduction

Elastic resource provisioning is one of the most attractive features provided by Infrastructure as a Service (IaaS) clouds [2]. Unfortunately, deciding when to get more resources, and how many to get, is hard in the face of dynamically-changing application workloads and service level objectives (SLOs) that need to be met. Existing commercial IaaS clouds such as Amazon EC2 [2] depend on the user to specify the conditions for adding or removing servers. However, workload changes and interference from other co-located applications make this difficult.

Previous work [19, 39] has proposed prediction-driven resource scaling schemes for adjusting how many resources to give to an application within a single host. But distributed resource scaling (e.g., adding or removing servers) is more difficult because of the latencies involved. For example, the mean instantiation latency in Amazon EC2 is around 2 minutes [8], and it may then take a while for the new server instance to warm up: in our experiments, it takes another 2 minutes for a Cassandra server [4] to reach its maximum throughput. Thus, it is insufficient to apply previous short-term (i.e., less than a minute) prediction techniques to the distributed resource scaling system.

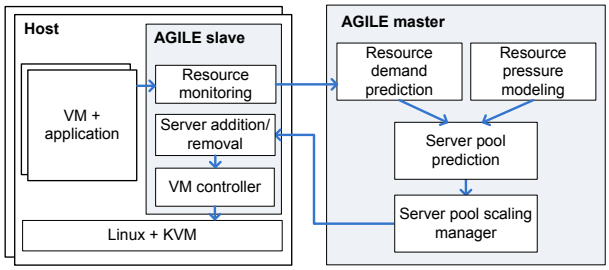


Figure 1: The overall structure of the AGILE system. The AGILE slave continuously monitors the resource usage of different servers running inside local VMs. The AGILE master collects the monitor data to predict future resource demands. The AGILE master maintains a dynamic resource pressure model for each application using online profiling. We use the term *server pool* to refer to the set of application VMs that provide the same replicated service. Based on the resource demand prediction result and the resource pressure model, the AGILE master invokes the server pool manager to add or remove servers.

In this paper, we present our solution: AGILE, a practical elastic distributed resource scaling system for IaaS cloud infrastructures. Figure 1 shows its overall structure. AGILE provides medium-term resource demand predictions for achieving enough time to scale up the server pool before the application SLO is affected by the increasing workload. AGILE leverages pre-copy live

cloning to replicate running VMs to achieve immediate performance scale up. In contrast to previous resource demand prediction schemes [19, 18], AGILE can achieve sufficient lead time without sacrificing prediction accuracy or requiring a periodic application workload.

AGILE uses online profiling and polynomial curve fitting to provide a black-box performance model of the application's SLO violation rate for a given resource pressure (i.e., ratio of the total resource demand to the total resource allocation for the server pool). This model is updated dynamically to adapt to environment changes such as workload mix variations, physical hardware changes, or interference from other users. This allows AGILE to derive the proper resource pressure to maintain to meet the application's SLO target.

By combining the medium-term resource demand prediction with the black-box performance model, AGILE can predict whether an application will enter the overload state and how many new servers should be added to avoid this.

Contributions

We make the following contributions in this paper.

- We present a wavelet-based resource demand prediction algorithm that achieves higher prediction accuracy than previous schemes when looking ahead for up to 2 minutes: the time it takes for AGILE to clone a VM.
- We describe a resource pressure model that can determine the amount of resources required to keep an application's SLO violation rate below a target (e.g., 5%).
- We show how these predictions can be used to clone VMs proactively before overloads occur, and how dynamic memory-copy rates can minimize the cost of cloning while still completing the copy in time.

We have implemented AGILE on top of the KVM virtualization platform [27]. We conducted extensive experiments using the RUBiS multi-tier online auction benchmark, the Cassandra key-value store system, and resource usage traces collected on a Google cluster [20]. Our results show that AGILE's wavelet-based resource demand predictor can achieve up to 3.42× better true positive rate and 0.34× the false positive rate than previous schemes on predicting overload states for real workload patterns. AGILE can efficiently handle changing application workloads while meeting target SLO violation rates. The dynamic copy-rate scheme completes the cloning before the application enters the overload state with minimum disturbance to the running system. AGILE is light-weight: its slave modules impose less than 1% CPU overhead.

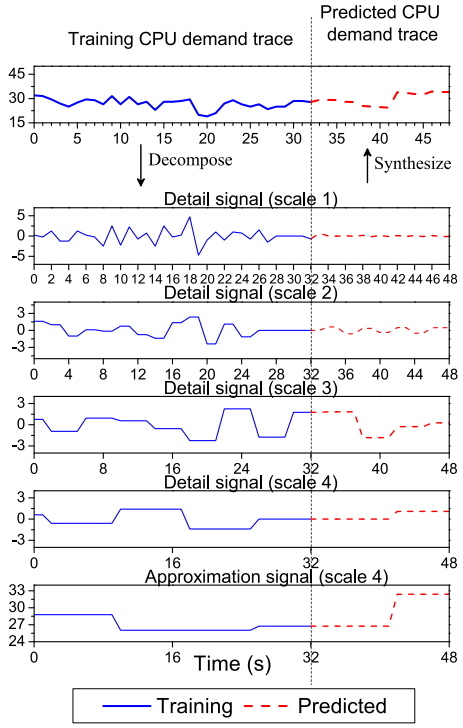


Figure 2: Wavelet decomposition of an Apache web server CPU demand under a real web server workload from the ClarkNet web server [24]. The original signal is decomposed into four detailed signals from scale 1 to 4 and one approximation signal using Haar wavelets. At each scale, the dotted line shows the predicted signal for the next future 16 seconds at time t = 32 second.

2 AGILE system design

In this section, we first describe our medium-term resource demand prediction scheme. By "medium-term", we mean up to 2 minutes (i.e., 60 sampling intervals given a 2-second sampling interval). We then introduce our online resource pressure modeling system for mapping SLO requirements to proper resource allocation. Next, we describe the dynamic server pool scaling mechanism using live VM cloning.

2.1 Medium-Term Resource demand prediction using Wavelets

AGILE provides *online* resource demand prediction using a sliding window D (e.g., $D = 6000$ seconds) of recent resource usage data. AGILE does not require advance application profiling or white-box/grey-box application modeling. Instead, it employs *wavelet transforms* [1] to make its medium-term predictions: at each sampling instant t, predicting the resource demand over the prediction window of length W (e.g., $W = 120$

seconds). The basic idea is to first decompose the original resource demand time series into a set of wavelet based signals. We then perform predictions for each decomposed signal separately. Finally, we synthesize the future resource demand by adding up all the individual signal predictions. Figure 2 illustrates our wavelet-based prediction results for an Apache web server's CPU demand trace.

Wavelet transforms decompose a signal into a set of wavelets at increasing scales. Wavelets at higher scales have larger duration, representing the original signal at coarser granularities. Each scale i corresponds to a wavelet duration of L_i seconds, typically $L_i = 2^i$. For example, in Figure 2, each wavelet at scale 1 covers 2^1 seconds while each wavelet at scale 4 covers $2^4 = 16$ seconds. After removing all the lower scale signals called *detailed signals* from the original signal, we obtain a smoothed version of the original signal called the *approximation signal*. For example, in Figure 2, the original CPU demand signal is decomposed into four detailed signals from scale 1 to 4, and one approximation signal. Then the prediction of the original signal is synthesized by adding up the predictions of these decomposed signals.

Wavelet transforms can use different basis functions such as the Haar and Daubechies wavelets [1]. In contrast, Fourier transforms [6] can only use the sinusoid as the basis function, which only works well for cyclic resource demand traces. Thus, wavelet transforms have advantages over Fourier transforms in analyzing acyclic patterns.

The scale signal i is a series of independent non-overlapping chunks of time, each with duration of 2^i (e.g., the time intervals [0-8), [8-16)). We need to predict $W/2^i$ values to construct the scale i signal in the look-ahead window W as adding one value will increase the length of the scale i signal by 2^i.

Since each wavelet in the higher scale signal has a larger duration, we have fewer values to predict for higher scale signals given the same look-ahead window. Thus, it is easier to achieve accurate predictions for higher scale signals as fewer prediction iterations are needed. For example, in Figure 2, suppose the look-ahead window is 16 seconds, we only need to predict 1 value for the approximation signal but we need to predict 8 values for the scale 1 detail signal.

Wavelet transforms have two key configuration parameters: 1) the wavelet function to use, and 2) the number of scales. AGILE dynamically configures these two parameters in order to minimize the prediction error. Since the approximation signal has fewer values to predict, we want to maximize the similarity between the approximation signal and the original signal. For each sliding window D, AGILE selects the wavelet function that results in the smallest Euclidean distance between the approximation signal and the original signal. Then, AGILE sets the number of values to be predicted for the approximation signal to 1. It does this by choosing the number of scales for the wavelet transforms. Given a look-ahead window W, let U denote the number of scales (e.g., scale of the approximation signal). Then, we have $W/2^U = 1$, or $U = \lceil log_2(W) \rceil$. For example, in Figure 2, the look-ahead window is 16 seconds, so AGILE sets the maximum scale to $U = \lceil log_2(16) \rceil = 4$.

We can use different prediction algorithms for predicting wavelet values at different scales. In our current prototype, we use a simple Markov model based prediction scheme presented in [19].

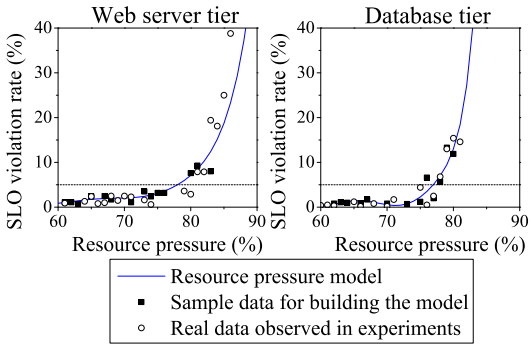


Figure 3: Dynamically derived CPU resource pressure models mapping from the resource pressure level to the SLO violation rate using online profiling for RUBiS web server and database server. The profiling time for constructing one resource pressure model is about 10 to 20 minutes.

2.2 Online resource pressure modeling

AGILE needs to pick an appropriate resource allocation to meet the application's SLO. One way to do this would be to predict the input workload [21] and infer the future resource usage by constructing a model that can map input workload (e.g., request rate, request type mix) into the resource requirements to meet an SLO. However, this approach often requires significant knowledge of the application, which is often unavailable in IaaS clouds and might be privacy sensitive, and building an accurate workload-to-resource demand model is nontrivial [22].

Instead, AGILE predicts an application's resource usage, and then uses an application-agnostic *resource pressure* model to map the application's SLO violation rate target (e.g., $< 5\%$) into a maximum resource pressure to maintain. Resource pressure is the ratio of resource usage to allocation. Note that it is necessary to allocate a little more resources than predicted in order to accommodate transient workload spikes and leave some headroom for the application to demonstrate a need for

more resources [39, 33, 31]. We use online profiling to derive a resource pressure model for each application tier. For example, Figure 3 shows the relationship between CPU resource pressure and the SLO violation rate for the two tiers in RUBiS, and the model that AGILE fits to the data. If the user requires the SLO violation rate to be no more than 5%, the resource pressure of the web server tier should be kept below 78% and the resource pressure of the database tier below 77%.

The resource pressure model is application specific, and may change at runtime due to variations in the workload mix. For example, in RUBiS, a workload with more write requests may require more CPU than the workload with more browse requests. To deal with both issues, AGILE generates the model dynamically at runtime with an application-agnostic scheme that uses online profiling and curve fitting.

The first step in building a new mapping function is to collect a few pairs of resource pressure and SLO violation rates by adjusting the application's resource allocation (and hence resource pressure) using the Linux `cgroups` interface. If the application consists of multiple tiers, the profiling is performed tier by tier; when one tier is being profiled, the other tiers are allocated sufficient resources to make sure that they are not bottlenecks. If the application's SLO is affected by multiple types of resources (e.g., CPU, memory), we profile each type of resource separately while allocating sufficient amounts of all the other resource types. We average the resource pressures of all the servers in the profiled tier and pair the mean resource pressure with the SLO violation rate collected during a profiling interval (e.g., 1 minute).

AGILE fits the profiling data against a set of polynomials with different orders (from 2 to 16 in our experiment) and selects the best fitting curve using the least-square error. We set the maximum order to 16 to avoid overfitting. At runtime, AGILE continuously monitors the current resource pressure and SLO violation rate, and updates the resource pressure model with the new data. If the mapping function changes significantly (e.g., due to variations in the workload mix), and the approximation error exceeds a pre-defined threshold (e.g., 5%), AGILE replaces the current model with a new one. Since we need to adjust the resource allocation gradually and wait for the application to become stable to get a good model, it takes about 10 to 20 minutes for AGILE to derive a new resource pressure model from scratch using the online profiling scheme. To avoid frequent model retraining, AGILE maintains a set of models and dynamically selects the best model for the current workload. This is useful for applications that have distinct phases of operation. A new model is built and added only if the approximation errors of all current models exceed the threshold.

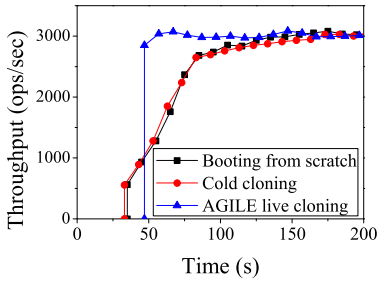


Figure 4: Performance of a new Cassandra server using different server instantiation mechanisms in KVM. All measurements start at the time of receiving a new server cloning request. We expect post-copy live cloning would behave similar to cold cloning.

2.3 Dynamic server pool scaling

Our technique for scaling up the server pool when overload is predicted distinguishes itself from previous work [28, 8] in terms of agility: servers can be dynamically added with little interference, provide near immediate performance scale-up, and low bandwidth cost using adaptive copy rate configuration.

There are multiple approaches to instantiate a new application server:

1. *Boot from scratch*: create a new VM and start the OS and application from the beginning.
2. *Cold cloning*: create a snapshot of the application VM beforehand and then instantiate a new server using the snapshot.
3. *Post-copy live cloning* [28]: instantiate a new server by cloning one of the currently running VMs, start it immediately after instantiation and use demand paging for memory copy.
4. *Pre-copy live cloning*: instantiate a new server from an already running VM. The new server is started after almost all the memory has been copied.

AGILE uses the last of these, augmented with rate control over the data transfer to achieve rapid performance scale-up, minimize interference with the source VMs, and avoid storing and maintaining VM snapshots. Figure 4 shows the throughput of a new Cassandra server [4] using different server instantiation schemes. AGILE allows the new instance to reach its maximum performance immediately, while the others take about 2 minutes to warm up. Note that AGILE triggers the live cloning *before* the application enters the overload state, so its performance is still good during the pre-copy phase, as we will show later.

Our live VM cloning scheme is similar to previous VM/process migration systems [13, 51]. In the pre-copy phase, the dirty memory pages of the source VM are copied iteratively in multiple rounds without stopping the

source VM. A stop-and-copy phase, where the source VM is paused temporarily, is used for transferring the remaining dirty pages. A typical pause is within 1 second.

AGILE also performs disk cloning to make the new VM independent of the source VM. In IaaS clouds, the VM's disk is typically located on a networked storage device. Because a full disk image is typically large and would take a long time to copy, AGILE performs *incremental disk cloning* using QEMU Copy On Write (QCOW). When we pause the source VM to perform the final round of memory copy, we make the disk image of the source VM a read-only base image, and build two incremental (copy-on-write) images for the source VM and the new VM. We can associate the new incremental image with the source VM on-the-fly without restarting the VM by redirecting the disk image driver at the hypervisor level. This is transparent to the guest OS of the source VM.

Because live VM cloning makes the new VM instance inherit all the state from the source VM, which includes the IP address, the new VM may immediately send out network packets using the same IP address as the source VM, causing duplicate network packets and application errors. To avoid this, AGILE first disconnects the network interface of the new VM, clears the network buffer, and then reconnects the network interface of the new VM with a new IP address.

AGILE introduces two features to live VM cloning.

Adaptive copy rate configuration. AGILE uses the minimum copy rate that can finish the cloning before the overload is predicted to start (T_o), and adjusts this dynamically based on how much data needs to be transferred. This uses the minimal network bandwidth, and minimizes impact on the source machine and application.

If the new application server configuration takes T_{config} seconds, the cloning must finish within $T_{clone} = T_o - T_{config}$. Intuitively, the total size of transferred memory should equal the original memory size plus the amount of memory that is modified while the cloning is taking place. Suppose the VM is using M memory pages, and the desired copy rate is r_{page_copy} pages per second. We have: $r_{page\_copy} \times T_{clone} = M + r_{dirty} \times T_{clone}$. From this, we have: $r_{page\_copy} = M/T_{clone} + r_{dirty}$. To estimate the page-dirty rate, we continuously sample the actual page-dirtying rate and use an exponential moving average of these values as the estimated value. AGILE will also adjust the copy rate if the predicted overload time T_o changes.

Event-driven application auto-configuration. AGILE allows VMs to subscribe to critical events that occur during the live cloning process to achieve auto-configuration. For example, the new VM can subscribe to the *NetworkConfigured* event so that it can configure itself to use its new IP address. The source VM can subscribe to the *Stopping* event that is triggered when the cloning enters the stop-and-copy phase, so that it can notify a front-end load balancer to buffer some user requests (e.g., write requests). Each VM image is associated with an XML configuration file specifying what to invoke on each cloning event.

Minimizing unhelpful cloning. Since live cloning takes resources, we want to avoid triggering unnecessary cloning on transient workload spikes: AGILE will only trigger cloning if the overload is predicted more than k (e.g. k=3) consecutive times. Similarly, AGILE cancels cloning if the overload is predicted to be gone more than k consecutive times. Furthermore, if the overload state will end before the new VM becomes ready, we should not trigger cloning.

To do this, AGILE checks whether an overload condition will appear in the look ahead window $[t, t+W]$. We want to ignore those transient overload states that will be gone before the cloning can be completed. Let $T_{RML} < W$ denote the required minimum lead time that AGILE's predictor needs to raise an alert in advance for the cloning to complete before the system enters the overload state. AGILE will ignore those overload alarms that only appear in the window $[t, t+T_{RML}]$ but disappear in the window $[t+T_{RML}, t+W]$. Furthermore, cloning is triggered only if the overload state is predicted to last for at least Q seconds in the window $[t+T_{RML}, t+W]$ $(0 < Q \leq W - T_{RML})$.

The least-loaded server in the pool is used as the source VM to be cloned. AGILE also supports concurrent cloning where it creates multiple new servers at the same time. Different source servers are used to avoid overloading any one of them.

Online prediction algorithms can raise false alarms. To address this issue, AGILE continuously checks whether previously predicted overload states still exist. Intuitively, as the system approaches the start of the overload state, the prediction should become more accurate. If the overload state is no longer predicted to occur, the cloning operation will be canceled; if this can be done during the pre-copy phase, it won't affect the application or the source VM.

3 Experimental evaluation

We implemented AGILE on top of the KVM virtualization platform, in which each VM runs as a KVM process. This lets AGILE monitor the VM's resource usage through the Linux `/proc` interface. AGILE periodically samples system-level metrics such as CPU consumption, memory allocation, network traffic, and disk I/O statistics. To implement pre-copy live cloning, we modified KVM to add a new KVM hypervisor mod-

ule and an interface in the `KVM monitor` that supports starting, stopping a clone, and adjusting the memory copy rate. AGILE controls the resources allocated to application VMs through the Linux `cgroups` interface.

We evaluated our KVM implementation of AGILE using the RUBiS online auction benchmark (PHP version) [38] and the Apache Cassandra key-value store 0.6.13 [4]. We also tested our prediction algorithm using Google cluster data [20]. This section describes our experiments and results.

3.1 Experiment methodology

Our experiments were conducted on a cloud testbed in our lab with 10 nodes. Each cloud node has a quad-core Xeon 2.53GHz processor, 8GiB memory and 1Gbps network bandwidth, and runs 64 bit CentOS 6.2 with KVM 0.12.1.2. Each guest VM runs 64 bit CentOS 5.2 with one virtual CPU core and 2GiB memory. This setup is enough to host our test benchmarks at their maximum workload.

Our experiments on RUBiS focus on the CPU resource, as that appears to be the bottleneck in our setup since all the RUBiS components have low memory consumption. To evaluate AGILE under workloads with realistic time variations, we used one day of per-minute workload intensity observed in 4 different real world web traces [24] to modulate the request rate of the RUBiS benchmark: (1) World Cup 98 web server trace starting at 1998-05-05:00.00; (2) NASA web server trace beginning at 1995-07-01:00.00; (3) EPA web server trace starting at 1995-08-29:23.53; and (4) ClarkNet web server trace beginning at 1995-08-28:00.00. These traces represent realistic load variations over time observed from well-known web sites. The resource usage is collected every 2 seconds. We perform fine-grained sampling for precise resource usage prediction and effective scaling [43]. Although the request rate is changed every minute, the resource usage may still change faster because different types of requests are generated.

At each sampling instant t, the resource demand prediction module uses a sliding window of size D of recent resource usage (i.e., from $t-D$ to t) and predicts future resource demands in the look-ahead window W (i.e., from t to $t+W$). We repeat each experiment 6 times.

We also tested our prediction algorithm using real system resource usage data collected on a Google cluster [20] to evaluate its accuracy on predicting machine overloads. To do this, we extracted CPU and memory usage traces from 100 machines randomly selected from the Google cluster data. We then aggregate the resource usages of all the tasks running on a given machine to get the usage for that machine. These traces represent various realistic workload patterns. The sampling interval in the Google cluster is 5 minutes and the trace lasts 29 days.

Parameter	RUBiS	Google data
Input data window (D)	6000 seconds	250 hours
Look-ahead window (W)	120 seconds	5 hours
Sampling interval (T_s)	2 seconds	5 minutes
Total trace length	one day	29 days
Overload duration threshold (Q)	20 seconds	25 minutes
Response time SLO	100 ms	NA

Table 1: Summary of parameter values used in our experiments.

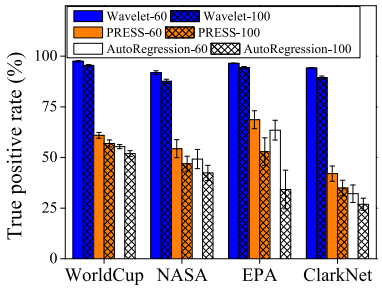
Table 1 shows the parameter values used in our experiments. We also performed comparisons under different threshold values by varying D, W, and Q, which show similar trends. Note that we used consistently larger D, W, and Q values for the Google trace data because the sampling interval of the Google data (5 minutes) is significantly larger than what we used in the RUBiS experiments (2 seconds).

To evaluate the accuracy of our wavelet-based prediction scheme, we compare it against the best alternatives we could find: PRESS [19] and auto-regression [9]. These have been shown to achieve higher accuracy and lower overheads than other alternatives. We calculate the overload-prediction accuracy as follows. The predictor is deemed to raise a valid overload alarm if the overload state (e.g., when the resource pressure is bigger than the overload threshold) is predicted earlier than the required minimum lead time (T_{RML}). Otherwise, we call the prediction a false negative. Note that we only consider those overload states that last at least Q seconds (Section 2.3). Moreover, we require that the prediction model accurately estimates when the overload will start, so we compare the predicted alarm time with the true overload start time to calculate a *prediction time error*. If the absolute prediction time error is small (i.e., $\leq 3 \cdot T_s$), we say the predictor raises a correct alarm. Otherwise, we say the predictor raises a false alarm.

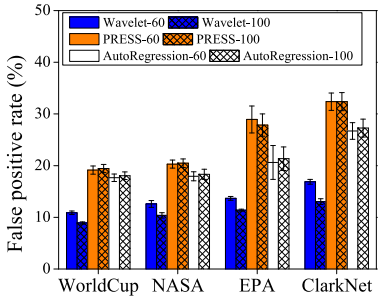
We use the standard metrics, *true positive rate* (A_T) and *false positive rate* (A_F), given in equation 1. P_{true}, P_{false}, N_{true}, and N_{false} denote the number of true positives, false positives, true negatives, and false negatives, respectively.

$$A_T = \frac{P_{true}}{P_{true}+N_{false}}, \; A_F = \frac{P_{false}}{P_{false}+N_{true}} \qquad (1)$$

A service provider can either rely on the application itself or an external tool [5] to tell whether the application SLO is being violated. In our experiments, we adopted the latter approach. With the RUBiS benchmark, the workload generator tracks the response time of the HTTP requests it makes. The SLO violation rate is the fraction



(a)



(b)

Figure 5: CPU demand prediction accuracy comparison for RUBiS web server driven by one-day request traces of different real web servers with T_{RML} = 60 and 100 seconds.

of requests that have response time larger than a predefined SLO threshold. In our experiments, this was 100ms, the 99th percentile of observed response times for a run with no resource constraints. We conduct our RUBiS experiments on both the Apache web server tier and the MySQL database tier.

For comparison, we also implemented a set of alternative resource provisioning schemes:

- *No scaling*: A non-elastic resource provisioning scheme that cannot change the size of the server pool, which is fixed at 1 server as this is sufficient for the average resource demand.
- *Reactive*: This scheme triggers live VM cloning when it observes that the application has become overloaded. It uses a fixed memory-copy rate, and for a fair comparison, we set this to the average copy rate used by AGILE so that both schemes incur a similar network cost for cloning.
- *PRESS*: Instead of using the wavelet-based prediction algorithm, PRESS uses a Markov+FFT resource demand prediction algorithm [19] to predict future overload state and triggers live cloning when an overload state is predicted to occur. PRESS uses the same false alarm filtering mechanism described in Section 2.3.
- *FixThreshold-65% and -80%*: This scheme triggers live VM cloning if the resource pressure exceeds 65% and 80%. This allows us to evaluate the effects of the resource pressure model.

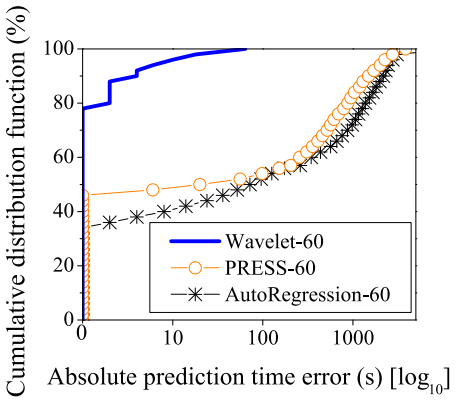


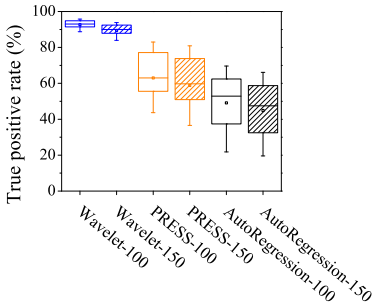
Figure 6: Cumulative distribution function of the prediction time error for the RUBiS web server driven by the ClarkNet workload.

Note that the *reactive* and *PRESS* schemes use the AGILE same resource pressure model to decide the resource pressure threshold for the target 5% SLO violation rate.

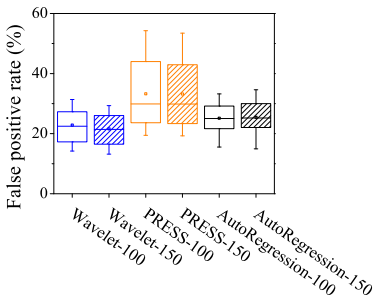
3.2 Experimental results

Prediction accuracy results. In this set of experiments, no cloning is performed. Figure 5 shows the overload prediction accuracy comparisons for RUBiS driven by different real workload traces. We test the prediction system with different lead time requirements (T_{RML}). The results show that our wavelet prediction scheme is statistically significantly better than the PRESS scheme and the auto-regression scheme (the independent two-sample t-test indicates $p\text{-}value \leq 0.01$). Particularly, the wavelet scheme can improve the true positive rate by up to 3.42× and reduce the false positive rate by up to 0.41×. The accuracy of the PRESS and auto-regression schemes suffers as the number of iterations increases, errors accumulate, and the correlation between the prediction model and the actual resource demand becomes weaker. This is especially so for ClarkNet, the most dynamic of the four traces.

In the above prediction accuracy figure, we consider the predictor raises a correct alarm if the absolute prediction time error is less than $\leq 3 \cdot T_s$. We further compare the distributions of the absolute prediction time error among different schemes. Figure 6 compares the cumulative distribution functions of the absolute prediction time error among different schemes. We observe that AGILE achieves much lower prediction time error (78% alarms have 0 absolute prediction time

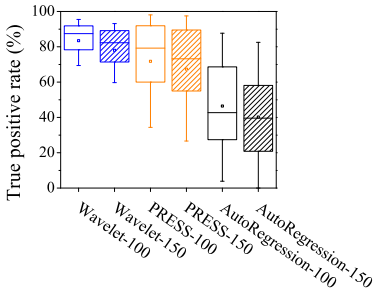


(a)

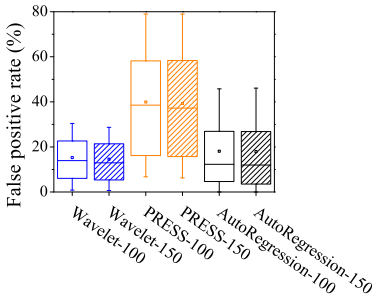


(b)

Figure 7: Prediction accuracy for 100 Google cluster CPU traces with T_{RML} = 100 and 150 minutes. The bottom and top of the box represent 25th and 75th percentile values, the ends of the whiskers represent 10th and 90th percentile values.



(a)



(b)

Figure 8: Prediction accuracy comparison for 100 Google cluster memory traces.

error) than auto-regression (34% alarms have 0 absolute prediction time error) and PRESS (46% alarms have 0 absolute prediction time error). Other traces show similar trend, which are omitted due to space limitation.

Figure 7 and Figure 8 show the prediction accuracy for the CPU and memory usage traces on 100 machines in a Google cluster. The overload threshold is set to the 70th percentile of all values in each trace. We observe that the wavelet scheme again consistently outperforms the PRESS scheme and the auto-regression scheme with up to 2.1× better true positive rate and 0.34× the false positive rate.

Overload handling results. Next, we evaluate how well AGILE handles overload using dynamic server pool scaling. The experiment covers 7000 seconds of a RUBiS run driven by the ClarkNet web server trace. The first 6000 seconds are used for training and no cloning is performed. The overload state starts at about t = 6500s. When examining the effects of scaling on different tiers in RUBiS, we limit the scaling to one tier and allocate sufficient resources to the other tier. We repeat each experiment 3 times.

Figure 9 shows the overall results of different schemes. Overall SLO violation *rate* denotes the percentage of requests that have response times larger than the SLO violation threshold (e.g., 100ms) during the experiment run. SLO violation *time* is the total time in which SLO violation rate (collected every 5 seconds) exceeds the target (e.g., 5%). We observe that AGILE consistently achieves the lowest SLO violation rate and shortest SLO violation time. Under the *no scaling* scheme, the application suffers from high SLO violation rate and long SLO violation time in both the web server tier and the database tier scaling experiments. The *reactive* scheme mitigates this by triggering live cloning to create a new server after the overload condition is detected, but since the application is already overloaded when the scaling is triggered, the application still experiences a high SLO violation rate for a significant time. The *FixThreshold-80%* scheme triggers the scaling too late, especially in the database experiment and thus does not show any noticeable improvement compared to without scaling. Using a lower threshold, *FixThreshold-65%* improves the SLO violation rate but at a higher resource cost: resource pressure is maintained at 65% while AGILE maintains the resource pressure at 75%. In contrast, AGILE predicts the overload state in advance, and successfully completes live cloning before the application enters the overload state. With more accurate predictions, AGILE also outperforms PRESS by predicting the overload sooner.

Figure 10 shows detailed performance measurements

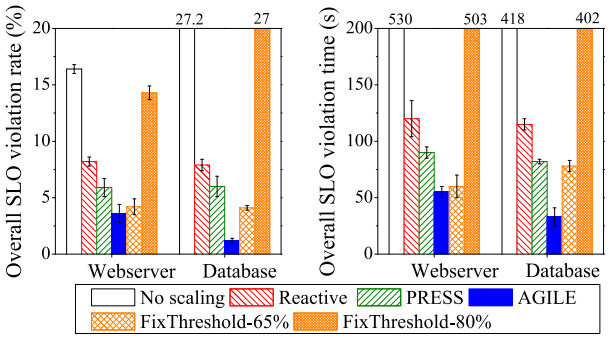


Figure 9: SLO violation rates and times for the two RUBiS tiers under a workload following the ClarkNet trace.

Application	In use	Copied	Ratio
RUBiS Webserver	530MiB	690MiB	1.3×
RUBiS Database	1092MiB	1331MiB	1.2×
Cassandra	671MiB	1001MiB	1.5×

Table 2: Amount of memory moved during cloning for different applications.

for the web server tier during the above experiment. We sample the average response time every second and plot the cumulative distribution functions for the whole run and during cloning. From Figure 10(a), we can see that the response time for most requests meets the SLO when using the AGILE system. In contrast, if no scaling is performed, the application suffers from a significant increase in response time. Figure 10(b) shows that all the scaling schemes, except AGILE, cause much worse performance during the cloning process: the application is overloaded and many requests suffer from a large response time until a new server is started. In contrast, using AGILE, the application experiences little response time increase since the application has not yet entered the overload state. Figure 11 shows the equivalent results for the database server and has similar trends.

Figure 12 and Figure 13 show the SLO violation rate timeline of RUBiS application under the ClarkNet workload. Compared to other schemes, AGILE triggers scaling before the system enters the overload state. Under the reactive scheme, the live cloning is executed when the system is already overloaded, which causes a significant impact to the application performance during the cloning time. Although PRESS can predict the overload state in advance, the lead time is not long enough for cloning to finish before the application is overloaded.

Dynamic copy-rate configuration results. Table 2 shows the amount of memory moved during cloning for different applications. AGILE moved at most 1.5 times the amount of the memory in use at the source VM. We also tested AGILE under different overload pending

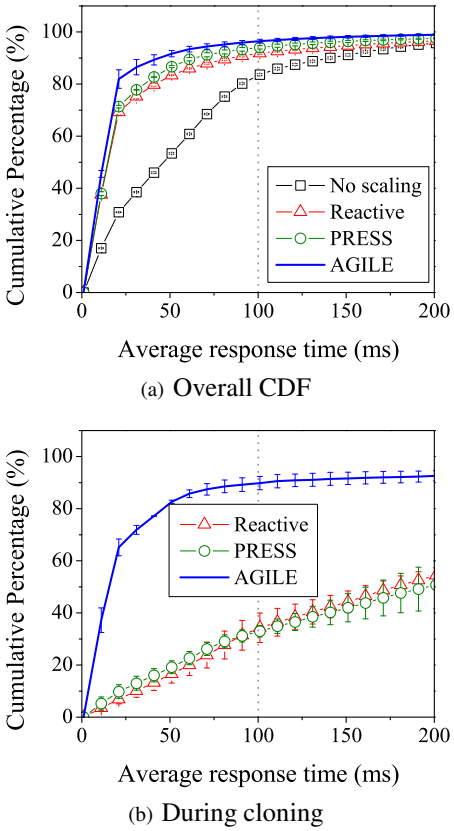


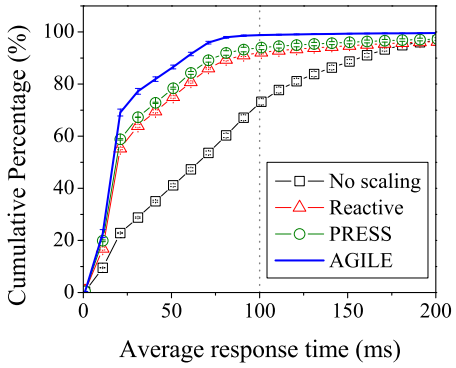
Figure 10: Scaling up the RUBiS web server tier from 1 server to 2 servers under a dynamic workload following the ClarkNet trace. *(a) Overall CDF* denotes the whole experiment. *(b) During cloning* denotes the period in which the scaling is being executed. AGILE always triggers scaling earlier than other schemes.

time deadlines (i.e., target time to finish cloning) and check whether the cloning can finish within the pending time. Figure 14 shows that our dynamic copy-rate setting can accurately control the cloning time under different deadlines.

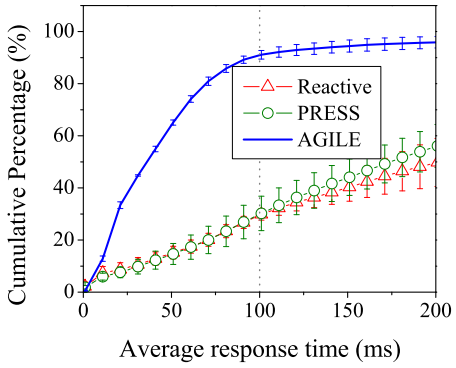
We measured the time spent in the different stages of the live VM cloning for different applications (Table 3). As expected, pre-copy dominates the cloning time (tens of seconds), while the stop-and-copy time is only 0.1 s, so the downtime of the source VM is negligible.

Overhead results. We first present the overhead imposed by our online profiling mechanism. Figure 15 shows the timeline of the average response time during profiling. Figure 16 shows the performance impact of the online profiling on the average response time over the period of 6 hours, in which AGILE performs profiling three times. Overall, the overhead measurements show that AGILE is practical for online system management.

We also evaluated the overhead of the AGILE system. The AGILE slave process on each cloud node imposes



(a) Overall CDF



(b) During cloning

Figure 11: Scaling up the RUBiS database server tier from 1 server to 2 servers under a dynamic workload following the ClarkNet trace. We used 9 web servers to make the database tier become the bottleneck.

Application	Pre-copy	Stop-and-copy	Configuration
RUBiS Webserver	31.2 ± 1.1 s	0.10 ± 0.01 s	16.8 ± 0.6 s
RUBiS Database	33.1 ± 0.9 s	0.10 ± 0.01 s	17.8 ± 0.8 s
Cassandra	31.5 ± 1.1 s	0.10 ± 0.01 s	17.5 ± 0.9 s

Table 3: Time spent in the different stages of live VM cloning.

less than 1% CPU overhead. The most computationally intensive component is the prediction module that runs on the master node. Table 4 shows the online training time and prediction time for AGILE, PRESS, and auto-regression schemes. AGILE has similar overheads at the master node as does PRESS. The auto-regression scheme is faster, however its accuracy is much worse than AGILE. Clearly, these costs still need to be reduced (e.g., by incremental retraining mechanisms and decentralized masters), and we hope to work on this in the future.

4 Related Work

AGILE is built on top of previous work on resource demand prediction, performance modeling, and VM

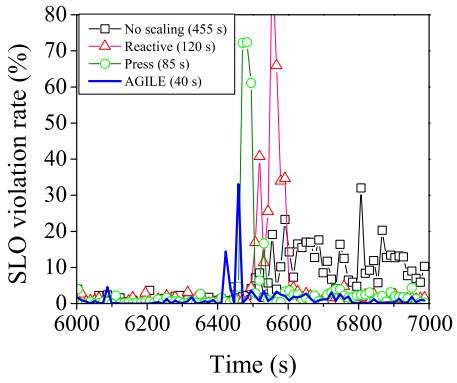


Figure 12: SLO violation timeline for web server tier experiment under the ClarkNet workload. The number in the bracket indicates the SLO violation time in seconds.

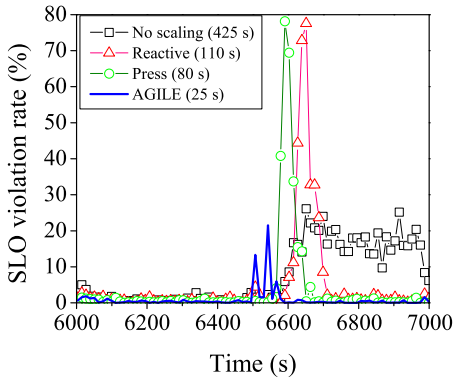


Figure 13: SLO violation timeline for database tier experiment under the ClarkNet workload.

cloning. Most previous work on server pool scaling (e.g., [29, 17]) adopts a *reactive* approach while AGILE provides a *prediction-driven* solution that allows the system to start new instances before SLO violation occurs.

Previous work has proposed white-box or grey-box approaches to addressing the problem of cluster sizing. Elastisizer [22] combines job profiling, black-box and white-box models, and simulation to compute an optimal cluster size for a specific MapReduce job. Verma et al. [47] proposed a MapReduce resource sizing framework that profiles the application on a smaller data set and applies linear regression scaling rules to generate a set of resource provisioning plans. The SCADS director framework [44] used a model-predictive control (MPC) framework to make cluster sizing decisions based on the current workload state, current data layout, and predicted SLO violation. Huber et al. [23] presented a self-adaptive resource management algorithm which leverages workload prediction and a performance model [7] that predicts application's performance

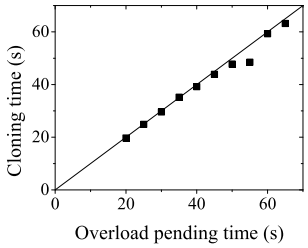


Figure 14: Cloning time achieved against predicted time to overload.

Scheme	Training time (3000 samples)	Prediction time (60 steps)
AGILE	575 ± 7 ms	2.2 ± 0.1 ms
PRESS	595 ± 6 ms	1.5 ± 0.1 ms
Auto-regression	168 ± 5 ms	2.2 ± 0.1 ms

Table 4: Prediction model training time and the prediction time comparison between AGILE, PRESS, and auto-regression schemes. The prediction module runs on the master host.

under different configurations and workloads. In contrast, AGILE does not require any prior application knowledge.

Previous work [53, 26, 35, 36, 34, 29] has applied control theory to achieve adaptive resource allocation. Such approaches often have parameters that need to be specified or tuned offline for different applications or workloads. The feedback control system also requires a feedback signal that is stable and well correlated with SLO measurement. Choosing suitable feedback signals for different applications is a non-trivial task [29]. Other projects used statistical learning methods [41, 42, 15, 40] or queueing theory [46, 45, 14] to estimate the impact of different resource allocation policies. Overdriver [48] used offline profiling to learn the memory overload probability of each VM to select different mitigation strategies: using migration for sustained overloads or network memory for transient overloads. Those models need to be built and calibrated in advance. Moreover, the resource allocation system needs to make certain assumptions about the application and the running platform (e.g., input data size, cache size, processor speed), which often is impractical in a virtualized, multi-tenant IaaS cloud system.

Trace-driven resource demand prediction has been applied to several dynamic resource allocation problems. Rolia et al. [37] described a resource demand prediction scheme that multiplies recent resource usage by a burst factor to provide some headroom. Chandra et al. [11] developed a prediction framework based on auto-regression to drive dynamic resource allocation decisions. Gmach et al. [18] used a Fourier transform-based scheme to perform offline extraction of long-term cyclic workload patterns. Andrzejak et al. [3] employed a

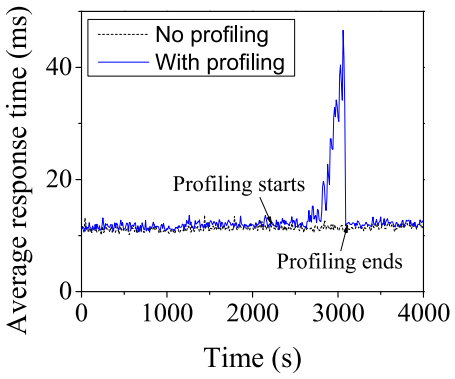


Figure 15: The effect of profiling on average response time for the RUBiS system under the ClarkNet workload.

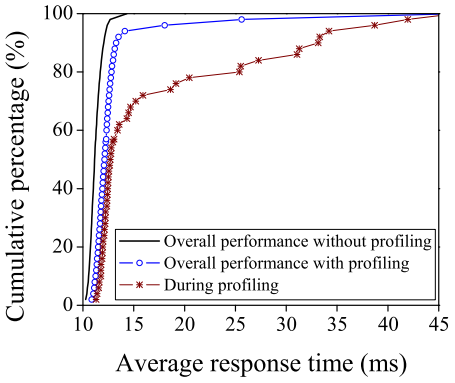


Figure 16: Profiling overhead for the RUBiS system under the ClarkNet workload. Profiling occurs every two hours.

genetic algorithm and fuzzy logic to address the problem of having little training data. Gandhi et al. [16] combined long-term predictive provisioning using periodic patterns with short-term reactive provisioning to minimize SLO violations and energy consumption. Matsunaga et al. [30] investigated several machine learning techniques for predicting spatio-temporal resource utilization. PRESS [19] developed a hybrid online resource demand prediction model that combines a Markov model and a fast Fourier transform-based technique. Previous prediction schemes either focus on short-term prediction or need to assume cyclic workload patterns. In contrast, AGILE focuses on medium-term prediction and works for arbitrary workload patterns.

VM cloning has been used to support elastic cloud computing. SnowFlock [28] provides a fast VM instantiation scheme using on-demand paging. However, the new instance suffers from an extended performance warmup period while the working set is copied over from the origin. Kaleidoscope [8] uses fractional VM cloning with VM state coloring to prefetch semantically-related regions. Although our current prototype uses full pre-copy, AGILE could readily work with fractional pre-

copy too: prediction-driven live cloning and dynamic copy rate adjustment can be applied to both cases. Fractional pre-copy could be especially useful if the overload duration is predicted to be short. Dolly [10] proposed a proactive database provisioning scheme that creates a new database instance in advance from a disk image snapshot and replays the transaction log to bring the new instance to the latest state. However, Dolly did not provide any performance predictions, and the new instance created from an image snapshot may need some warmup time. In contrast, the new instance created by AGILE can reach its peak performance immediately after start.

Local resource scaling (e.g., [39]) or live VM migration [13, 50, 49, 25] can also relieve local, per-server application overloads, but distributed resource scaling will be needed if the workload exceeds the maximum capacity of any single server. Although previous work [39, 50] has used overload prediction to proactively trigger local resource scaling or live VM migration, AGILE addresses the specific challenges of using predictions in distributed resource scaling. Compared to local resource scaling and migration, cloning requires longer lead time and is more sensitive to prediction accuracy, since we need to pay the cost of maintaining extra servers. AGILE provides medium-term predictions to tackle this challenge.

5 Future Work

Although AGILE showed its practicality and efficiency in experiments, there are several limitations which we plan to address in our future work.

AGILE currently derives resource pressure models for just CPU. Our future work will extend the resource pressure model to consider other resources such as memory, network bandwidth, and disk I/O. There are two ways to build a multi-resource model. We can build one resource pressure model for each resource separately or build a single resource pressure model incorporating all of them. We plan to explore both approaches and compare them.

AGILE currently uses resource capping (a Linux `cgroups` feature) to achieve performance isolation among different VMs [39]. Although we observed that the resource capping scheme works well for common bottleneck resources such as CPU and memory, there may still exist interference among co-located VMs [52]. We need to take such interference into account to build more precise resource pressure models and achieve more accurate overload predictions.

Our resource pressure model profiling can be triggered either periodically or by workload mix changes. To make AGILE more intelligent, we plan to incorporate workload change detection mechanism [32, 12] in AGILE. Upon detecting a workload change, AGILE starts a new profiling phase to build a new resource pressure model for the current workload type.

6 Conclusion

AGILE is an application-agnostic, prediction-driven, distributed resource scaling system for IaaS clouds. It uses wavelets to provide medium-term performance predictions; it provides an automatically-determined model of how an application's performance relates to the resources it has available; and it implements a way of cloning VMs that minimizes application startup time. Together, these allow AGILE to predict performance problems far enough in advance that they can be avoided.

To minimize the impact of cloning a VM, AGILE copies memory at a rate that completes the clone just before the new VM is needed. AGILE performs continuous prediction validation to detect false alarms and cancels unnecessary cloning.

We implemented AGILE on top of the KVM virtualization platform, and conducted experiments under a number of time-varying application loads derived from real-life web workload traces and real resource usage traces. Our results show that AGILE can significantly reduce SLO violations when compared to existing resource scaling schemes. Finally, AGILE is lightweight, which makes it practical for IaaS clouds.

7 Acknowledgement

This work was sponsored in part by NSF CNS0915567 grant, NSF CNS0915861 grant, NSF CAREER Award CNS1149445, U.S. Army Research Office (ARO) under grant W911NF-10-1-0273, IBM Faculty Awards and Google Research Awards. Any opinions expressed in this paper are those of the authors and do not necessarily reflect the views of NSF, ARO, or U.S. Government.

References

[1] N. A. Ali and R. H. Paul. *Multiresolution signal decomposition*. Academic Press, 2000.

[2] Amazon Elastic Compute Cloud. `http://aws.amazon.com/ec2/`.

[3] A. Andrzejak, S. Graupner, and S. Plantikow. Predicting resource demand in dynamic utility computing environments. In *Autonomic and Autonomous Systems*, 2006.

[4] Apache Cassandra Database. `http://cassandra.apache.org/`.

[5] M. Ben-Yehuda, D. Breitgand, M. Factor, H. Kolodner, V. Kravtsov, and D. Pelleg. NAP: a building block for remediating performance bottlenecks via black box network analysis. In *ICAC*, 2009.

[6] E. Brigham and R. Morrow. The fast Fourier transform. *IEEE Spectrum*, 1967.

[7] F. Brosig, N. Huber, and S. Kounev. Automated extraction of architecture-level performance models of distributed component-based systems. In *Automated Software Engineering*, 2011.

[8] R. Bryant, A. Tumanov, O. Irzak, A. Scannell, K. Joshi, M. Hiltunen, A. Lagar-Cavilla, and E. de Lara. Kaleidoscope: cloud micro-elasticity via VM state coloring. In *EuroSys*, 2011.

[9] E. S. Buneci and D. A. Reed. Analysis of application heartbeats: Learning structural and temporal features in time series data for identification of performance problems. In *Supercomputing*, 2008.

[10] E. Cecchet, R. Singh, U. Sharma, and P. Shenoy. Dolly: virtualization-driven database provisioning for the cloud. In *VEE*, 2011.

[11] A. Chandra, W. Gong, and P. Shenoy. Dynamic resource allocation for shared data centers using online measurements. In *IWQoS*, 2003.

[12] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni. Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change. In *Dependable Systems and Networks*, 2008.

[13] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI*, 2005.

[14] R. P. Doyle, J. S. Chase, O. M. Asad, W. Jin, and A. M. Vahdat. Model-based resource provisioning in a web service utility. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[15] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting multiple metrics for queries: better decisions enabled by machine learning. In *International Conference on Data Engineering*, 2009.

[16] A. Gandhi, Y. Chen, D. Gmach, M. Arlitt, and M. Marwah. Minimizing data center sla violations and power consumption via hybrid resource provisioning. In *Green Computing Conference and Workshops*, 2011.

[17] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. In *Transactions on Computer Systems*, 2012.

[18] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper. Capacity management and demand prediction for next generation data centers. In *International Conference on Web Services*, 2007.

[19] Z. Gong, X. Gu, and J. Wilkes. PRESS: PRedictive Elastic ReSource Scaling for cloud systems. In *International Conference on Network and Service Management*, 2010.

[20] Google cluster-usage traces: format + scheme (2011.11.08 external). `http://goo.gl/5uJri`.

[21] N. R. Herbst, N. Huber, S. Kounev, and E. Amrehn. Self-adaptive workload classification and forecasting for proactive resource provisioning. In *International Conference on Performance Engineering*, 2013.

[22] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *SoCC*, 2011.

[23] N. Huber, F. Brosig, and S. Kounev. Model-based self-adaptive resource allocation in virtualized environments. In *Software Engineering for Adaptive and Self-Managing Systems*, 2011.

[24] The IRCache Project. `http://www.ircache.net/`.

[25] C. Isci, J. Liu, B. Abali, J. Kephart, and J. Kouloheris. Improving server utilization using fast virtual machine migration. In *IBM Journal of Research and Development*, 2011.

[26] E. Kalyvianaki, T. Charalambous, and S. Hand. Self-adaptive and self-configured CPU resource provisioning for virtualized servers using Kalman filters. In *ICAC*, 2009.

[27] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the linux virtual machine monitor. In *Linux Symposium*, 2007.

[28] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. SnowFlock: rapid virtual machine cloning for cloud computing. In *EuroSys*, 2009.

[29] H. C. Lim, S. Babu, and J. S. Chase. Automated control for elastic storage. In *ICAC*, 2010.

[30] A. Matsunaga and J. Fortes. On the use of machine learning to predict the time and resources consumed by applications. In *Cluster, Cloud and Grid Computing*, 2010.

[31] A. Neogi, V. R. Somisetty, and C. Nero. Optimizing the cloud infrastructure: tool design and a case study. *International IBM Cloud Academy Conference*, 2012.

[32] H. Nguyen, Z. Shen, Y. Tan, and X. Gu. FChain: Toward black-box online fault localization for cloud systems. In *ICDCS*, 2013.

[33] Oracle. Best practices for database consolidation in private clouds, 2012. `http://www.oracle.com/technetwork/database/focus-areas/database-cloud/database-cons-best-practices-1561461.pdf`.

[34] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated

control of multiple virtualized resources. In *EuroSys*, 2009.

[35] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *EuroSys*, 2007.

[36] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus. Using control theory to achieve service level objectives in performance management. In *Real-Time Systems*, 2002.

[37] J. Rolia, L. Cherkasova, M. Arlitt, and V. Machiraju. Supporting application quality of service in shared resource pools. *Communications of the ACM*, 2006.

[38] RUBiS Online Auction System. `http://rubis.ow2.org/`.

[39] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. CloudScale: elastic resource scaling for multi-tenant cloud systems. In *SoCC*, 2011.

[40] P. Shivam, S. Babu, and J. Chase. Active and accelerated learning of cost models for optimizing scientific applications. In *VLDB*, 2006.

[41] P. Shivam, S. Babu, and J. S. Chase. Learning application models for utility resource planning. In *ICAC*, 2006.

[42] C. Stewart, T. Kelly, A. Zhang, and K. Shen. A dollar from 15 cents: cross-platform management for internet services. In *USENIX ATC*, 2008.

[43] Y. Tan, V. Venkatesh, and X. Gu. Resilient self-compressive monitoring for large-scale hosting infrastructures. In *TPDS*, 2012.

[44] B. Trushkowsky, P. Bodík, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. The SCADS director: scaling a distributed storage system under stringent performance requirements. In *FAST*, 2011.

[45] B. Urgaonkar and A. Chandra. Dynamic provisioning of multi-tier internet applications. In *ICAC*, 2005.

[46] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An analytical model for multi-tier internet services and its applications. In *SIGMETRICS*, 2005.

[47] A. Verma, L. Cherkasova, and R. Campbell. Resource provisioning framework for MapReduce jobs with performance goals. In *Middleware*, 2011.

[48] D. Williams, H. Jamjoom, Y. Liu, and H. Weatherspoon. Overdriver: Handling memory overload in an oversubscribed cloud. In *VEE*, 2011.

[49] D. Williams, H. Jamjoom, and H. Weatherspoon. The Xen-Blanket: virtualize once, run everywhere. In *Eurosys*, 2012.

[50] T. Wood, P. J. Shenoy, A. Venkataramani, and M. S. Yousif. Black-box and gray-box strategies for virtual machine migration. In *NSDI*, 2007.

[51] E. Zayas. Attacking the process migration bottleneck. In *SOSP*, 1987.

[52] X. Zhang, E. Tune, R. Hagmann, R. J. V. Gokhale, and J. Wilkes. *CPI*^{2}: CPU performance isolation for shared compute clusters. In *Eurosys*, 2013.

[53] X. Zhu, D. Young, B. J. Watson, Z. Wang, J. Rolia, S. Singhal, B. McKee, C. Hyser, D. Gmach, R. Gardner, T. Christian, and L. Cherkasova. 1000 Islands: integrated capacity and workload management for the next generation data center. In *ICAC*, 2008.