The SCADS Director: Scaling a Distributed Storage System Under Stringent Performance Requirements

Beth Trushkowsky, Peter Bodík, Armando Fox,
Michael J. Franklin, Michael I. Jordan, David A. Patterson
{*trush, bodik, fox, franklin, jordan, pattrsn*}*@eecs.berkeley.edu*
University of California, Berkeley

Abstract

Elasticity of cloud computing environments provides an economic incentive for automatic resource allocation of stateful systems running in the cloud. However, these systems have to meet strict performance Service-Level Objectives (SLOs) expressed using upper percentiles of request latency, such as the 99th. Such latency measurements are very noisy, which complicates the design of the dynamic resource allocation. We design and evaluate the SCADS Director, a control framework that reconfigures the storage system on-the-fly in response to workload changes using a performance model of the system. We demonstrate that such a framework can respond to both unexpected data hotspots and diurnal workload patterns without violating strict performance SLOs.

1 Introduction

Cloud computing has emerged as a preferred technology for delivering large-scale internet applications, in part because its elasticity provides the ability to dynamically provision and reclaim resources in response to fluctuations in workload. As cloud environments and their applications expand in scale and complexity, it becomes increasingly important to automate such dynamic resource allocation.

Techniques for automatically scaling stateless systems such as web servers or application servers are fairly well understood. However, many applications that can most benefit from elasticity, such as social networking, e-commerce and auction sites, are both data-intensive and interactive. Such applications present three major challenges for automatic scaling.

First, in most data-intensive services, a request for a specific data item can only be satisfied by a copy of that particular data item, so not every server can handle every request, which complicates load balancing. Second, interactivity means that a successful application must provide highly-responsive, low-latency service to the vast majority of users: a typical Service Level Objective (SLO) might be expressed as "99% of all requests must be answered within 100ms" [20, 17]. Third, the workloads presented by large-scale applications can be highly volatile, with quickly-occurring unexpected spikes (due to flash crowds) and diurnal fluctuations.

This "perfect storm" of statefulness, workload volatility and stringent performance requirements complicates the development of automatic scaling mechanisms. To scale a data-intensive system, data items must be moved (i.e., partitioned or coalesced) or copied (i.e., replicated) among the nodes of the system. Such data movement takes time and can place additional load on an already overloaded system. Provisioning of new nodes incurs significant start-up delay, so decisions must be made early to react effectively to workload changes. But most importantly, the SLOs on upper percentile latency significantly complicate the problem compared to requirements based on average latency, as statistical estimates based on observations in the upper percentiles of the latency distribution have higher variance than estimates obtained from the center of the distribution. This variance is exacerbated by "environmental" application noise uncorrelated to particular queries or data items [19]. The resulting noisy latency signal can cause oscillations in classical closed-loop control [7].

In this paper we describe the design of a control framework for dynamically scaling distributed storage systems that addresses these challenges. Our approach leverages key features of modern distributed storage systems and uses a performance model coupled with workload statistics to predict whether each server is likely to continue to meet its SLO. Based on this model, the framework moves and replicates data as necessary. In particular, we make the following contributions:

- We identify the challenges and opportunities that arise in designing dynamic resource allocation frameworks for stateful systems that maintain performance SLOs on upper quantiles of request latency.

To appear in FAST 2011

- We describe the design and implementation of a modular control framework based on Model-Predictive Control [30] that addresses these challenges.
- We evaluate the effectiveness of the control framework through experiments using a storage system running on Amazon's Elastic Compute Cloud (EC2), using workloads that exhibit both periodic and erratic fluctuations comparable to those observed in production systems.

The rest of the paper proceeds as follows. Section 2 describes background and challenges, and Section 3 discusses the design considerations that address those challenges. Related work is in Section 4. Section 5 details the implementation of our control framework, and Section 6 demonstrates experimental results of the control framework using Amazon's EC2. Further discussion is in Section 7, and we remark on future work and conclude in Sections 8 and 9.

2 Scaling Challenges

2.1 Background

We address dynamic resource allocation for distributed storage systems for which the performance SLO is specified using an upper percentile of latency. The goal is to design a *control framework* that tries to avoid SLO violations, while keeping the cost of leased resources low.

Our solution is targeted for storage systems designed for horizontal scalability, such as key-value stores, that back interactive web applications. Examples of such systems are PNUTS [17], BigTable [14], Cassandra [3], SCADS [6], and HBase [4]. Requests in these systems have a simple communication pattern; each system at minimum provides `get` and `put` functionality on keys, and each request is single unit of work. We take advantage of this simplicity in our approach.

This simplified model also lends itself to easy partitioning of the key space across multiple servers, typically using a hash or range partitioning scheme. Each server node stores a subset of the data and serves requests for that subset. The control framework has two knobs: it can partition or replicate data to prevent servers from being overloaded when workload increases (e.g. due to diurnal variation or hotspots), or it can coalesce data and remove unnecessary replicas when the workload decreases. To make these configuration changes, the underlying storage system must be easy to reconfigure on-the-fly. Specifically, we require that it allows data to be copied from one server to another or deleted from a server, and that it provides methods like `AddServer` and `RemoveServer` to alter the number of leased servers. We previously designed and built SCADS [6] to both support this functionality and provide the simple communication pattern described above. As we further discuss in Section 7, running our own key-value store in the cloud has advantages over using a cloud-provided data service such as Amazon's S3.

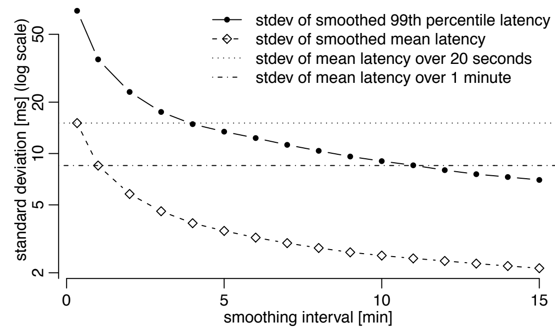


Figure 1: Standard deviation for the mean and 99th percentile of latency for increasing smoothing window sizes. The left-most points represent the raw measurements over 20-second periods. The average of the mean and 99th percentile latencies are 11 ms and 82 ms, respectively.

SCADS was designed to keep data memory-resident so that applications aren't required to use ad-hoc caching techniques to reach performance goals. This design provides similar performance benefits as Memcached; however, SCADS also supports real-time replication and load balancing. An example target application would be the highly interactive social networking site Facebook.com; most of their data remains memory-resident in order to hit performance targets [31].

In this section, we identify two challenges in scaling a storage system while maintaining a high-percentile SLO: noise and data movement. Benchmarks are presented to show the effects of each of these challenges.

2.2 Controlling a Noisy Signal

Figure 1 shows request latencies achieved by several key-value storage servers under a steady workload.[1] As expected, the standard deviation of both the mean latency and 99th percentile latency decreases as we increase the *smoothing window*, or time period over which the measurements are aggregated. However, as can be seen in the figure, the 99th percentile of latency would have be to smoothed over a four-minute window to achieve the same standard deviation as that achieved by the mean smoothed over a 20-second window (an 11x longer smoothing window). Similar effects are illustrated in experiments with Dynamo [20][2].

This observation has serious consequences if we are

[1]The workload consists of `get` and `put` requests against the SCADS [6] storage system, running on ten Amazon Elastic Compute Cloud (EC2) "Small" instances. Details of our experimental setup are in Section 6.1.

[2]See Figure 4 in [20]

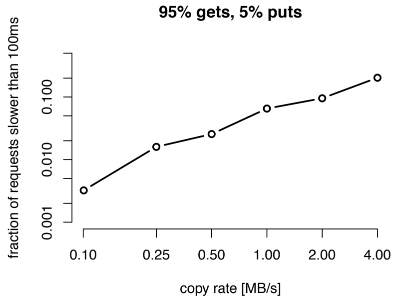


Figure 2: Impact on read performance during data copying on the write target. The x-axis represents the copy rate (in log scale) and the y-axis represents the fraction of requests slower than 100 ms (in log scale).

contemplating using classical closed-loop control. A long smoothing window means a longer delay before the control loop can make its next decision, resulting in more SLO violations. Furthermore, too much smoothing could mask a real spike in workload, and the controller would not respond at all. A short smoothing window mitigates both problems but can lead to oscillatory behavior [7]. Due to the high variance associated with a shorter smoothing window, the controller cannot tell if a server with high latency is actually overloaded or if it is simply exhibiting normally-occurring higher latency. A classical closed-loop controller might add servers in one iteration just to remove them in the next or may move data back and forth unnecessarily in response to such "false alarms." We show in Section 3 that a more effective approach is a *model-based* control in which the controller uses a different input signal than the quantity it is trying to control.

2.3 Data Movement Hurts Performance

Scaling a storage system requires data movement. Because each server is responsible for its own *state*, i.e., the data it stores, it is not generally true that any server can service any request. Simply adding and removing servers is not sufficient to respond to changes in workload, we additionally need to copy and move data between servers. However, data movement impacts performance and this impact is especially noticeable in the tail of the latency distribution. Impacting the tail of the distribution is of particular interest since we target upper percentile SLOs. As demonstrated in Figure 2, copying data increases the fraction of slow requests. In Dynamo [20], the data copy operations are run in low priority mode to minimize their impact on performance of interactive operations. Since one of our operational goals is to respond to spikes while minimizing SLO violations, our approach instead identifies and copies the smallest amount of data needed to relieve SLO pressure.

3 Design Techniques and Approach

Having outlined our goals and identified key challenges in Section 2, we now describe the design techniques in our solution. In particular, we use a model-predictive control, fine-grained workload statistics, and replication for performance predictability.

3.1 Model-Predictive Control

Model-predictive control (MPC) can yield improvements over classical closed-loop control systems in the presence of noisy signals because the controller takes as input a different signal than the one it is trying to control. In MPC, the controller uses a model of the system and its current state to compute the (near) optimal sequence of actions that maintain desired constraints. To simplify the computation of these actions, MPC considers a short receding time horizon. The controller executes only the first action in the sequence and then uses the new current state to compute a new sequence of actions. In each iteration, the controller reevaluates the system state and computes a new target state to adjust to changing conditions.

Realizing the improvements of MPC requires constructing an accurate model of the controlled system, which can be difficult in general. However, a distributed system with simple requests (see Section 2.1) is simpler to control: by avoiding per-server SLO violations, the controller avoids global violations.

We use a model of the system that predicts SLO violations based on the workload from individual servers. An overloaded server is in danger of a violation and needs to have data moved away. Similarly, the control framework uses the model to estimate how much spare capacity is left on an underloaded server, helpful for deciding which data should be moved there. Details of our model are in Section 5.4.

3.2 Reduce Data Movement

Figure 2 demonstrates that data movement negatively impacts performance. To reduce the amount of data copied between servers, we organize data as small units (*bins*), monitor workload to these bins, and move individual bins of data. This approach is commonly used to ease load-balancing [14, 17].

Monitoring workload statistics at a granularity finer than per-server is essential for the control framework to decide which data should be moved or copied. Without this information, it would be impossible to determine the minimal amount of data that could be moved from an overloaded server to bring it back to an SLO-compliant state. The performance model can predict how much "extra room" underloaded servers have, allowing the control framework to choose where to move the data. A "best-fit" policy that keeps the servers as fully utilized

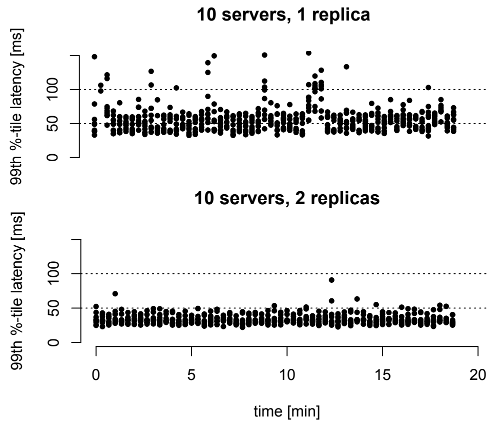


Figure 3: 99th percentile of latency over time measured during two experiments with steady workload. We kept the workload volume and number of servers the same, but changed the replication level from one data copy (top) to two (bottom). Horizontal lines representing the latencies 50 ms and 100 ms are provided for reference.

as possible is also important for scaling down leased resources, as unused servers can be released. Monitoring workload on small ranges of data give the control framework fine-grained information to move as little data as necessary to alleviate performance issues and to safely coalesce servers so they can be released.

3.3 Replication for Predictability

Distributed systems, particularly those operating in a cloud environment, typically experience environmental noise uncorrelated to a particular query or data [19]. In our benchmarks, we saw fluctuations in 99th percentile of latency over time and between different servers.

However, distributed systems also present the opportunity to use replication as a means of improving performance. In Dynamo, setting the read/write quorum parameters to be less than the total number of replicas achieves better request latency [20]. Another example is in the Google File System [21], which writes logs to different servers.

We handle performance perturbations caused by environmental noise by exploiting data replication; replication in the cloud environment is useful for performance predictability. Each request is sent to multiple replicas of the requested item and the first response is sent back to the client; this is the technique described in [20].

Figure 3 compares using one replica versus two on the same number of total servers (ten); shown is the 99th percentile of latency over time measured with steady workload. Note that the latency using replication is both smaller and more stable, even though each of these servers is doing more work than a server in the single replica scenario. It may seem that using single replicas with higher utilization would yield higher overall goodput (i.e., the amount of useful work accomplished per unit time). However, the extra work done by increasing the utilization will be in vain if those requests violate the SLO. In other words, the stringent SLO lowers the *useful* utilization of a single server.

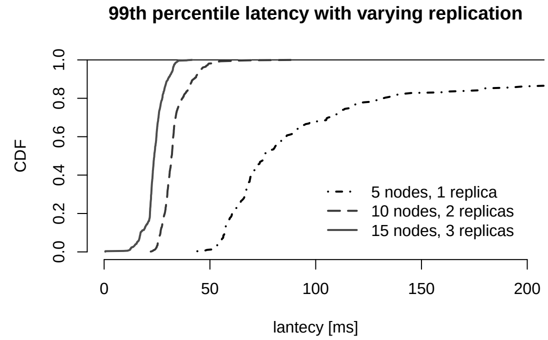


Figure 4: CDFs of 99th percentile latency measured every 20 seconds in three experiments. Each experiment yields the same goodput, however using more replicas results in lower and less variable latency.

Using more replicas yields lower variance in the 99th percentile. Figure 4 shows three Cumulative Distribution Functions (CDFs) of the 99th percentile of latency during three experiments using up to three replicas; each experiment yields the same goodput (workload to fully load five single replicas). Note the shorter tails on the distributions as the replication factor increases.

An advantage of using replication for performance is that it helps mask the effects of data movement during dynamic scaling. Thus replication is beneficial for alleviating both naturally-occurring and introduced noise.

Note that this data replication technique improves the 99th percentile latency from the perspective of the client, but does not reduce variance of the upper percentiles of latency of requests from an individual server. Therefore, the need for model-based control due to the difficulty in controlling a noisy signal remains present.

4 Related Work

Previous projects have addressed various subsets of our problem space, but to our knowledge none tackle the entire problem of the online control of the upper percentiles of latency in stateful, distributed systems.

Some work [2, 33] aims to optimize the static provisioning of a storage system before deploying to production. They search the configuration space for a cluster configuration that optimizes a specified utility function, but this optimization is done offline and performance is not considered during the re-configuration.

Other work tackles online configuration changes in storage systems, but only considers mean request latency rather than the upper percentile SLOs we consider. In

[16, 32], the authors propose a database replication policy for automatic scale up and down. In [32], they use a reactive, feed-back controller which monitors request latency and adds additional full replicas of the database. An enhancement in [16] uses a performance model to add replicas via a proactive controller. These papers additionally differ from our work in their assumption that the full dataset fits on a single server, thus they only consider adding a full replica when scaling up (instead of also partitioning).

In [25], the controller adds and removes nodes from a distributed file system, rebalancing data as servers come and go. However this work focuses more on controlling the rebalance speed rather than choosing which data to move to which servers; the work additionally does not focus on upper-percentile SLOs.

Some systems target large-scale storage servers with terabytes of data on each machine and thus cannot handle a sustained workload spike or data hotspot because the data layout cannot change on-the-fly. For example: in Everest [28], the authors propose a *write off-loading technique* that allows them to absorb short burst of writes to a large-scale storage system. Performance improvement is measured as 99th percentile of latency during the 30 minute experiments, however they do not attempt to maintain a stringent SLO over short time intervals. Sierra [35] and Rabbit [1] are power-proportional systems that alter power consumption based on workload. The approach that both papers take is to first provision the system for the peak load with multiple replicas of all data and then turn off servers when the workload decreases. Both papers evaluate the performance of the system under the power-proportional controller (Sierra uses the 99th percentile of latency), but these systems could not respond to workload spikes taller than the provisioned capacity or to unexpected hotspots that affect individual servers. SMART [38] is evaluated on a large file system that prevents it from quickly responding to unexpected spikes and does not consider upper percentiles of latency.

Most DHTs [8] are designed to withstand churn in the server population without affecting the availability and durability of the data. However, quickly adapting to changes in user workload and maintaining a stringent performance SLO during such changes are not design goals. Amazon's Dynamo [20] is an example of a DHT that provides an SLO on the 99.9th percentile of latency, but the authors mention that during a busy holiday season it took almost a day to copy data to a new server due to running the copy action slow enough to avoid performance issues; this low-priority copying would be slow to respond to unexpected spikes.

Much has been published on dynamic resource allocation for stateless systems such as Web servers or application servers [15, 36, 26, 23, 22, 34], even considering stringent performance SLOs. However, most of that work does not directly apply to stateful storage systems: the control polices for stateless systems need only vary the number of active servers because any server can handle any request. These policies do not have to consider the complexities of data movement.

Aqueduct [27] is a migration engine that moves data in a storage system while guaranteeing a performance SLO on mean request latency. It does not directly respond to workload, but could be used instead of the action scheduler in our control framework (see Section 5.6).

5 The Control Framework

This section describes the design and implementation of the control framework, incorporating the strategies outlined in Section 3. The framework uses per-server workload and the performance model to determine when a server is overloaded and thus when to copy data. It chooses *what* to copy based on workload statistics on small units of data (*bins*). Finer statistics together with the models inform *where* to copy data.

5.1 The control loop

The control framework consists of a controller, workload forecaster, and action scheduler which, together with the storage system and performance models, form a control loop (see Figure 5). These components are described in more detail in subsequent sections.

We focus on the controller, which is responsible for altering the configuration of the cluster by prescribing actions that add/remove servers and move/copy data between servers. Its decisions are based on a view of the current state given by the workload forecaster and the current data layout, in consultation with models that predict how servers will perform under particular loads. After the controller compiles a list of actions to run on the cluster, the action scheduler executes them.

Workload statistics are maintained for small ranges of

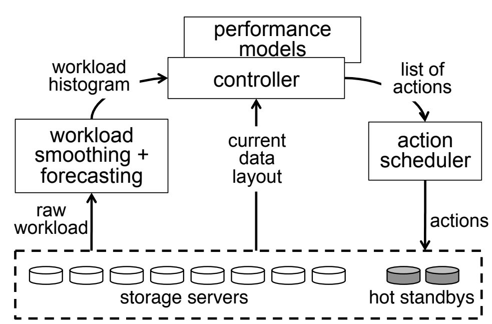


Figure 5: The control framework modules—workload forecasting, controller, performance model, and action execution—form a control loop that interacts with the storage system.

data called *bins*; each bin is about 10-100 MB of data. These bins also represent the unit of data movement. We assume a bin cannot be further partitioned and will need to be replicated if its workload exceeds the capacity of a single server. The total number of bins is a parameter of the control framework. Setting the value too low or too high has its drawbacks. With too few data bins, the controller does not have enough flexibility in terms of moving data from overloaded servers and might have to copy more data than necessary. Having too many data bins increases the load on the monitoring system and running the controller might take longer since it would have to consider more options. In practice, having on average five to ten bins per server is a good compromise.

5.2 A manipulable storage system

The SCADS [6] storage system provides an interface for dynamic scaling: it is easy to control which servers have which data, and data can be manipulated as small bins. SCADS is an eventually consistent key-value store with range partitioning. Each node can serve multiple small ranges; e.g., keys A-C, G-I. We use the `get` and `put` operators; read requests are satisfied from one or more servers, and writes are asynchronously propagated and flushed to all replicas.

SCADS provides an interface for copying and moving data between pairs of servers; replication is accomplished by copying the target data range to another server, and partitioning is the result of moving data from one server to another. The SCADS design makes low latency a top priority, thus all data is kept in memory. This characteristic has little impact on the control framework, besides simplifying the performance modeling described in Section 5.4.

5.3 Controller

Given the workload statistics in each bin, the minimal number of servers would be achieved by solving a bin-packing problem—packing the data bins into servers—an NP-complete problem. While approximate algorithms exist [37], they typically do not consider the current locations of the bins and thus could completely reshuffle the data on the servers, a costly operation. Instead, our controller uses a greedy heuristic that moves data from the overloaded servers and coalesces underloaded servers. While there are many possible controller implementations, we describe our design that leverages the solutions outlined above.

The controller executes periodically to decide how to alter the configuration of the cluster; the frequency is an implementation parameter. In each iteration, the controller prescribes actions for overloaded and underloaded servers as well as changing the number of servers. By the end of an iteration, the controller has compiled a list of actions to be run on the cluster, which are then executed by the action scheduler (see Section 5.6).

Algorithm 1 Controller iteration

```
1:  estimate workload on each server
2:  identify servers that are overloaded or underloaded
3:
4:  for all overloaded server S do
5:    while S is overloaded do
6:      determine hottest bin H on S
7:      if workload on H is too high for a single server then
8:        move and replicate H to empty servers
9:      else
10:       move H to the most-loaded underloaded server
          that can accept H without SLO violation
11:
12: for all underloaded server S do
13:   if S contains only a single bin replica then
14:     remove the bin if no longer necessary
15:   else
16:     for all bin B on S do
17:       move B to most-loaded underloaded server that
          can accept B
18:       if cannot move B then
19:         leave it on S
20:
21: add/remove servers as necessary, as per previous actions
```

Pseudocode for the controller is shown in Algorithm 1. Using a performance model (described in the next subsection), the controller predicts which servers are underloaded or overloaded. Lines 4-10 describe the steps for fixing an overloaded server: moving bins that have too much workload for one server to dedicated servers, or moving bins to the most loaded servers that have enough capacity, a “best-fit” approach. Next, in lines 12-19, in an attempt to deallocate servers for scaling down, the controller moves bins from the least loaded loaded servers to other underloaded servers. Finally, servers are added and removed from the cluster. To simplify its reasoning about the current state of the system, the controller waits until previously scheduled copy actions complete. Long-running actions could block the controller from executing, preventing it from responding to sudden changes in workload. An action that needs to move many bins from one server to another. To avoid scheduling such actions, the controller uses a copy-duration model to estimate action duration and splits potentially long-running actions into shorter ones. For example, an action that needs to move many bins from one server to another can be split into several actions that move fewer bins between the two servers. If some of the actions do not complete within a time threshold, the controller can cancel them to reassess the current state and continue to respond to workload changes.

The controller can also maintain a user-specified number of standby servers, a form of extra capacity in addition to overprovisioning in the workload smoothing component (see Section 5.5). These standbys help the controller avoid waiting for new servers to boot up during a sudden workload spike, as they are already running the storage system software but not serving any data. Standbys are particularly useful for handling hotspots when replicas of a bin require an empty server.

The presence of a centralized component such as the controller does not necessarily mean the system isn't scalable[19]. Nevertheless, there is likely a limit to the number of decisions the controller can make per unit time for a given number of servers and/or bins. In our results, the controller inspects forty servers in a few seconds; experimenting with a larger cluster is future work. If a decision-making limit is approached, the controller may need to make decisions less frequently; this could impact the attainable SLO if the workload changes rapidly. However, with more servers, the controller has more flexibility in placing data, meaning it doesn't have to consider many servers when relocating a particular bin.

5.4 Benchmarking and modeling

The controller uses models of system performance to determine which servers are overloaded/underloaded and to guide its decisions as to which data to move where, as well as how many servers to add or remove. Recall that Model-Predictive Control requires an accurate model of the system. Instead of responding to changes in 99th percentile of request latency, our controller responds directly to changes in system workload. Therefore, the controller needs a model that accurately predicts whether a server can handle a particular workload without violating the performance SLO. Our controller also uses a model of duration of the data copy operations to create short copy actions.

One of the standard approaches to performance modeling is using analytical models based on network of queues. These models require detailed understanding of the system and often make strong assumptions about the request arrival and service time distributions. Consequently, analytical models are difficult to construct and their predictions might not match the performance of the system in production environments.

Instead, we use statistical machine learning (SML) models. As noted in the solutions above, a model-based approach allows us to use a signal other than latency in the control loop. Consequently, the controller needs an accurate model of the system on which to base its decisions. Building a model typically involves gathering training data by introducing a range of inputs into the system and observing the outcomes. In a large-scale system it becomes more difficult to construct the appropriate set of inputs [7]. Furthermore, it is more likely in a larger system to only be able to observe a subset of the component interactions that actually take place. Not having knowledge of all interactions (*unmodeled dynamics*) leads to a less accurate model.

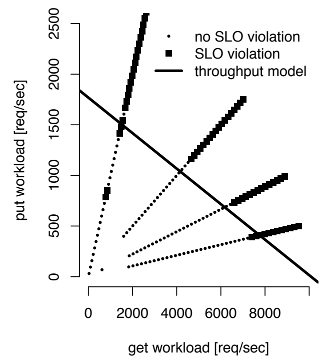


Figure 6: The training data and steady-state model for two replicas. The x- and y-axes represent the request rates of get and put operations, and the small dots and large squares represent workloads that the server can and cannot handle, respectively. The solid line crossing the four others is the boundary of the performance model. SCADS can handle workload rates to the left of this line.

Fortunately, we can leverage the simple communication pattern of SCADS requests to simplify the modeling process. Other key-value stores with similar simple requests would also be amenable to modeling. Below we describe the development and use of two models, the *steady-state* model and the *copy-duration* model. All benchmarks were run on tens of SCADS servers and workload-generating clients on Amazon's Elastic Compute Cloud (EC2) on m1.small instances.

Simple changes in workload, such as a shift in popularity of individual objects [11, 5, 9], will not affect the accuracy of these offline models as all SCADS requests are served from memory. The performance of these offline models (and thus the system) may degrade over time if new, unmodeled features are added to the application. For example, an individual request may become more expensive if it returns more data or if new types of requests are supported. The model's degradation speed would be application-specific, however these feature-change events are known to the developer and the offline models can be periodically rebuilt via benchmarking and fine-tuned in production [12].

Steady-state model: The steady-state performance model is used to predict whether a server can handle a particular workload without violating a given latency threshold. The controller uses this model to detect which servers are overloaded and to decide where data should be moved. To build this model, we benchmark SCADS under steady workload for a variety of workload mixes: read/write ratios 50/50, 80/20, 90/10 and 95/5 (these mixes are also used in [18]). We then create a linear classification model using logistic regression, based on training data from the benchmarks. The model has two covariates (features): the workload rate of `get` and `put` requests. For each workload mix, we determine the workload volume at which the latency threshold specified by the SLO would be surpassed. This workload volume separates two classes: SLO violation or no violation. Thus, for a particular workload, the model can predict whether a server with that workload would violate the SLO. Figure 6 illustrates the steady-state linear model and the training data used to generate it.

Copy-duration model: To allow the controller to estimate how long it will take to copy data between two servers, we build a model that predicts the rate of data transfer during a `copy` action. While the `copy` operation in SCADS has a parameter for specifying the number of bytes/second at which to transfer data, the actual rate is often lower because of activity on both servers involved. Our model thus predicts the *copy-rate factor*—the ratio of observed to specified copy-rate. A factor of 0.8 means that the actual `copy` operation is only 80% the specified rate. We use this estimate of the actual rate to compute the duration of the `copy` action.

To build the model, we benchmark duration of `copy` actions between pairs of servers operating at various workload rates. We then model the copy rate factor using linear regression; covariates are linear and quadratic in the specified rate and `get` and `put` request rates.

While our controller does not directly consider the effects of data copy on system performance during real-time decisions, we considered these effects when designing the controller and the action execution modules. Recall that Figure 2 summarizes the results of benchmarking SCADS during copy operations; performance is affected mostly on the target servers for the copy action. Also note that in both performance models network utilization and activity of other VMs are ignored. These effects are part of environmental noise described earlier, and are compensated for with replication.

5.5 Workload Monitoring and Smoothing

In addition to performance models, the controller needs to know how workload is distributed amongst the data. Workload is represented by a histogram that contains request rates for individual request types (`get` and `put`) for each bin. To minimize the impact of monitoring on performance, we sample 2% of `get` requests for use in our statistics (`put` requests are sampled at 40% because there are fewer `put` requests in our workload mixes). We found that using higher sampling rates did not greatly improve accuracy.

Every twenty seconds, a summary of the workload volume is generated for each bin. This creates the raw workload histogram: for each bin we have counts of the number of `get` and `put` requests to keys in that bin. To prevent the controller from reacting to small variance in workload, the raw workload is smoothed via hysteresis. As scaling up is more important than scaling down with respect to performance, we want to respond quickly to workload spikes while coalescing servers more slowly. We apply smoothing with two parameters: α_{up} and α_{down}. If the workload in a bin increases relative to the last time step's smoothed workload, we smooth that bin's workload with α_{up}; otherwise we use the α_{down} smoothing parameter. For example, in the case of increasing workload at time t we have: $smoothed_t = smoothed_{t-1} + \alpha_{up} * (raw_t - smoothed_{t-1})$.

The smoothed workload can also be amplified using an *overprovisioning* factor. Overprovisioning causes the controller to think the workload on a server is higher than it actually is. For instance, an overprovisioning factor of 0.1 would make an actual workload of w appear to the controller as $1.1w$. Thus overprovisioning creates a "safety buffer" that buys the controller more time to move data. For more discussion of tradeoffs, see Section 7.

The controller bases its decisions on an estimate of the workload at each server, determined by sampling the requests. Calculating per-bin workload in a centralized controller may prove unscalable as the number of requests to sample grows large. While we used a single server to process the requests and compute the per-bin workloads, the Chukwa monitoring system [29] could be distributed over a cluster of servers. The monitoring system could then prioritize the delivery of the monitoring data to the controller, sending updates only for bins with significant changes in workload. Another approach would have each server maintain workload information over a specified time interval. The controller could then query for the workload information when it begins its decision-making process.

5.6 Action Scheduler

On most storage systems, copying data between servers has a negative impact on performance of the interactive workload. In SCADS, the copy operation significantly affects the target server (see Figure 2), while the source server is mostly unaffected. Therefore, executing all data copy actions concurrently might overwhelm the system and reduce performance. Executing the actions sequentially would minimize the performance impact, but would be very slow.

In addition to improving steady-state performance of storage systems, replication helps smooth performance during data copy. We specify a constraint that each bin have at least one replica on a server that is not affected by data copy. The action scheduler iterates through its list of actions and schedules concurrently all actions that do not violate the constraint. When an action completes, the scheduler repeats this process with the remaining unscheduled actions.

5.7 Controller Parameters

A summary of the parameters used by the controller appear in Table 5.7, along with the values used in our experiments (in Section 6). The hysteresis parameters α_{up} and α_{down} affect how abruptly the controller will scale up and down. Reasonable values for these parameters can be chosen via simulation [13].

Controller Parameter	Value
execution period	20 seconds
α_{up}, α_{down}	0.9, 0.1
number standbys	2
overprovisioning	0.1 or 0.3
copyrate	4 MB/s

6 Experimental Results

We evaluate our control framework implementation by stress testing it with two workload profiles that represent the main scenarios where our proposed control framework could be applied. The first workload contains a spike on a single data item; as shown in [11], web applications typically experience hotspots on a small fraction of the data. Unexpected workload spikes with data hotspots are difficult to handle in stateful systems because the location of the hotspot is unknown before the spike. Therefore, statically overprovisioning for such spikes would be expensive. Managing and monitoring small data ranges is especially important for dealing with these hotspots, particularly when quick replication is needed. The second workload exhibits a diurnal workload pattern: workload volume increases during the day and decreases at night; this profile demonstrates the effectiveness of both scale-up and scale-down.

For the hotspot workload, we observe how well the control framework is able to react to a sudden increase in workload volume, as well as how quickly performance stabilizes. We also look at the performance impact during this transition period. Note, however, that any system will likely have some visible impact for sufficiently strict characteristics of the spike (i.e., how rapidly it arrives and how much extra workload there is). The diurnal workload additionally exercises the control framework's ability to both scale up and down. Finally, we discuss some of the tradeoffs of SLO parameters and cost of leased resources, as well as potential savings to be gained by scaling up and down.

6.1 Experiment setup

Experiments were run using Amazon's Elastic Compute Cloud (EC2). We ran SCADS servers on m1.small instances using 800 MB of the available RAM as each storage server's in-memory cache. We gained an understanding of the variance present in this environment by benchmarking SCADS' performance both in the absence and presence of data movement, see Section 5.4. As described in Section 2, latency variance occurs in the upper quantiles even in the absence of data movement. Therefore we maintain at least two copies of each data item, using the replication strategy described earlier: each `get` request is sent to both replicas and we count the faster response as its latency. We do not consider the latency of `put` requests, as the work described in this paper is targeted towards OLTP-type applications similar to those described by [18], in which read requests dominate writes. Furthermore, evaluating latency for write requests isn't applicable in an eventually consistent system, such as SCADS. More appropriate would be an SLO on data staleness, a subject for future work.

Workload is generated by a separate set of machines, also m1.small instances on EC2. These experiments use sixty workload-generating instances and twenty server instances. The control framework runs on one m1.xlarge instance. The controller uses a 100 ms SLO threshold on latency for `get` requests, and in the description of each experiment we discuss the other two parameters of the SLO: the percentile at which to evaluate the threshold, and the interval over which to assess violations. Table 1 summarizes the parameter values used in the two experiments. To avoid running an experiment for an entire day, we execute it in a shorter time. We control the length of the boot-up time in the experiment by leasing all the virtual machines needed before the experiment begins and simply adding a delay before a "new" server can be used. This technique allows us to replay the Ebates.com work-

Parameter	Hotspot	Diurnal
server boot-up time	3 minutes	15 seconds
server charge interval	60 minutes	5 minutes
server capacity	800 MB	66.7 MB
size of 1 key-value	256 B	256 B
total number of keys	4.8 million	400,000
minimum # of replicas	2	2
total data size	2.2 GB	196 MB
read/write ratio	95/5	95/5

Table 1: Various experiment parameters for the hotspot and diurnal workload experiments. We replay the diurnal workload with a speed-up factor of 12 and thus also reduce the server boot-up and charge intervals and the data size by a factor of 12.

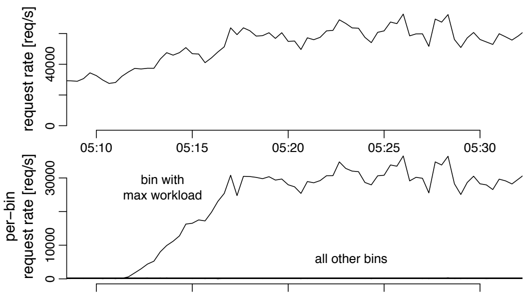


Figure 7: Workload over time in the Hotspot experiment. Top row: aggregate request rate during the spike doubled between 5:12 and 5:17. Bottom row: request rate for each of the 200 data bins; the rate for the hot bin increased to approximately 30,000 reqs/sec.

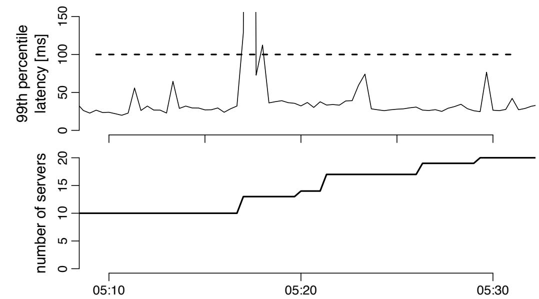


Figure 8: Performance and resources in the Hotspot experiment. Top row: 99$^{\text{th}}$ percentile of latency along with the 100 ms threshold (dashed line). Bottom row: number of servers over time. The controller keeps up with the spike for the first few minutes, then latency increases above the threshold, but the system quickly recovers.

Interval	Max percentile
5 minutes	98
1 minute	95
20 seconds	80

Table 2: The maximum percentile without SLO violations for each interval in the Hotspot experiment. Notice that we can support higher latency percentiles for longer time intervals.

load trace [10] 12x faster: replaying twenty-four hours of the trace in two hours. To retain the proportionality of the other time-related parameters, we scale down by 12x the data size, server cost interval, boot up time, and server release time. The data size is scaled down because we can't speed up the copy rate higher than the network bandwidth on m1.small instances allows. Additionally, the total data size is limited by the maximum storage on the number of servers when the cluster is scaled down. As SCADS keeps its data in memory, server capacity is limited by available memory on the m1.small instance.

6.2 Hotspot

We create a synthetic spike workload based on the statistics of a spike experienced by CNN.com after the September 11 attacks [24]. The workload increased by an order of magnitude in 15 minutes, which corresponds to about 100% increase in 5 minutes. We simulate this workload by using a flat, one-hour long period of the Ebates.com trace [10] to which we add a workload spike with a single hotspot. During a five minute period, the aggregate workload volume increases linearly by a factor of two, but all the additional workload is directed at a single key in the system. Figure 7 depicts the aggregate workload and the per-bin workload over time. Notice that when the spike occurs, the workload in the hot bin greatly exceeds that in all other bins.

Our controller dynamically creates eight additional replicas of this hot data bin to handle the spike. Figure 8 shows the performance (99th percentile latency) and the number of servers over time. The workload spike impacts performance for a brief period. However, the controller quickly begins replicating the hot data bin. It first uses the two standbys, then requests additional servers. Performance stabilizes in less than three minutes.

It is relatively easy for our control framework to react to spikes like this because only a very small fraction of the data has to be replicated. We can thus handle a spike with data hotspots with resources proportional to the magnitude of the spike, not proportional to the size of the full dataset or the number of servers.

The performance impact when the spike first arrives is brief, but may result in an SLO violation, depending how the SLO is specified. The SLO is parameterized by the latency threshold, latency percentile, and duration of the SLO interval. Fixing the latency threshold at 100 ms, in Table 6.2 we show how varying the interval affects the maximum percentile under which no violations occurred.

In general, SLOs specified over a longer time interval are easier to maintain despite drastic workload changes; this experiment has one five-minute violation. Similarly, an SLO with a lower percentile will have fewer violations than a higher one. In this experiment, there are zero violations over a twenty-second window when looking at the 80$^{\text{th}}$ percentile of latency, but extending the interval to five minutes can yield the 98th percentile.

The cost tradeoff between SLO violations and leased resources depends in part on the cost of a violation. Whether a violation costs more than leasing enough servers to overprovision the system to satisfy a hotspot on *any* data item will be application-specific. Dynamic scaling, however, has the advantage of not having to estimate the magnitude of unexpected spikes.

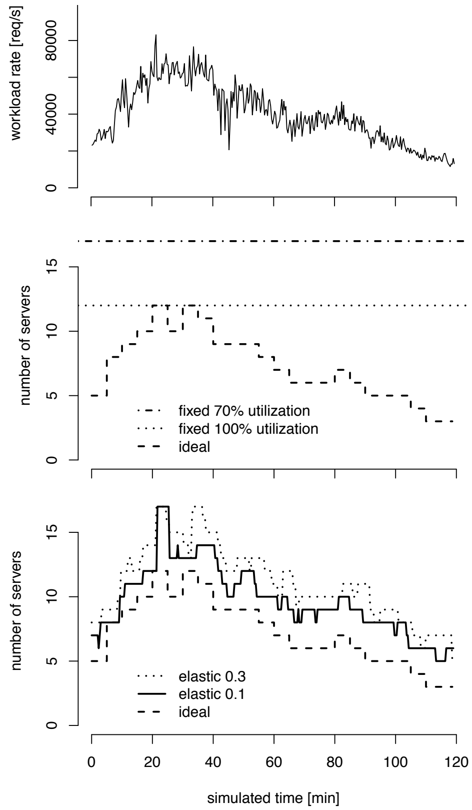


Figure 9: Top: Diurnal workload pattern. Middle: number of servers assuming the ideal server allocation and two fixed allocations during the diurnal workload experiment. Bottom: ideal server allocation and two elastic allocations using our control framework.

6.3 Ebates.com diurnal workload trace

The diurnal workload profile is derived from a trace from Ebates.com [10]; we use the trace's aggregate workload pattern; data accesses follow a constant zipfian distribution. This profile shows the control framework's effectiveness in scaling both up and down as the workload volume on all data items fluctuates. We replay twenty-four hours of the trace in two hours, a 12x speedup.

We experiment using two overprovisioning parameters (see Section 5.5 on workload smoothing). With 0.3 overprovisioning, the smoothed workload is multiplied by a factor of 1.3. With more headroom, the system can better absorb small spikes in the workload. Using 0.1 overprovisioning has less headroom, thus higher savings at the cost of worse performance.

We compare the results of our experiments with the ideal resource allocation and two fixed allocation calculations. In the ideal allocation, we assume that we know the workload at each time step throughout the experiment and compute the minimum number of servers we would need to support this workload for each 5-minute interval (the scaled-down server cost interval). The ideal allocation assumes that moving data is instantaneous and has no effect on performance, and provides the lower bound on the number of compute resources required to handle this workload without SLO violations.

The fixed-100% and fixed-70% allocations use a constant number of servers throughout the experiment. Fixed-100% assumes the workload's peak value is known a priori, and computes the number of servers based on that value and the maximum throughput of each server (7000 requests per second, see Section 5.4). The number of servers used in the fixed-100% allocation equals the maximum number of servers used by the ideal allocation. Fixed-70% is calculated similarly to the fixed-100%, but restricts the servers' utilization to 70% of their potential throughput (i.e., $7,000 * 0.7 = 4,900$ requests per second). Fixed-100% is the ideal fixed allocation, but in practice datacenter operators often add more headroom to absorb unexpected spikes.

Figure 9 shows the workload profile and the number of server units used by the different allocation policies: ideal, fixed-100%, fixed-70%, and our elastic policy with overprovisioning of 0.3 and 0.1. A server unit corresponds to one server being used for one charge interval, thus fewer server units used translates to monetary cost savings. The policy with 0.1 overprovisioning achieves savings of 16% and 41% compared to the fixed-100% and fixed-70% allocations, respectively.

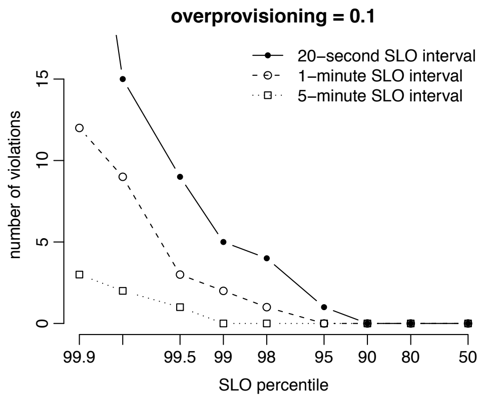
The ideal resource allocation uses 175 servers units, while using overprovisioning of 0.1 uses 241 server units. However, recall that our controller maintains two empty standby servers to quickly respond to data hotspots that require replication. The actual number of server units used for serving data is thus 191 which is within 10% of the ideal allocation[3].

Performance and SLO violations are summarized in Figure 10. Note that it is more difficult to maintain SLOs with shorter time intervals and higher percentiles.

7 Discussion

The experiments demonstrate the control framework's effectiveness in scaling both up and down for typical workload profiles that exhibit fluctuating workload patterns. Having the same mechanism work well in scenarios with rapidly appearing hotspots as well as more gradual variations is advantageous because application developers won't need to decide a priori what type of growth to prepare for: the same control framework can dynamically scale as needed in either case. For operators who still prefer to maintain a fixed allocation for non-spike, peak traffic, say on their own hardware, there is still potential to utilize the control framework for surge computing in the cloud. A temporary spike could be sat-

[3]The experiment has a total of 25 5-minute server-charging intervals which yields 50 server units used by the standbys.



Interval	Max percentile	
	0.3 overprovision	**0.1 overprovision**
5 minutes	99.5	99
1 minute	99	95
20 seconds	95	90

Figure 10: Top: Number of SLO violations during the 0.1 overprovisioning diurnal experiment, for different values of the SLO percentile. The three lines in the graph correspond to the three intervals over which to evaluate the SLO: 5 minutes, 1 minute, and 20 seconds. Bottom: summary of SLO violations and maximum latency percentile supported with no SLO violations during the diurnal workload with two different overprovisioning parameters.

isfied with leased resources from the cloud, which would be relinquished once the spike subsides.

There are cost implications in setting some of the control framework's parameters to manage "extra capacity," namely the number of standbys and the overprovisioning factor. Both these techniques result in higher server costs, either due to maintaining booted empty servers for standbys or underutilization of active servers in the case of overprovisioning. Standby servers are particularly helpful for dealing with workload spikes which necessitate replication, as empty servers are waiting and ready to receive data. Overprovisioning is better for workload profiles like a diurnal pattern in which all data items more slowly experience increased access rates; this headroom allows the control framework more time to shuffle data around without overloading the servers. Reducing the number of standbys and/or the overprovisioning factor can yield cost savings, with the associated risk of SLO violations if scaling up is not performed rapidly enough.

We presented results of our controller using replication to both smooth variance and lessen the effects of data movement. To see that the controller remains robust to the variance in the environment without replication, we performed the same two experiments using only a single copy of each data item. While SCADS still scales effectively, the variance limits the attainable SLO percentile. For example, in the hotspot workload, the 5-minute, 1-minute, and 20-second attainable percentiles were 95, 80, and 80, respectively, compared to 98, 95, and 80 when using replication. The replication factor thus offers a tradeoff between performance/robustness and the cost of running the system. Note, however, that a different environment than EC2, like dedicated hardware, may have less variance and thus may achieve the desired SLO without replication.

The ability to control these performance tradeoffs is an advantage of running the SCADS key-value store on EC2 rather than simply using S3 for data storage. In general, S3 is optimized for larger files and has nontrivial overhead per HTTP request. S3 also does not offer a SLO on latency, while SCADS offers a developer-specified SLO. Data replication factor and data location are not tunable with S3, which would make maintaining a particular SLO difficult. More fundamentally, S3 does not provide the API that SCADS on EC2 does. SCADS supports features like `TestAndSet()` and various methods on ranges of keys; this enables a higher level query language on top. Additionally, the SCADS client library supports read/write quorums for trading off performance and consistency, this would also be meaningless without being able to control the replication factor.

8 Future Work

Future work includes incorporating resource heterogeneity in the control framework, as well as designing a framework simulator for performing what-if analysis. Cloud providers typically offer a variety of resources at different cost, e.g., paying more per hour for a server with more CPU or disk capacity. By modeling performance of different server types, we could include in the control framework decisions about which type of server to use. Additionally, we hope to use the performance models in a control framework simulator that emulates the behavior of real servers. The simulator could be used for assessing the performance-cost tradeoff for unseen workloads; developers could create synthetic workloads using the features described in [11].

9 Conclusion

The elasticity of the cloud provides an opportunity for dynamic resource allocation, scaling up when workload increases and scaling down to save money. To date, this opportunity has been exploited primarily by stateless services, in which simply adding and removing servers is sufficient to track workload variation. Our goal was to design a control framework that could automatically scale a *stateful* key-value store in the cloud while complying with a stringent performance SLO in which a very high percentile of requests (typically 99%) must meet a specific latency bound. As described in Section 2, meeting such a stringent SLO is challenging both because of high variance in the tail of the request latency distribution and because of the need to copy data in addition to adding and removing servers. Our solution avoids trying to control for such a noisy latency signal, instead using a model-based approach that maps workload to latency. This model, combined with fine-grained workload statistics, allows the framework to move only necessary data to alleviate performance issues while keeping the amount of leased resources needed to satisfy the current workload. In the event of an unexpected hotspot, replicas are added proportional to the magnitude of the spike, not the total number of servers. For workload that exhibits a diurnal pattern, the framework easily scales both up and down as the workload fluctuates. In the midst of this dynamic scaling, we use replication to mask both inherent environmental noise and the performance perturbations introduced by data movement. We anticipate that this work provides a useful starting point for allowing large-scale storage systems to take advantage of the elasticity of cloud computing.

10 Acknowledgements

We would like to thank our fellow students and faculty in the RAD lab for their ongoing thoughtful advice throughout this project. We additionally thank Kim Keeton, Tim Kraska, Ari Rabkin, Eno Thereska, and John Wilkes for their feedback on this paper.

This research is supported in part by a National Science Foundation graduate fellowship, and gifts from Sun Microsystems, Google, Microsoft, Amazon Web Services, Cisco Systems, Cloudera, eBay, Facebook, Fujitsu, Hewlett-Packard, Intel, Network Appliance, SAP, VMWare and Yahoo! and by matching funds from the State of California's MICRO program (grants 06-152, 07-010, 06-148, 07-012, 06-146, 07-009, 06-147, 07-013, 06-149, 06-150, and 07-008), the National Science Foundation (grant #CNS-0509559), and the University of California Industry/University Cooperative Research Program (UC Discovery) grant COM07-10240.

References

[1] H. Amur, J. Cipar, V. Gupta, G. R. Ganger, M. A. Kozuch, and K. Schwan. Robust and Flexible Power-Proportional Storage. In *SoCC: ACM Symposium on Cloud Computing*, 2010.

[2] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running Circles Around Storage Administration. In *FAST: Conference on File and Storage Technologies*, 2002.

[3] Apache. Cassandra. incubator.apache.org/cassandra, 2010.

[4] Apache. HBase. hadoop.apache.org/hbase, 2010.

[5] M. Arlitt and T. Jin. Workload Characterization of the 1998 World Cup Web Site. Technical Report HPL-1999-35R1, HP Labs, 1999.

[6] M. Armbrust, A. Fox, D. Patterson, N. Lanham, H. Oh, B. Trushkowsky, and J. Trutna. SCADS: Scale-independent Storage for Social Computing Applications. In *Conference on Innovative Data Systems Research (CIDR)*, 2009.

[7] K. J. Åström. *Introduction to Stochastic Control Theory*. Academic Press, 1970.

[8] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking Up Data in P2P Systems. *Communications of the ACM*, 46(2), February 2003.

[9] P. Barford and M. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proceedings of the ACM SIGMETRICS Joint International Conference*, 1998.

[10] P. Bodík et al. Combining Visualization and Statistical Analysis to Improve Operator Confidence and Efficiency for Failure Detection and Localization. In *International Conference on Autonomic Computing (ICAC)*, 2005.

[11] P. Bodík, A. Fox, M. J. Franklin, M. I. Jordan, and D. Patterson. Characterizing, Modeling, and Generating Workload Spikes for Stateful Services. In *SoCC: ACM Symposium on Cloud Computing*, 2010.

[12] P. Bodík, R. Griffith, C. Sutton, A. Fox, M. I. Jordan, and D. A. Patterson. Automatic Exploration of Datacenter Performance Regimes. In *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds*, 2009.

[13] P. Bodík, R. Griffith, C. Sutton, A. Fox, M. I. Jordan, and D. A. Patterson. Statistical Machine Learning Makes Automatic Control Practical for Internet Datacenters. In *Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[14] F. Chang et al. Bigtable: A Distributed Storage System for Structured Data. In *OSDI: Symposium on Operating Systems Design and Implementation*, 2006.

[15] J. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing Energy and Server Resources in Hosting Centers. In *Symposium on Operating Systems Principles (SOSP)*, 2001.

[16] J. Chen, G. Soundararajan, and C. Amza. Autonomic Provisioning of Backend Databases in Dynamic Content Web Servers. In *International Conference on Autonomic Computing (ICAC)*, 2006.

[17] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2008.

[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC: ACM Symposium on Cloud Computing*, 2010.

[19] J. Dean. Evolution and Future Directions of Large-scale Storage and Computation Systems at Google. In *SoCC: ACM Symposium on Cloud Computing*, 2010.

[20] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Symposium on Operating Systems Principles (SOSP)*, 2007.

[21] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Symposium on Operating Systems Principles (SOSP)*, 2003.

[22] J. L. Hellerstein, V. Morrison, and E. Eilebrecht. Optimizing Concurrency Levels in the .NET ThreadPool: A Case Study of Controller Design and Implementation. In *Workshop on Feedback Control Implementation and Design in Computing Systems and Networks*, 2008.

[23] D. Kusic et al. Power and Performance Management of Virtualized Computing Environments Via Lookahead Control. In *International Conference on Autonomic Computing (ICAC)*, 2008.

[24] W. LeFebvre. CNN.com: Facing a world crisis. www.tcsa.org/lisa2001/cnn.txt, 2001.

[25] H. C. Lim, S. Babu, and J. S. Chase. Automated Control for Elastic Storage. In *International Conference on Autonomic Computing (ICAC)*, 2010.

[26] X. Liu, J. Heo, L. Sha, and X. Zhu. Adaptive Control of Multi-Tiered Web Applications Using Queueing Predictor. *Network Operations and Management Symposium*, April 2006.

[27] C. Lu, G. A. Alvarez, and J. Wilkes. Aqueduct: Online Data Migration with Performance Guarantees. In *FAST: Conference on File and Storage Technologies*, 2002.

[28] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. I. T. Rowstron. Everest: Scaling Down Peak Loads Through I/O Off-Loading. In *OSDI: Symposium on Operating Systems Design and Implementation*, 2008.

[29] A. Rabkin and R. H. Katz. Chukwa: A System for Reliable Large-scale Log Collection. Master's thesis, EECS Department, University of California, Berkeley, Mar 2010.

[30] J. A. Rossiter. *Model Based Predictive Control: A Practical Approach*. CRC Press, 2003.

[31] J. Rothschild. High Performance at Massive Scale - Lessons Learned at Facebook. http://cns.ucsd.edu/lecturearchive09.shtml#Roth, October 2009.

[32] G. Soundararajan, C. Amza, and A. Goel. Database Replication Policies for Dynamic Content Applications. In *EuroSys: ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2006.

[33] J. D. Strunk, E. Thereska, C. Faloutsos, and G. R. Ganger. Using Utility to Provision Storage Systems. In *FAST: Conference on File and Storage Technologies*, 2008.

[34] G. Tesauro, N. Jong, R. Das, and M. Bennani. A Hybrid Reinforcement Learning Aproach to Autonomic Resource Allocation. In *International Conference on Autonomic Computing (ICAC)*, 2006.

[35] E. Thereska, A. Donnelly, and D. Narayanan. Sierra: A Power-Proportional, Distributed storage System. Technical Report MSR-TR-2009-153, Microsoft, 2009.

[36] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. Dynamic Provisioning of Multi-tier Internet Applications. In *International Conference on Autonomic Computing (ICAC)*, 2005.

[37] V. V. Vazirani. *Approximation Algorithms*. Springer, 2003.

[38] L. Yin, S. Uttamchandani, M. Korupolu, K. Voruganti, and R. Katz. SMART: An Integrated Multi-Action Advisor for Storage Systems. In *USENIX Annual Technical Conference*, 2006.